Formulating Test Oracles via Anomaly Detection Techniques

Rafig Almaghairbe

Department of Computer and Information Sciences

University of Strathclyde

This dissertation is submitted for the degree of

Doctor of Philosophy

September 2017

I would like to dedicate this thesis to my father's soul, my mother, my beloved wife and lovely daughter.

Declaration

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

The copyright of this thesis belongs to the author under the terms of the United Kingdom Copyright Acts as qualified by University of Strathclyde Regulation 3.50. Due acknowledgement must always be made of the use of any material contained in, or derived from, this thesis.

Rafig Almaghairbe
September 2017

Acknowledgements

In the name of Allah, the most merciful, the most kind.

The Prophet Mohammed, peace and blessings be upon him, said, *“He has not thanked Allah who has not thanked people”*.

I would like to express my sincere gratitude to my supervisor Dr. Marc Roper for his endless encouragement and support during my research and writing of this thesis. For his patience, motivation, enthusiasm, and immense knowledge. He was always there to listen with care to my problems as if they were his own and answer all my questions with a smile. Thank you for inspiring me to challenge myself and be a better scientist.

I would like also to thank all my colleagues and staff in the Department of Computer and Information Sciences at the University of Strathclyde for their help, encouragement and comments. Thank you for providing such a friendly, dynamic and supportive research environment. I would particularly like to acknowledge all the anonymous reviewers whose feedback guided the papers the thesis is based on, and the other researchers whose conversations and questions about my work have helped to shape it. I would especially like to thank all those developers who contributed to the open-source software that this thesis relies on, both as subject matter and to assist with analysis. Funding for this project was provided by the Ministry of Higher Education of Libya, and for that I am thankful.

"Good friends are like stars, you do not always see them, but you know they are always there". I thank all of my friends for their encouragement and being there whenever I need help.

I have been blessed with an incredibly supportive family; thank you to my lovely parents and sisters for encouraging me to follow my dreams. Special thanks to all of my brothers in law Yousif El Gumati, Ahmed Habel and Ramadan Bozaid for their well wishes and being there whenever I need help.

My sincere thanks also go to my wife’s family for their encouragement and support given to me over many years.

I am in debt to my beloved wife Keria, and lovely daughter Fawzia for their unlimited love and strong support. I will be grateful forever for your love.

Finally, the ability and motivation required to complete this thesis are gift from Allah, and for that I will always be grateful. I would like to end these acknowledgements with a quote from 'Quran'

"My success can only come from Allah: in Him I trust and unto Him I turn".

Abstract

Developments in the automation of test data generation have greatly improved efficiency of the software testing process but the so-called "oracle problem" (deciding the pass or fail outcome of a test execution) is still primarily an expensive and error-prone manual activity. This thesis presents an approach to build an automated test oracle using anomaly detection techniques (based on semi-supervised and unsupervised learning approaches) on dynamic execution data (test input/output pairs and execution traces).

Firstly, anomaly detection techniques based on semi-supervised learning approach were investigated to automatically classify passing and failing executions. A small proportion of the test data is labelled as passing or failing and used in conjunction with the unlabelled data to build a classifier which labels the remaining outputs (classify them as passing or failing tests). A range of learning algorithms are investigated using several faulty versions of three systems along with varying types of data (inputs/outputs alone, or in combination with execution traces) and different labelling strategies (both failing and passing tests, and passing tests alone). The results show that in many cases labelling just a small proportion of the test cases – as low as 10% – is sufficient to build a classifier that is able to correctly categorise the large majority of the remaining test cases. This has important practical potential: when checking the test results from a system a developer need only examine a small proportion of these and use this information to train a learning algorithm to automatically classify the remainder.

Secondly, anomaly detection techniques based on unsupervised learning (mainly clustering algorithms) were investigated to automatically detect passing and failing executions. The key hypothesis is that failures will group into small clusters whereas passing executions will group into larger ones. In this investigation, the same dynamic execution data and systems used in previous study were used to evaluate the proposed approach. The results show that this hypothesis to be valid, and illustrates that the approach has the potential to substantially reduce the numbers of outputs that would need to be manually examined following a test run.

Finally, a comparison study was performed between existing techniques from the specifications mining domain (the data invariant detector Daikon [30]) and anomaly detection techniques (based on semi-supervised and unsupervised learning approaches). In most cases

semi-supervised learning techniques (mainly Self-training approach - Naïve Bayes with EM clustering algorithm - and Co-training approach - Naïve Bayes) perform far better under both scenarios (two different labelling strategies) as an automated test classifier than Daikon especially when input/output pairs are used together with execution traces. Furthermore, unsupervised learning techniques performed on a par when compared with Daikon in several cases.

Table of contents

List of figures xii

List of tables xv

1 Introduction 1

1.1 Problem Statement and Motivation 1
1.2 Research Questions and Goals 2
1.3 Contribution of the Thesis 3
1.4 Research Methodology Outline 4
1.5 Publications 5
1.6 Thesis Outline 6

2 Background and Related Work on Automated Test Oracles 8

2.1 Introduction 8
2.2 Anomaly Detection Background 9
2.3 Software Testing Background 13
2.3.1 Software Testing Concepts 14
2.3.2 Test Oracles 17
2.4 Related Work on Automated Test Oracles 20
2.4.1 Test Oracles Based on Anomaly Detection Techniques 23
2.4.2 Test Oracles Based on Invariant Detection 37
2.5 Discussion 40
2.6 Conclusions 41

3 Automatically Classifying Test Results by Semi-Supervised Learning 44

3.1 Introduction 44
3.2 Methodology 45
3.2.1 Semi-Supervised Learning 45
3.2.2 Semi-Supervised Learning Algorithms 46

3.3 Experiment Design 49
3.3.1 Subject Programs 49
3.3.2 Experiment Set-up 50
3.3.3 Evaluation 57
3.3.4 Comparison with Daikon 58
3.3.5 Tools and Configuration 59
3.4 Experimental Results and Discussion 60
3.4.1 Study 1: Test Result Classification Based on Input/Output Pairs . . 60
3.4.2 Study 2: Test Result Classification Based on Input/Output Pairs Augmented with Execution Traces 62
3.5 Statistical Test for semi-supervised learning Hypothesis 75
3.6 Semi-supervised Learning Techniques versus Daikon 75
3.7 Discussion 78
3.8 Conclusions 81

4 Separating Passing and Failing Test Executions by Clustering Anomalies 82
4.1 Introduction 82
4.2 Methodology 83
4.2.1 Clustering Analysis 83
4.2.2 Clustering Algorithms 84
4.2.3 (Dis)similarity Measures 86
4.2.4 Linkage Metrics 86
4.2.5 Number of Clusters 87
4.2.6 Small Cluster Size 87
4.3 Experimental Evaluation 88
4.3.1 Experimental Set-up 88
4.3.2 Evaluation of Clustering Techniques 89
4.4 Experiment 1 (Clustering Test Input/Output Pairs): Results And Discussion 91
4.4.1 Distribution of Failures 91
4.4.2 Failures Found verses Cluster Counts and Cluster Sizes 91
4.4.3 Failure Density of Smallest Clusters 102
4.5 Experiment 2 (Clustering Test Input/Output Pairs and Execution Traces): Results And Discussion 103
4.5.1 Distribution of Failures over Clusters 103
4.5.2 Failure Composition of Small Clusters 105
4.5.3 Fault Density of Smallest Clusters 111
4.5.4 Impact of Failure Density 116

4.6 Statistical Test for Clustering Hypothesis 120
4.7 Clustering Algorithms versus Daikon 120
4.8 Discussion 123
4.9 Conclusion 124

5 Conclusions and Future Work 126
5.1 Summary of the Thesis 126
5.2 Contributions of the Thesis 129
5.3 Threats to Validity 130
5.4 Future Work 131
5.5 Closing remarks 133

References 134

Appendix A Fault Details for All Systems 151

Appendix B An Example to Explain the Encoding Scheme in More Detail 159

Appendix C Trace Compression Algorithm Developed by Nguyen et al. [69] 163

Appendix D A Cross Validation Results for All Experiments 166

Appendix E A Thesis Data 184

List of figures

2.1 Generic Test Oracle Structure from [72]. 18

2.2 Test Oracles Structure Using Expected Output Behaviors from [72]. 19

2.3 Test Oracles Structure Using Formal Model Specification from [72]. 19

2.4 Test Oracles Structure Using Test Data from [72]. 20

2.5 Test Oracles Structure Using Human Oracles from [72]. 20

2.6 Principle of Anomaly Detection from [18]. 24

3.1 The Average F-measure (F) Values vs. Labelled Data Size for Self-Training (EM-Naïve) Using Input/Output Pairs and Training on Normal and Abnormal Cases 65

3.2 The Average F-measure (F) Values vs. Labelled Data Size for Co-Training (Co-Naïve) Using Input/Output Pairs and Training on Normal and Abnormal Cases 65

3.3 The Average F-measure (F) Values vs. Labelled Data Size for Self-Training (EM-Naïve) Using Input/Output Pairs Augmented with Execution Traces and Training on Normal and Abnormal Cases 67

3.4 The Average F-measure (F) Values vs. Labelled Data Size for Co-Training (Co-Naïve) Using Input/Output Pairs Augmented with Execution Traces and Training on Normal and Abnormal Cases 70

3.5 The Average F-measure (F) Values vs. Labelled Data Size for Self-Training (EM-Naïve) Using Input/Output Pairs Augmented with Execution Traces and Training on Normal Cases Alone 74

3.6 The Average F-measure (F) Values vs. Labelled Data Size for Co-Training (Co-Naïve) Using Input/Output Pairs Augmented with Execution Traces and Training on Normal Cases Alone 74

4.1 Evaluation Example 90

4.2 Hierarchical Clustering Algorithm with Average Linkage for NanoXML (Version 1) Using Input/Output Pairs only. 92
4.3 Hierarchical Clustering Algorithm with Average Linkage for NanoXML (Version 2) Input/Output Pairs only. 92
4.4 Hierarchical Clustering Algorithm with Average Linkage for NanoXML (Version 3) Input/Output Pairs only. 93
4.5 Hierarchical Clustering Algorithm with Average Linkage for NanoXML (Version 5) Input/Output Pairs only. 93
4.6 Hierarchical Clustering Algorithm with Average Linkage for Siena (Version 2) Input/Output Pairs only. 94
4.7 Hierarchical Clustering Algorithm with Average Linkage for Sed (Version 5) Input/Output Pairs Only. 94
4.8 Percentage of Failures Found Over the Smallest Clusters for all NanoXML Versions Using Single Linkage and Input/Output Pairs Only. 99
4.9 Percentage of Failures Found Over the Smallest Clusters for all NanoXML Versions Using Average Linkage and Input/Output Pairs Only. 100
4.10 Percentage of Failures Found Over the Smallest Clusters for all NanoXML Versions Using Complete Linkage and Input/Output Pairs Only. 100
4.11 Percentage of Failures Found Over the Smallest Clusters for Siena Version Using Linkage Metrics and Input/Output Pairs Only. 101
4.12 Percentage of Failures Found Over the Smallest Clusters for Sed Version Using Linkage Metrics and Input/Output Pairs Only. 101
4.13 Hierarchical Clustering Algorithm with Single Linkage for NanoXML (Version 1) Using Input/Output Pairs Augmented with Execution Traces. 106
4.14 Hierarchical Clustering Algorithm with Single Linkage for NanoXML (Version 2) Using Input/Output Pairs Augmented with Execution Traces. 106
4.15 Hierarchical Clustering Algorithm with Single Linkage for NanoXML (Version 3) Using Input/Output Pairs Augmented with Execution Traces. 107
4.16 Hierarchical Clustering Algorithm with Single Linkage for NanoXML (Version 5) Using Input/Output Pairs Augmented with Execution Traces. 107
4.17 Hierarchical Clustering Algorithm with Single Linkage for Siena (Version 2) Using Input/Output Pairs Augmented with Execution Traces. 108
4.18 Hierarchical Clustering Algorithm with Average Linkage for Sed (Version 5) Using Input/Output Pairs Augmented with Execution Traces. 108

B.1 Trace Generated by Daikon 160
B.2 Extracted Sequences 160

B.3 Hash Key for the Collections of Method Sequence 161
B.4 The Final Trace Representation 162

List of tables

3.1 Example Coding of Input/Output Pairs 53

3.2 Example Coding of Sequence Traces 54

3.3 Labelled training data set sizes and class distribution for NanoXML (all versions) for scenario 1 55

3.4 Labelled training data set sizes and class distribution for Siena (all versions) for scenario 1 55

3.5 Labelled training data set sizes and class distribution for Sed (version 5) for scenario 1 55

3.6 Abnormally Labelled data items for NanoXML studies using scenario 1 . . 56

3.7 Abnormally Labelled data items for Sed studies using scenario 1 56

3.8 Domain size for the three systems 56

3.9 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs and Training on Normal and Abnormal Cases 63

3.10 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs and Training on Normal and Abnormal Cases 64

3.11 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs Augmented with Execution Traces and Training on Normal and Abnormal Cases 68

3.12 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs Augmented with Execution Traces and Training on Normal and Abnormal Cases 69

3.13 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs Augmented with Execution Traces and Training on Normal Cases Alone 72
3.14 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs Augmented with Execution Traces and Training on Normal Cases Alone 73
3.15 Random Tested F-measure from Several Experiments 76
3.16 The Results of Z-Test and p-value on Tested F-measure 76
3.17 Daikon Versus Semi-supervised Learning Techniques 79
3.18 Daikon Versus Semi-supervised Learning Techniques 80

4.1 Recall (Failures Found) and F-measure vs. Cluster Size for Hierarchical Clustering with Different Linkage Metrics Using Input/Output Pairs Only. . 97
4.2 Recall (Failures Found) and F-measure vs. Cluster Size for Hierarchical Clustering with Different Linkage Metrics Using Input/Output Pairs Only. . 98
4.3 Percentage of Failures and F-measure vs. Cluster Size for EM clustering Algorithm Using Input/Output Pairs Only. 99
4.4 Percentage of Failures and F-measure vs. Cluster Size for DBSCAN clustering Algorithm Using Input/Output Pairs Only. Note for NanoXML (Epsilon = 0.9 Minpoints = 2) and for Siena and Sed (Epsilon = 1.5 Minpoints = 1). . 99
4.5 Failure Distribution over less than Average Sized Clusters for Nanoxml Using Input/Output Pairs Only. 104
4.6 Failure Distribution over less than Average Sized Clusters for Sed Version 5 Using Input/Output Pairs Only. 104
4.7 Percentage of Failures Found (Recall) and F-measure vs. Cluster Size for Hierarchical Clustering with Different Linkage Metrics Using Input/Output Pairs Augmented with Execution Traces. 112
4.8 Percentage of Failures Found (Recall) and F-measure vs. Cluster Size for Hierarchical Clustering with Different Linkage Metrics Using Input/Output Pairs Augmented with Execution Traces. 113
4.9 Percentage of Failures Found (Recall) and F-measure vs. Cluster Size for Hierarchical Clustering with Different Linkage Metrics Using Input/Output Pairs Augmented with Execution Traces. 114
4.10 Percentage of Failures Found (Recall) and F-measure vs. Cluster Size for DBSCAN clustering Algorithm Using Input/Output Pairs Augmented with Execution Traces. 115

4.11 Percentage of Failures Found (Recall) and F-measure vs. Cluster Size for EM clustering Algorithm Using Input/Output Pairs Augmented with Execution Traces. 115
4.12 Failure Distribution over less than Average Sized Clusters for Nanoxml . . 117
4.13 Failure Distribution over less than Average Sized Clusters for Siena 118
4.14 Failure Distribution over less than Average Sized Clusters for Sed Version 5 118
4.15 NanoXML V3 with Reduced Failure Rate 119
4.16 Siena with Reduced Failure Rate 119
4.17 Tested Proportion of Failures Found on the Smallest Clusters on Several Experiments 121
4.18 The Results of Z-Test and p-value on Tested Failures Proportion 121
4.19 F-measure for Failures Found for Clustering Algorithms Using Input/Output Pairs Versus Daikon in Nanoxml. 123
4.20 F-measure for Failures Found for Clustering Algorithms Using Input/Output Pairs Versus Daikon in Siena Version 2 and Sed Version 5. 123
4.21 F-measure for Failures Found for Clustering Algorithms Using Input/Output Pairs Augmented with Execution Traces Versus Daikon in Nanoxml. 124
4.22 F-measure for Failures Found for Clustering Algorithms Using Input/Output Pairs Augmented with Execution Traces Versus Daikon in Siena Version 2 and Sed Version 5. 124

D.1 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using self-training (EM-Naïve) 167
D.2 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 1 using self-training (EM-Naïve) 167
D.3 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 1 using self-training (EM-Naïve) 168
D.4 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 1 using self-training (EM-Naïve) 168
D.5 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using co-training (Co-Naïve) 168
D.6 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 1 using co-training (Co-Naïve) 169
D.7 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 1 using co-training (Co-Naïve) 169
D.8 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 1 using co-training (Co-Naïve) 169

D.9 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using Co-EM (EM-SVM) 170
D.10 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using Co-training (Co-SVM) 170
D.11 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 1 using Self-training (EM-Naïve) 170
D.12 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 1 using Self-training (EM-Naïve) . 171
D.13 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 1 using Co-training (Co-Naïve) . 171
D.14 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 1 using Co-training (Co-Naïve) . . 172
D.15 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using self-training (EM-Naïve) 172
D.16 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 1 using self-training (EM-Naïve) 173
D.17 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 1 using self-training (EM-Naïve) 173
D.18 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 1 using self-training (EM-Naïve) 173
D.19 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using Co-training (Co-Naïve) 174
D.20 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 1 using Co-training (Co-Naïve) 174
D.21 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 1 using Co-training (Co-Naïve) 174
D.22 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 1 using Co-training (Co-Naïve) 175
D.23 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 1 using Co-training (Co-SVM) 175
D.24 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 1 using self-training (EM-Naïve) . 176
D.25 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 1 using Self-training (EM-Naïve) . 176
D.26 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 1 using Co-training (Co-Naïve) . 177

D.27 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 1 using Co-training (Co-Naïve) . . 177
D.28 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 1 using Co-training (Co-SVM) . . 178
D.29 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 1 using Co-training (Co-SVM) . . . 178
D.30 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 2 using self-training (EM-Naïve) 179
D.31 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 2 using self-training (EM-Naïve) 179
D.32 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 2 using self-training (EM-Naïve) 179
D.33 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 2 using self-training (EM-Naïve) 180
D.34 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 2 using Co-training (Co-Naïve) 180
D.35 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 2 using Co-training (Co-Naïve) 180
D.36 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 2 using Co-training (Co-Naïve) 181
D.37 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 2 using Self-training (EM-Naïve) 181
D.38 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 2 using Self-training (EM-Naïve) . 182
D.39 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 2 using Co-training (Co-Naïve) . 182
D.40 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 2 using Co-training (Co-Naïve) . . 183

Chapter 1

Introduction

1.1 Problem Statement and Motivation

The importance of testing, and the consequences of poor or inadequate testing to software development projects are only too well understood. The National Institute of Standards Technology found that faulty software cost the US economy about $60 billion every year [99]. This figure aside, many companies spend over 50% of a software development on testing and related activities and this figure rises to 90% in certain critical applications [66]. In recent years, the rise in popularity of the Xunit suite and initiatives such as test first development have done much to raise the profile of the topic, but such approaches still demand a lot from the developer in that test cases need to be hand coded and acceptable results clearly specified.

Pex [38] tool is an example of the advances that have been made in the area of test data generation which represents a step change in testing technology. With such a tool the developer can save a significant amount of time and effort by being able to take an arbitrary system and automatically generate test data that will exercise several paths of the system. However, there is a missing essential component which is a mechanism for automatically determining whether the output associated with a particular input is correct or not and therefore indicative of a fault in the code. This mechanisms is known as test oracle. If such oracle is missing then the developer needs to check the output to determine its correctness or otherwise but this task can be tedious and time consuming for developer especially in the

case of automatically generated data for large systems. Although not entirely mythical, test oracles are fairly rare beasts.. Some systems may have accurate, complete and up-to-date machine readable specifications, others may be well populated by contracts, and in such cases both of these can perform the role of an oracle. However, the vast majority of systems lack such provisions, and the massive benefits that can be gained from using test data generation tools are outweighed by the need to check the results by hand.

Existing approaches to generate oracles range from the inexpensive and ineffective to the effective but very costly. At one end of the scale, specified oracles can be generated from formal specifications [9], and are effective in identifying failures, but defining and maintaining such specifications is demanding and consequently such specifications are very rare. At the other end, implicit oracles are easy to obtain at practically no cost but are not able to identify semantic and complex failures, revealing only general errors like system crashes or unhandled exceptions [9, 74]. The goal of this thesis is to strike a balance between these approaches and develop a technique which combines the effectiveness of a specified oracle and the cost of an implicit one by using anomaly detection detection approaches (machine learning, data mining etc.) to automatically identify failing tests. The research will be focused on building several models of test oracles by using various kinds of software data such as input/output pairs alone or in combination with execution traces.

1.2 Research Questions and Goals

The overall aim of this work is to investigate and evaluate the feasibility of the idea of using anomaly detection to detect software bugs by formulating test oracles. To achieve this aim, the following research questions are investigated in this research:

1. Which of the variety of anomaly detection approaches (machine learning, data mining etc.) is most appropriate for the test oracle problem?

2. Which anomaly detection strategies (classification, clustering etc.) are most effective for the test oracle problem?

3. What data from a system provides the anomaly detection approach with the best chance of building an effective oracle?

The research goals of this thesis are identified below:

- Identification of the anomaly detection approaches that are appropriate and effective for building test oracles. This goal will be achieved by performing a set of experiments with a range of clustering and classification approaches.
- Identification, again via experimentation, of the most useful combinations of data to feed anomaly detection algorithms.

1.3 Contribution of the Thesis

The work presented in this thesis makes the following contributions to the area of test oracle generation techniques:

- An experimental investigation and evaluation into the use of semi-supervised learning anomaly detection techniques to automatically classify passing and failing tests.
- An experimental investigation and evaluation into the use of a range of clustering-based anomaly detection techniques to support the construction of a test oracle.
- An experimental investigation of the most useful combinations of execution data (input/output pairs and execution traces).
- A comparison study between the proposed approaches in this thesis and the Diakon specification mining tool.

Key findings

- The results from semi-supervised learning study suggested that the approach is applicable to automatically build an oracle. It was found that in many cases labelling just small proportion of the test cases as low as 10% is sufficient to build a classifier

that is able to classify correctly the large majority of the remaining test cases. The results has pointed out to important practical potential: when checking the test results from a system a developer need only examine a small proportion of these and use this information to train a learning algorithm to automatically classify the remainder.

- The results from unsupervised learning study suggested that the approach is applicable to automatically build an oracle. The results showed that the key clustering hypothesis (failures will group into small clusters, whereas passing executions will group into larger ones) is valid - in many cases small clusters were composed of at least 60% failures (and often more). The results illustrated that the approach has the potential to substantially reduce the numbers of outputs that would need to be manually examined following a test run.

- The results from the comparison study showed that the proposed approaches (anomaly detection based on semi-supervised and unsupervised learning techniques) did perform well in comparison to Daikon especially when input/output pairs augmented with execution traces. It must be stressed that Daikon requires a fault free version of the system under test with complete and large test suite from which to build assertions - a luxury that the proposed approaches did not require.

1.4 Research Methodology Outline

The thesis reviews the available published papers that have used anomaly detection techniques along with dynamic execution data in the area of test oracles. The aim is to identify the most commonly used anomaly detection approaches to date for building test oracles along with various types of dynamic execution data that have been employed, and also to forms a basis for comparison with the proposed work in this thesis.

It turns out that anomaly detection techniques based on semi-supervised and unsupervised learning have not been extensively investigated to construct an automated test oracles. Therefore, the thesis starts by investigating and evaluating the use of anomaly detection

techniques based on semi-supervised learning in this context. The thesis investigates several learning algorithms with two real practical scenarios and two types of dynamic execution data (input/output pairs and execution traces). The thesis also observes the potential impact of using different sizes of labelled data.

Then, the thesis investigates and evaluates the use of anomaly detection techniques based on unsupervised learning to automatically construct a test oracle. The thesis investigates several clustering algorithms with their main hypothesis (*"Normal data instances belong to large and dense clusters, while anomalies either belong to small or sparse clusters"*[18]), and also again with similar types of dynamic execution data as in the previous investigation. The thesis explores the optimal number of clusters to employ in relation to the system domain to generate an effective oracle.

Finally, a comparison study with the Daikon specification mining tool plays the role of evaluation to anomaly detection techniques based semi-supervised and unsupervised learning as test oracles.

1.5 Publications

Portions of this thesis have previously been published in the following papers:

- Rafig Almaghairbe and Marc Roper, “Anomaly detection techniques for automated software fault detection via dynamic execution data”, Doctoral Symposium, 29th IEEE International Conference on Software Maintenance (ICSM), 2013 (Reviewed, accepted for presentation in the doctoral symposium track and as a poster in the conference poster session, but not formally published).

- Rafig Almaghairbe and Marc Roper, “Building Test Oracles by Clustering Failures”, in Proceedings of the 10th International Workshop on Automation of Software Test (AST@ICSE), 2015

- Rafig Almaghairbe and Marc Roper, "Automatically Classifying Test Results by Semi-Supervised Learning", in Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE), 2016

- Rafig Almaghairbe and Marc Roper, "Separating Passing and Failing Test Executions by Clustering Anomalies", Accepted for publication on Software Quality Journal (A Special Issue on Automation of Software Test), 2016

1.6 Thesis Outline

The reminder of the thesis is structured in the following way:

- Chapter 2 introduces an overview of anomaly detection techniques, software testing principles and the concept of test oracles that underpin the subject of this thesis, and then follows with a classification of the exiting techniques and related work. The related work is narrowed down to include only the most relevant papers to the subject of this thesis.

- Chapter 3 presents an approach to classifying passing and failing execution data using semi-supervised learning techniques on dynamic execution data based on firstly, just a system's input/output pairs and secondly, amalgamations of input/output pairs and execution traces. In the experimental evaluation, a small proportion of the test data is labelled by the developer as passing or failing along with a significant amount of unlabelled test data and the learning algorithms use this to build a classifier which is then used to label each remaining element (i.e. classify it as being either a passing or failing test). A range of learning algorithms are investigated using several faulty versions of three systems and different labelling scenarios (both failing and passing tests, and just passing tests alone). A comparison study between the proposed approach and the Diakon specification mining tool is performed.

- Chapter 4 presents an approach using cluster based anomaly detection on dynamic execution data. In the experimental evaluation, two different studies are performed.

In the first study, a range of clustering algorithms are applied to just the test case input/output pairs of three systems and the effectiveness of this approach is evaluated. In the second study, the test case input/output pairs are augmented with their associated execution traces with the aim of improving the accuracy of the approach. A comparison study between the proposed approache and the Diakon specification mining tool is performed.

- Chapter 5 summarizes this thesis and proposes some directions to be explored in future work.

Chapter 2

Background and Related Work on Automated Test Oracles

2.1 Introduction

Research in software testing has focused on automating many aspects of the testing process such as generating and executing test cases and maintaining and managing test suites. A relatively neglected, but essential, aspect of testing is the production of an oracle: a mechanism to determine the (in)correctness of an output associated with an input. Whilst there are tools capable of completely automatically generating test inputs [32], few techniques exist to generate test oracles, making the process of checking test outputs primarily human-centred and consequently expensive and error prone [8].

The early section of this chapter will give a general view of anomaly detection (section 2.2). Software testing concepts and the concept of test oracles in the testing scenario will be presented next (subsection 2.3.1 and 2.3.2). The following section will discuss the related work that addresses the test oracle problem using anomaly detection (mainly clustering and machine learning techniques), and also it will identify the gaps and the limitations on those works (section 2.4).

2.2 Anomaly Detection Background

Anomaly detection is a general set of strategies that can be used to detect unusual values or outliers in large data sets. It has been employed successfully in various research areas such as fraud detection for credit cards, insurance for health care, intrusion detection for cyber-security, fault detection in safety critical systems, and military surveillance for enemy activities.

Mitchell described the concept of learning as "acquiring the definition of a general category given a sample of positive and negative training examples of the category" (Mitchell, 1997). According to the learning problem definition, there are three main components to consider which are:

- The training experience *(E)* related to the training data sets from which the system will learn.
- The class of tasks *(T)* related to the definition of the target function that determines the types of knowledge that will be learnt.
- The performance measure *(P)* of the knowledge that is acquired in the process.

Anomaly detection techniques can be classified to three broad categories based on learning process:

- *Supervised Learning:* Techniques under this category assume the availability of a training data set which has labelled instances for normal as well as anomaly classes and is therefore the least generally applicable. The typical approach in such cases is to build a predictive model for normal against anomaly classes. Any unseen data instance is compared against the model to determine which class it belongs to. There are two major issues that arise in supervised anomaly detection: (1) the anomalous instances are far fewer compared to the normal instances in the training data (imbalanced class distribution); (2) obtaining accurate and representative labels, especially for the anomaly class is usually challenging. The principal tasks associated with this kind of learning are Classification, Regression and Prediction.

- *Semi-supervised Learning:* Techniques operating under this category assume that training data has a small proportion of labelled instances for both normal and abnormal classes which may be used in conjunction with the unlabelled instances to build a model which labels the remaining instances. Since they do not require a large set of labelled training instances, they are more widely applicable than supervised learning techniques.

- *Unsupervised Learning:* Techniques that operate in unsupervised mode do not require training data, and thus are most widely applicable. The techniques in this category make the implicit assumption that normal instances are far more frequent than anomalies in the test data. If this assumption is not true then such techniques suffer from a high false alarm rate. The principal tasks associated with this kind of learning are Clustering and Association Rules.

Anomaly detection tasks require general steps which have to be performed for successful pattern recognition. This mainly includes collecting data (variables or features), performing feature selection (for example removing irrelevant and redundant features), choosing the right learning algorithm (for example evaluating several alternatives), training the classifier or model, and finally evaluating the performance of the classifier (usually performed on a separate test set).

Feature selection is a critical step in the classification process. With a large data set and high dimensional feature vectors, it would be expected that the classifier would perform poorly due to the redundant and irrelevant features present in the training set. But, by selecting features that are invariant to irrelevant transformation, insensitive to noise and highly discriminatory then we could expect to achieve a more successful pattern recognition model.

The choice of a learning algorithm is also an important step. For example, some methods such as Support Vector Machines (SVM) are very flexible and able to deal with high dimensionality. Some learning algorithms are severely affected by the imbalanced training data sets problem (imbalanced training data sets means that one class is represented by a large

number of examples while the other is represented by only a few) such as SVM and Decision Trees (DT) whereas others like Naïve Bayes (NB) are not. Some learning algorithms produce human readable results, while others are “black boxes". SVM is an example of a black box approach. However, they are often highly accurate in their results, particularly on continuous real-valued numeric data.

After training a classifier, the classifier performance is measured by applying an evaluation procedure. Many statistical and other measurements exist in Data Mining and Machine Learning areas. One problem that might affect the evaluation procedure is over fitting. This arises when a classifier allows for perfect classification on the training data while performing poorly on a new data set (test data). The common approach to solve such problem is to provide an independent test data set called validation set. This approach is commonly known as cross validation - a statistical method of evaluating and comparing learning algorithms by dividing data into two data sets; one used to learn or train a model and the other used to validate the model. However, this is most likely to happen when a large amount of data is available. On a smaller amount of data, holding out a large enough independent test set may result in not enough data being available for training. In this case the cross validation procedure will be the common solution.

Several anomaly detection techniques have been proposed in literature. Chandola et al. [18] discussed in details the applications of anomaly detection techniques. Therefore, the most popular techniques that have been used in other areas (such as intrusion detection and fraud detection etc.) are discussed briefly below. For further details on the techniques the reader is referred to the work of Bishop [14], Mitchell [65], Witten and Frank [103] and Larose [53] for example.

- *Decision Trees (DT):* is a supervised learning technique that uses approximating discrete functions to estimate and classify the example. The nodes of trees are attributes and the leaves are values of discrete function. The decision tree can be rewritten in a set of “if-then” rules and also give an estimation of the probability of occurrence of a particular case. This is an inductive learning method which is very popular and mostly used for variety of classification tasks [65].

- *Naïve Bayes (NB):* is a simple probabilistic classifier based on applying Bayes theorem with strong (naïve) independence assumptions. A more descriptive term for the underlying probability model would be an "independent feature model". In simple terms, a Naïve Bayes classifier assumes that the presence (or absence) of a particular feature of a class is unrelated to the presence (or absence) of any other feature. Depending on the precise nature of the probability model, Naïve Bayes classifiers can be trained very efficiently in a supervised learning setting. In spite of their Naïve design and apparently over-simplified assumptions, Naïve Bayes classifiers often perform much better in many complex real world situations than one might expect [65].

- *K-Nearest Neighbour (K-NN):* is the simplest method in machine learning techniques. The main idea of (K-NN) is to find the closest *(k)* points of the training data to the test data point, and then give a label to the test data point by a majority vote between the *(k)* points. This method is very simple and remarkably has a low classification error. However, it requires a large memory to store the training data, and also it is computationally expensive [65].

- *Artificial Neural Network (ANN):* is a computer system that simulates the learning process of the human brain. ANN is massively parallel systems inspired by the architecture of biological neural networks, comprising simple interconnected units (artificial neurons). Neurons compute a weighted sum of their input and generate an output if the sum exceeds a certain threshold. This output then becomes an excitatory (positive) or inhibitory (negative) input to other neurons in the network. The process continues until one or more outputs is generated [65].

- *Support Vector Machines (SVM):* is a supervised learning method, mainly applied to classification and regression problems. The main idea of SVM is to separate classes with a surface that maximizes the margins between them. This method combines two main ideas. The first is the concept of an optimum linear margin classifier that constructs a separating hyperplane that maximizes distances to the training point. This hyperplane is supported by some of these training points. The second is the

concept of a kernel which is a function that calculates the dot product of two training vectors. Kernels calculate these dot products in feature space. When using feature transformation, which reformulates input vectors into new features, the dot product is calculated in feature space, even if the new feature space has higher dimensionally. The linear classifier is unaffected. Note that different kernel functions can be used with SVM algorithm to solve nonlinear classification problems such as a quadratic kernel function, polynomial kernel function, Gaussian radial basis kernel function and multilayer perceptron kernel function [103].

- *Clustering:* seeks to segment the entire data into relatively homogeneous subgroups or clusters, where the similarity of the records within the cluster is maximized, and the similarity to records outside this cluster is minimized. The clustering differs from classification in that there is no target variable for clusters. The clustering task does not try to classify, estimate, or predict the value of a target variable. Hierarchical clustering and k-Means clustering methods are the most common types of clustering techniques [53].

- *Association Rules:* Association is the job of finding which attributes “go together”. The technique of association rules is commonly used in the business world, where it is known as affinity analysis or market basket analysis. The task of association seeks to uncover rules for quantifying the relationship between two or more attributes. Association rules are of the form “if antecedent, then consequent”, together with a measure of the support and confidence associated with the rule [53].

2.3 Software Testing Background

Nowadays software developers have access to advanced resources to support software development but programmers can make mistakes during the process of writing code (the implementation phase in the software development life cycle). Following the standard definition [6], an "*error*" is a mistake made by a human that leads to a fault that may result

in a failure. A "*fault*" occurs during the execution of software and results in an incorrect state that may or may not lead to a failure. A "*failure*" is a deviation between the observed behavior and the required behavior of a software system. The main focus of this chapter is on test oracles which refers to the means for determining whether the current results agree with the expected outcomes in software testing scenario: i.e. identifying failures. Therefore, this section introduces a wide view of software testing concepts and the concept of test oracles.

2.3.1 Software Testing Concepts

Researchers in the software engineering *(SE)* area proposed different useful definitions to describe software testing. Rafael et al. [72] define software testing as a way to verify and validate whether the software under test *(SUT)* behaves according to the software specification in a controlled execution. Hunter and Strooper [47] point out that software testing is the most important resource to check software behaviours against its specification. According to Bertolino [10], [11], software testing can be defined as an execution process of developed software aiming to check whether it meets its specifications considering the environment in which it was designed. In addition, software testing can be recognised as a reference framework which allowing different associations with testing concepts: "*Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite execution domain, against the specified expected behavior*" [11].

Nardi identifies software testing concepts as the input domain, obtained output, test case and test set [67]. The *input domain* is the set of all input data that can be applied to the program, also called test data. *Obtained output* is the result of the program execution and *expected output* is the one that should be produced by executing the program (or part of it) according to a given input data. A *test case* is a pair formed by the input data and expected output. A *test set* represents all the test cases used during the software testing.

Most testing activities have shared the same testing criteria and techniques in order to mitigate effort to discover errors and to decrease testing costs [72]. "*Black Box*" and "*White Box*" are the two main criteria in software testing. Black box criteria set requirements for

a test set based upon external characteristics. A traditional black box criterion is correct performance on a specified number of randomly chosen data points[4, 83]. On the other hand, white box criteria set requirements for a test based upon coverage of internal components or elements associated with the software. Structural coverage and fault coverage are the most common white box criteria. A structural coverage criterion specifies the extent to which a given test exercises or covers structural components of the software such as statements, branches or dataflow chains. A fault coverage criterion specifies errors, faults or classes of faults that are prevented or covered if the given test is passed [4, 83]. Major software testing techniques can be defined as follow:

- *Functional Testing* is a family of black box testing criteria that generate tests based on specified properties (e.g. functional properties) of the software under test. An example of functional testing criterion is a special values test that generates data points at which correctly functioning software exhibits some special behavior [4, 83].

- *Random Testing* are black box tests based upon known or assumed probability distribution of inputs. Random inputs may be chosen from operational profiles, simulations, or by purely statistical means. Random testing is sometimes associated with development methodologies and can be used to predict statistical parameters of operational software systems [4, 83].

- *Structural Testing* is the simplest family of white box methods where structural testing criteria sets coverage thresholds for program components such as statements, decision-to-decision branches, control flow paths and dataflow chains (i.e. program segments between successive definitions and uses of specified variables). One hundred per cent statement coverage is frequently considered to be the lowest acceptable threshold criterion for an effective software test [4, 83].

- *Partition Testing* relates to white box methods where analysis of a program usually leads to partitions of the input space. For instance, a common partitioning scheme may identify all those input values that cause the program to execute the same control path. A partition testing method is used to select test data from the partitions. An example

of such method, domain analysis, produces a geometric model of the partitions which is used as a guide to test data selection [4, 83].

- *Mutation Testing* refers to a family of white box methods based upon fault coverage. The goal is to create test that distinguish the program being tested from mutant programs that contains faults or bugs. Variations have been developed to study fault propagation (e.g. relay testing), early fault detection (e.g. weak mutation testing), and test case generation (error-sensitive test case analysis) [4, 83].

- *Regression Testing* are used to retest the system after modifications that have been made to enhance functionality or remove faults. A regression test can be conducted as either black box or white box test. The critical factor in regression tests is the cost of the test so that regressions tests are typically organized to minimize the amount of retesting [4, 83].

- *Smoke Testing* is a type of software testing that comprises of a non-exhaustive set of tests that aim at ensuring that the most important functions work. The results of this testing is used to decide if a build is stable enough to proceed with further testing [4, 83].

The testing phase usually consists of several testing levels such as "*unit test*", "*integration test*", "*system testing*" and "*acceptance testing*". Each level has different goals and parts of the SUT that need to be examined. "*Unit test*" is the level that the main test focus is to verify the validity of smaller units of the SUT such as procedures and functions. On the other hand, "*integration test*" is the level which aims to assess the adequacy between different software units working together as a whole. Moreover, the test applied to detect faults at the system level is the"*system testing*". Finaly, "*acceptance testing*" is a level of software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery [11, 66, 67, 72].

Generally, by increasing the size and the complexity of the SUT, the costs of software testing activities will be increased as well [72, 88]. Despite the costs of software testing

activities, applying the appropriate software testing techniques and strategies is required at every stage of software development process. Consequently, software developers and testers have claimed that software testing activities take more than 50% of software development time [66, 72]. To solve the problems of cost and time related to the software testing phase, automated testing approaches are needed during different stages of the software development process [10, 72]. Automated software testing activities can make significant cost reduction to software development projects and are a key factor to help saving costs [4, 72].

In recent years, automated testing approaches have shown significant advancements in the area of software testing. Automated test oracles can be considered one of these advancements. Several automated test oracle approaches are proposed to provide more efficient and productive ways to check the outputs of the SUT with the aim of reducing human intervention and effort. Test oracle concepts and problems will be introduced in the remainder of this section.

2.3.2 Test Oracles

Oracles can be defined as a mechanism that determines and judges whether a system's test results have passed or failed [100]. This function can be exercised by a tester (human oracle), or by automated/semi-automated means. Shahamiri et al. [89] summarised the test oracle process in the following points: (1) generate expected outputs; (2) save the generated outputs; (3) execute the test cases; (4) compare expected and actual outputs; (5) decide if there is fault or not. Note that test case execution is not part of test oracle, but it is part of the oracle process.

Mao et al. [107] have characterised a perfect and complete automated test oracle as follows: (1) it should have source of information which makes it possible to produce a reliable and equivalent behavior to the SUT; (2) it should accept all entries for the specified system and always produce the correct result; (3) it should have the answers to the data which is actually used in the test.

A traditional and generic test oracle structure can be seen in Figure 2.1 [72]. In this scenario, the test oracle accesses the set of data needed to evaluate the correctness of the

test output. This set of data comes from the specification of the SUT and contains sufficient information for supporting the oracle's final decision. Figures 2.2 - 2.5 present several structures of test oracle functions using different sources of information [72]. It can be noticed from those figures that some test oracles require test cases in order to be structured. However, there are cases where test oracles are able to provide a test result based on test data (inputs) alone.

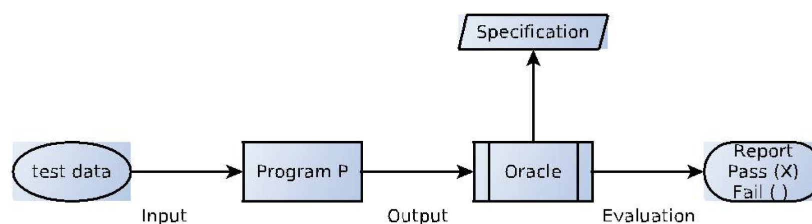

Fig. 2.1 Generic Test Oracle Structure from [72].

Figure 2.2 shows a test oracle based on a set of information and data about the expected results in order to decide the correctness of the SUT [39, 45, 48, 52, 61, 96]. The tester can implement this oracle by using one of the various frameworks known collectively as an "*xUnit*" family. These frameworks ("*xUnit*") allow the unit testing of SUTs implemented in different programming languages [49] for instance, "*JUnit*" is an "*xUnit*" framework for Java. Testers develop test oracles in their code by inserting a true/false statement (assertions) in a program to check unit or partial results. This oracle still demands a lot from the developer in that test cases need to be hand coded and acceptable results clearly specified. An example of such oracle is shown in the following piece of code [72].

```
1 public void testBOOKInLibrary ( ) {
2 Library library = new Library ( );
3 boolean search = library.checkByTitle ("Data Mining");
4 assertEquals (true, search); // A test oracle to check the
     correctness of the method "boolean
     Library.checkByTitle(String)"
5 }
```

Listing 2.1 Test Oracle Using the JUnit Framework

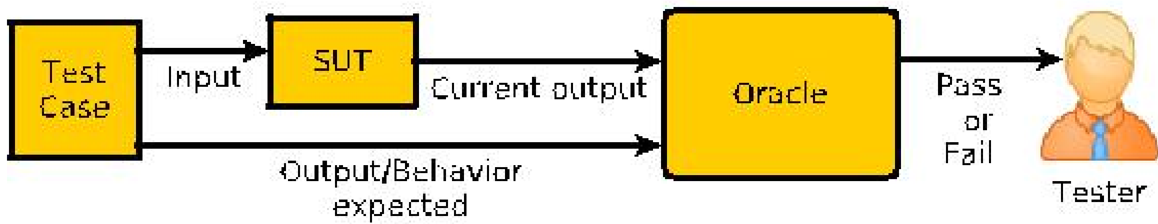

Fig. 2.2 Test Oracles Structure Using Expected Output Behaviors from [72].

Test oracles can be generated from formal models or specifications (Figure 2.3) [3, 5, 19, 28, 37]. In this scenario, test oracle can be automated when a mathematical model (e.g. *Finite State Machine (FSM) or Petri net*) of the SUT is available for testers. Test oracles based on formal models or specifications are effective in identifying failures, but defining and maintaining formal specifications is expensive to the point that such specifications are very rare.

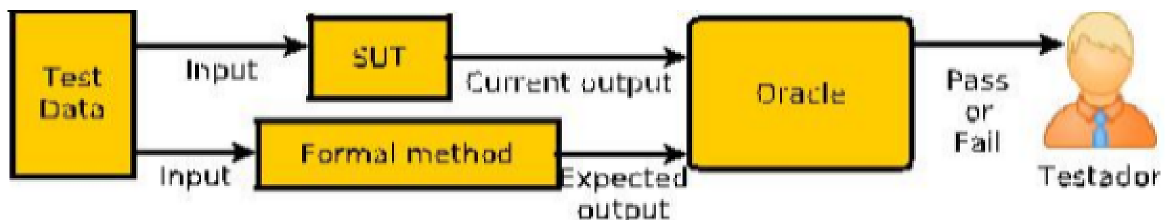

Fig. 2.3 Test Oracles Structure Using Formal Model Specification from [72].

Figure 2.4 illustrates test oracles that derive expected outputs of the SUT based on test data inputs [84, 85, 90, 92, 101]. This could be possible by using another version of the SUT to generate outputs from which the tester can build a test oracle to compare those outputs and the current outputs. In this case, testers must assume that the version used (reference program) meets all specifications of the SUT. This type of test oracle is widely used in the case of regression testing and mutation testing, but is not sufficient in the general case.

Finally, testers can use their own knowledge about the SUT to check if outputs meet the SUT specification, this test oracle is known as human oracle (Figure 2.5) [1, 46, 62, 95]. Human oracle suffers from several disadvantages: (1) it is error prone as a human oracle might make error in analysis; (2) it may be slower than the speed with which the program

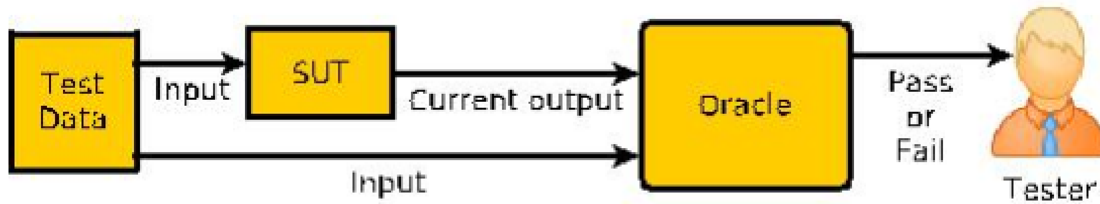

Fig. 2.4 Test Oracles Structure Using Test Data from [72].

computed the results; (3) it might result in the checking of only trivial input/output (I/O) behaviour.

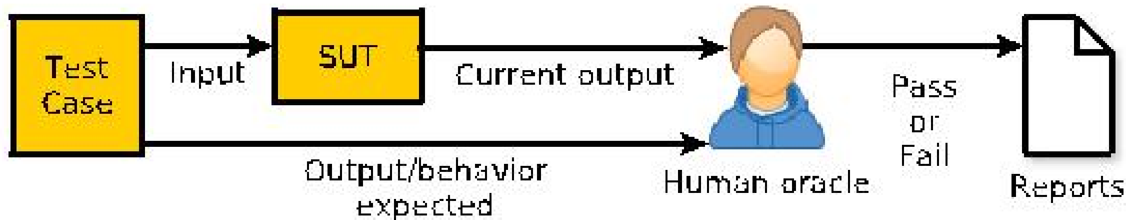

Fig. 2.5 Test Oracles Structure Using Human Oracles from [72].

The oracle problem usually occurs when it is difficult to interpret test results by testers [36]. In some cases, it is extremely difficult to predict expected behaviors of the SUT to be compared against current behaviors (this depends on the SUT) [23, 67]. Failures can be manifested under different circumstances which make checking the results complex or impossible to be performed [63]. Some SUTs produce outputs in very complex formats such as images, sounds or virtual environments which make the oracle problem very challenging [23]. There are several works that alleviate the oracle problem using specific techniques. This thesis is focused on building automated test oracles based on anomaly detection techniques, and previous work related to this is presented in the following section.

2.4 Related Work on Automated Test Oracles

The automatic generation of test oracles is an important problem in software testing area, but this problem has received considerably less attention compared to other testing problems such as the generation of test cases. There have been three extensive reviews of topics

relating to test oracles. The first review by Baresi and Young [8] covered four important topics in the test oracle area: assertions, specification, state-based conformance testing and log file analysis. The second review by Pezzè and Zhan [77] presented the main approaches to oracle generation according to the source of information. In their survey, the source of information for test oracles was classified either as the software specification or as a program code. Specifications were further classified according to the type of formal model into state-based specifications, transition-based specifications, history-based specifications and algebraic specifications. Code-based information includes values from other versions, results of program analysis, machine learning models and metamorphic relations. Finally, Barr et al. [9] proposed a comprehensive analysis and review of work on the oracle problem, and they classified the existing literature on test oracles into three broad categories: (1) specified oracles; (2) implicit oracles; (3) derived oracles.

Specified oracles are obtained from a formal specification of the system behaviour. Those oracles can be grouped into three categories: specification based languages, assertions and contracts, and algebraic specifications [9]. An example of a specified oracle is the work presented by Doong and Frankl [27]. They proposed a notion that is appropriate for unit testing object oriented programs and developed an algebraic specification language called LOBAS and a tool called ASTOOT. The main feature of the approach is the notion of self-checking test cases for classes which use a class method that approximates observational equivalence. In other words, ASTOOT suggested a new way of generating oracles based on the equivalence of sequences of method calls. The test case in ASTOOT is a triple which consists of two sequences of method calls and a tag. The tag is a boolean value that denotes if the two sequences are equivalent, therefore they should produce the same results, or not. Sequences and the tag are obtained automatically from the algebraic specification. The work of Andrews and Zhang [5] is another example of specified oracles. The authors used state machines as oracles to test an elevator system. To build a test oracle based on a state machine a parser is applied to generate an analyser from the machine, and then the SUT's output is inserted into the analyzer. If the output is not expected in the machine then an error is detected. In this case, the state machine and the analyzer represent the oracle information

and the oracle procedure respectively. Overall, specified oracles are effective in finding failures but their success depends heavily on the availability of a formal specification which is limiting factor for most systems[9].

Implicit oracles are generated without reference to any domain knowledge or formal specification [9]. Fuzzing can be considered one effective way of detecting implicit anomalies [64]. Miller et al. [64] performed a systematic test of the utility programs running on various versions of the UNIX operating system. Their idea proceeded in four steps: (1) construct a program to generate random characters with a program to help test interactive utilities; (2) use these programs to test a large number of utilities on random input strings to see if they crash; (3) identify the strings or types of strings that crash these programs; (4) identify the cause of program crashes and categorise the common mistakes that cause these crashes. As a result of testing about 90 different utility programs on seven versions of UNIX, they were able to crash roughly 24% of these programs. This approach is usually used to find security vulnerabilities in the form of buffer overflows and memory leaks [98]. The work of Pacheco and Ernst [74] is another example of implicit oracles. The authors developed the Randoop tool to generate test suites. Randoop takes as input a set of classes under test, a time limit, and properties to check. The outputs of Randoop are test suites. Randoop classifies test cases into three types: ones that detect bugs in the current code of the SUT, ones that can be used as regression tests to detect future bugs, and ones that are invalid and are discarded. The classification depends primarily on whether the last statement throws an exception or violates a contract. Generally, implicit oracles are inexpensive and easy to obtain but are limited in their scope as they are not able to identify semantic and complex failures, revealing only general errors like system crashes or unhandled exceptions[9].

Derived oracles are created from properties of the system and artefacts other than the specification (e.g. textual documentation, execution information, regression test suites, metamorphic relations and pseudo-oracles). A pseudo-oracle is an example of a derived oracle proposed by Weyuker [102] and is a program (executable model or code) written in parallel to a system development by a second team following the same specification of the SUT. Both programs (the developed oracle and the SUT) are run with the same input data,

and then generated outputs are compared. If the outputs are equal or are within an acceptable margin of accuracy then the original program is considered to be fault free. However, there is no guarantee that the oracle is fault free. For instance, if obtained outputs for both programs are not equivalent a tester must go through a debug process to check which of the programs actually has the fault. Another example of derived oracles is the one generated based on known relationships between multiple inputs and outputs and known as metamorphic relations (MR). MR specify how the output of the program should change according to a specific change made to the input and represents some necessary properties [20]. Zhou et al.[112] used metamorphic testing to test search engines such as Google and Yahoo!. The experimental results showed that some commonly used search engines, including Google, Yahoo!, and Live Search, are not as reliable as most users would expect. For example, users may fail to find pages that exist in their own repositories, or rank pages in a way that is logically inconsistent. The authors have made some suggestions for search service providers to improve their service quality. Researchers claimed that derived oracles were able to reduce the cost of generating oracles but they were not effective compared to other types of oracles such as specified ones [78].

Each oracle category (specified, implicit and derived) could merit an entire survey in its own right. This thesis is focused on generating test oracles using anomaly detection techniques on dynamic execution data such as input/output pairs and execution traces. Therefore, the related work can be divided in two main sections: test oracles based on anomaly detection and test oracles based on invariant detection. Note that invariant detection approaches are included in the related work section because they have machine learning at their core.

2.4.1 Test Oracles Based on Anomaly Detection Techniques

Chandola et al. define anomaly detection as a matter of spotting patterns in data that correspond to abnormal behaviour [18]. This concept is illustrated in Figure 2.6 - N represents regions of normal behaviour, whereas O points represent the anomalous data. The aim of the work reported in this thesis to investigate whether software bugs generate a non-conformant

pattern of behaviour that can be distinguished from the conformant or normal behaviour - in other words, in Figure 2.6 do the groups marked N corresponded to passed tests and those marked O with failures? If this is the case then the possibility of detecting bugs automatically can be raised.

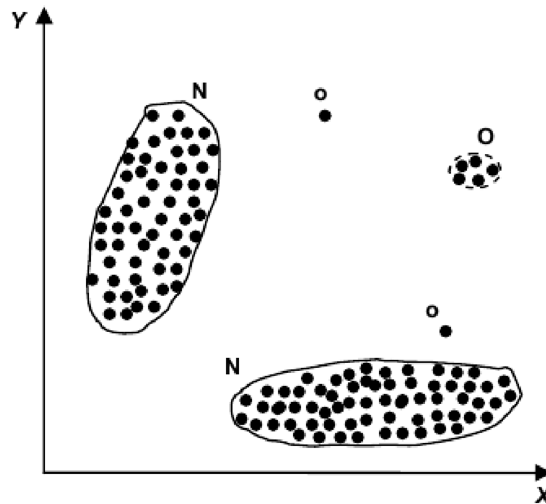

Fig. 2.6 Principle of Anomaly Detection from [18].

The main principle of creating test oracles in this context is to hypothesize a formal model of program behaviours from sets of observations. However, the application of anomaly detection strategies in this context has not been extensively investigated (for a detailed review of anomaly detection techniques and applications see the work of Chandola, Banerjee and Kumar [18]). The following subsections discuss some recent work in this area. The work in those subsections will be classified in to three main categories: (1) supervised learning techniques; (2) semi-supervised learning techniques; (3) unsupervised learning techniques.

Supervised Learning Techniques: This subsection presents related work that used supervised anomaly techniques for automated test oracles.

Vanmali et al. used Artificial Neural Networks (ANNs) to build an automated test oracle for the SUT [101]. A backpropagation algorithm was trained on the original version of the system by using randomly generated test cases that conformed to the specification. When new versions of the original system are created and regression testing was required, the tested code was executed on the test cases to yield outputs that are compared with those of the ANN.

They assumed that the new versions do not change the existing functions, which means that the new versions of the system were supposed to produce the same output for the same inputs. Then a comparison tool was used to make the decision whether the output of the tested system is incorrect or correct based on the ANN activation function. The experimental results showed that ANN model performed much better on binary outputs compared to continuous outputs, the minimum average error for binary outputs (the proportion of misclassified outputs) was 8.31% and for continuous outputs was 20.79%. The authors stated that the trained ANN can be used as an oracle to evaluate the correctness of the output produced by new versions of the system and can be used as a simulated model even though that model can not guarantee 100% correctness over the original system.

Aggarwal et al. [2] presented a case study of test oracles for triangle classification into isosceles, scalene, equilateral or invalid triangle by using ANNs. The ANN algorithm takes two inputs: a triple which represents three sides of a triangle and the category in which it fits. After the ANN is trained, it is capable of predicting in which category new triplets belong. The experimental results showed that ANN model was able to classify triangle with a reasonable degree of accuracy, with an average misclassification error between 15.9% to 19.02%, and also ANNs with 25 neurons in the hidden layer performed better. The work was extended by adding another subject program (a common metric known as function point) and performing more experiments [51]. The experimental results showed that the quality of prediction achieved was 92.18% for the triangle classification program and the 100% for the function point program. These results were in agreement with previous study. Their two studies were followed by Jin et al. [50] who used the triangle classification program with larger data sets (2000 test cases for each data set). The experimental results were not good as the previous studies, and the overall average performance for the ANN model was 60.33% but they showed that ANN model was able at least to build an automatic oracle to detect software faults. The main weakness of these studies is the subject programs which are very small with limited functionality. Therefore, it is not possible to generalise their results.

Mao et al. used the same methodology to build a test oracle for continuous functions [58, 108]. They considered the continuous function $\boldsymbol{y=F(x)}$ where $\boldsymbol{x}$ is the software input

vector, $\boldsymbol{y}$ is the corresponding output vector, and $\boldsymbol{F}$ is the software behaviour. The function $\boldsymbol{F}$ was modelled and expected output were generated using a trained ANNs (the backpropagation algorithm was used in the training phase). Their study was replicated by Lu et al. using RBF ANNs to test a small mathematical continuous function [56]. The experimental results for all studies showed that backpropagation and RBF ANNs were able to build an automatic oracle for continuous functions. The ANNs algorithms were also able to generate approximate outputs which were close to expected outputs for SUT. However, the subject programs in all studies were small which make it difficult to assess and generalise their results.

Shahamiri et al. built an automated test oracle by using ANNs to test the decision making structures for a student registration system [93]. The goal of the system was to maintain and manage the students' records and validate their registration using a decision making process based on the student's data that were given to the ANN as the input vector and consisted of six data items. The output was the result of applying the registration policies on the inputs to decide the validity of the students' registrations. The results of their study indicate that the resulting oracle succeeded in finding the injected faults with an accuracy 97.4%. Moreover, it found 98% of the mutants successfully. Shahamiri et al. replicated the same work by performing more experiments [91]. The accuracy of the proposed oracle was between 95.73% to 96.8%. However, both studies have the same problem that other studies in this area suffer from in that one subject program was used to evaluate their approach and only two types of faults were injected (operator and value changes).

All of previous studies used a single ANN oracle. Shahamiri et al. proposed a multi ANN oracle to perform input/output mapping in order to test more complicated software applications where a single ANN oracle may fail to deliver a high quality oracle [94]. A single ANN was defined for each of the output items of the output domain; then all of the networks together made the oracle. For example if the software under test produced 7 output items then seven ANNs are needed to create the multi ANN oracle. As a result, the complexity of the software may be distributed between several ANNs instead of having a single one to do all of the learning, and also separating the ANNs may reduce the complexity of the training process and increase the oracle's ability to find faults. Note that the training

process must be done for each of the ANNs separately using the same input vectors but only the output to be generated by the ANN. The experimental results indicate that multi ANNs oracle performed much better than single ANN oracle. The average accuracy of the multi ANN oracle was between 95.7% to 98.82%. On the other hand, the average accuracy of the single ANN oracle was between 84.65% to 95.9%. However, there was no significant difference between the average accuracy of single neural network oracle performance and multi ANNs oracle performance for the second subject program. In addition, building single ANNs for each output item to create a multi ANN oracles could be expensive.

Although all of previous studies show the ability of ANNs as a test oracle, they may not be reliable when the complexities of subject programs increase because they require larger training samples that could make the ANNs learning process complicated. A small ANN error could increase the oracle misclassification error considerably in large software applications. Moreover, most of these studies were evaluated by small subject programs having small input/output domains. Therefore, the ANN was able to perform the mapping in most of these studies. It is possible that a tiny difference exists between expected output generated by the ANN based oracle and the correct program. These issues could happen because of the complexity of the application under test. Consequently, generating the most representative data sets to train the ANNs could enhance the ANN performance and reduce the misclassification error. In addition, the structure of the network (e.g. the number of layers and neurons) is another issue which may not be easy to decide and solve.

Zheng et al. [111] built test oracles for search engines by using association rules. The main idea of their work is to mine implicit relations between queries and search results. They defined a set of items of queries and search results. After that, association rules were used to mine frequent rules between these items to construct a test oracle. The experimental results showed that their approach mines many high confidence rules that help to understand search engines and detect suspicious search results.

Frounchi et al. constructed automated test oracles to validate the correctness of image segmentation algorithms by using decision trees [33]. Two different algorithms (J48 and PART) were used to build the decision trees. In order to train the classifiers (oracles) to be

able to distinguish between consistent and inconsistent image segmentations, their technique used nine different attributes sets which consist of a number of measures related to volume difference, overlap and geometrical measures. Also, the correlation-based feature approach was used by the technique to select the attributes. The technique was able to achieve an average accuracy of approximately 90% during the evaluation phase. Yilmaz et al. used hybrid program spectra to train a decision tree classifier (J48 algorithm) that distinguishes passing and failing executions [109]. They used six types of program spectra. The first three program spectra are collected using hardware performance counters: (1) TOTINS counts the number of machine instructions executed; (2) BRNTKN counts the number of branches taken; (3) LSTINS counts the number of load and store memory instructions executed. The three remaining spectra are gathered using traditional software profiling: (1) CALLSWT records the functions invoked; (2) STMTFREQ counts the number of times the source code statements are executed; (3) TIME measures execution times of functions at the level of nanoseconds. The experimental results showed that hybrid spectra can be used to build a reliable classifier to distinguish failed executions from successful executions.

Parsa et al. [76] trained a support vector machine (SVM) to detect faults during the execution of subject programs. The proposed method was performed in two main phases: training and deployment. The training phase consisted of three steps: instrumentation, execution and learning. In the instrumentation step, probes are inserted before the branch statements or the locations within the program code where the value of predicates may change. In the execution step, the test cases are run against the instrumented original program code and the defective program code to generate failing and passing executions. At each run for each instrumented point within the program a separate vector is built. In the learning step, a SVM model is built by using predicate vectors obtained from the previous step. The detection accuracy for the model was between 80% - 90%. The work was extended by using a SVM with a customised kernel function to measure the similarities between passing and failing executions, represented as sequences of program predicates [75]. The fault detection accuracy of the proposed approach was 63% (83 bugs were revealed out of 132 bugs).

Lo et al. [54] proposed a new technique to classify unknown executions. Their technique first mined a set of discriminative features capturing repetitive series of events from program execution traces. After that, feature selection was performed in order to select the best features for classification. Then, these features were used to train a classifier (a support vector machine) to detect failures. Three kinds of frequently encountered bugs were injected (omission bugs, additional bugs and ordering bugs). Omission bugs are those where methods/functions that should have been invoked were absent. Additional bugs are those where methods/functions were invoked but were unnecessary and caused a failure of the execution run. Ordering bugs are those methods/functions that are called out of sequence. The experimental results showed the utility of the technique in capturing failures and anomalies; the classification accuracy rate ranged between 86.26% and 99.94%. However, the technique worked well with omission and additional bugs but poorly with ordering bugs, and the results showed that the proposed techniques outperformed the baseline approach by 24.68% in accuracy.

Other learning algorithms were used by Haran et al. [44] to classify execution data collected from applications in the field as coming from either passing or failing program runs. They used a statistical learning algorithm called random forests to model and predict the outcome of an execution based on the corresponding execution data (random forests is an ensemble learning method which builds a robust tree based classifier by integrating hundreds of different tree classifiers via a voting scheme). More specifically, the technique built a model by analysing execution data collected in a controlled environment by executing a large set of test cases in-house. It then used this information to lightly instrument numerous instances of the software (i.e., capture only the small subset of predictors referenced by the model). These lightly instrumented instances were then distributed to users who ran them in the field. As instances run, the execution data were fed to previously built model to predict whether the run is likely to be a passing or failing execution. Three types of execution data were used to build the model such as statement count (the number of times each basic block is executed for a given program run), throw count and catch count (throw counts measures the number of times each throw statement is executed in given run, and catch counts measures

the number of times each catch block is executed) and method count (the number of times each method has been executed for a given program run). The experimental results suggested that it is possible to classify a binary program outcome using various kinds of execution data. However, statement counts and method counts succeeded in building a classification model with higher accuracy compared to the classification model built using throw counts and catch counts.

The previous work was extended by proposing two different classification techniques which can build models with significantly less data than that required by first technique (random forests) but maintaining the same accuracy [43]. One of these techniques, called association trees, is based on the random forest algorithm. The technique has the following three stages: (1) the predictors are transformed into items that are present or absent; (2) association rules are found from these items; (3) a classification model is constructed based on these association rules. The technique was able to build a reliable model by using less than 10% of the complete execution data. Each instance collects a different subset of the execution data, chosen via uniform random sampling. The other technique is an adaptive sampling association tree and was also able to build reliable models based on a small fraction of the execution data. However, the adaptive sampling association tree has the ability to adapt the sampling over time so as to maximise the amount of information in the data. Overall, all three techniques performed well with overall misclassification rate typically below 2%.

All of proposed approaches under this category performed reasonably well but they require an accurate and reliable oracle to train the model (a large set of passing and failing executions) which might be difficult to obtain this oracle in real software testing scenario. In addition, most of proposed approaches were evaluated on a small systems which make it difficult to generalise their results and ability with larger and complex systems.

Semi-supervised Learning Techniques: Semi-supervised learning techniques have not yet been used to construct automated test oracles according to the existing literature [9, 77]. This subsection presents related work that uses semi-supervised learning techniques to support other software engineering tasks (e.g. reverse engineering and bug localisation) but not specifically aim for anomaly detection or automated oracle creation. However, these

works may potentially be useful for the purpose of constructing automated test oracles. Note that some of the related work in this subsection combines supervised learning with unsupervised learning, thus can be considered as semi-supervised learning. For instance, clustering (unsupervised learning) is often performed as a preliminary step in the data mining process; with the resulting clusters being used as further inputs into different techniques downstream such as a (supervised) neural network (it is often helpful to apply clustering analysis first to reduce the search space for the downstream algorithm).

Podgurski et al. built a system which would cluster bugs represented by a failed test that had the same cause [81]. Their system worked based on the analysis of the execution profile that corresponded to reported failures of the test. Their system also combined supervised and unsupervised learning strategies. The supervised learning strategy (logistic regression) was used to identify the program execution features in the profile such as the execution count for each function in the program. The feature selection was decided by generating candidate feature sets and used each one to create and train pattern classifiers to distinguish failures from successful executions, then selected the features of the classifier that performed best overall. The study reported that the execution count for each function in the program was the selected feature. The unsupervised learning strategy (cluster analysis) was used to analyse the execution profiles of reported failures. The experimental results showed that over 70% of automatically generated clusters appear to have the failures with the same cause clustered together.

Francis et al. proposed two new tree-based techniques for refining an initial classification of failures [31]. The first of these techniques was based on the use of dendrograms which are tree-like diagrams used to represent the results of hierarchical cluster analysis. Their dendrogram-based technique for refining failure classification was used to decide how non-homogeneous clusters should be considered for merging. The second technique for refining an initial failure classification relied on generating a classification tree to recognize failed executions. A classification tree was constructed algorithmically using a training set containing positive and negative instances of the class of interest. The experimental results indicated that both techniques were effective for grouping together failures with the same

or similar causes. All techniques were aimed at bug localisation by identifying groups of failures with closely related causes among a set of reported failures based on user feedback.

Bowring et al. proposed an automatic classification of program behaviours using execution data [16]. Their approach is aimed at reverse engineering a more abstract description of systems behaviour, rather than aiming to detect software defects. Their work focused on an active learning approach (rather than batch learning approach), where, for each iteration of learning, the classifier is trained incrementally on a series of labelled data elements and then applied to series of unlabelled data to predict those elements that most significantly extend the range of behaviours that can be classified. These selected elements are then labelled and added to the training set for the next round of learning. Their technique builds a classifier for software behaviour in two stages. Initially, a model of individual program executions was built as a Markov model by using the profiles of event transitions such as branches (a binary matrix was used to transform data to a suitable set of feature vectors). Each of these models thus represents one instance of the program's behaviour. The technique then used an automatic clustering algorithm to build clusters of these Markov models, which then together form a classifier tuned to predict specific behavioural characteristics of the considered program. The proposed technique was evaluated by conducting three empirical studies that explore a scenario illustrating automated test plan and augmentation. The scenario showed how their technique could reduce the costs and help to quantify the risks of software testing and development. The experimental results showed that the trained classifier on an active learning approach had a very high classification accuracy rate: up to 97.7%.

Mao et al. [57] used the Markov model approach proposed by Bowring et al. [16] with clustering analysis to aid fault localization. The methodology starts by using a Markov model and the profiles of event transitions such as branches to depict program behaviours. Based on the obtained model, the dissimilarity of two profile matrices is defined. After separating the failure executions and non-failure executions into different subsets, iterative partition clustering and a new sampling strategy called priority-ranked n-per-cluster are employed to extract representative failure executions. Note that the clustering and sampling strategy were

performed on the failure execution subset in order to choose the most representative sample of failures to reduce the debugging effort.

Baah et al. proposed a new machine learning technique that performed anomaly detection during software execution [7]. A Markov model was trained on trace predicate information and the Baum-Welch algorithm was used to find unknown parameters for the Markov model. Probes were inserted into the subject program to sample tuples in the form of <class name, method name, line number, predicate state>. These tuples were used to train a Markov model. Clustering of predicate states was used also in the training phase to gather predicate state information based on the line number and method number. In the line number clustering, all predicate states generated at an instrumented line number are grouped into one cluster. In method clustering, all predicate states belonging to a method are grouped into one cluster. The subject program along with the Markov model were then deployed together to detect faults as they occur and to possibly perform fault correction actions to prevent failures. The experimental results showed that the proposed technique performed well with domain faults with accuracy up to 100% in some cases. However, the technique did not perform well with computation error with accuracy less than 50% and dropping to 0% in some cases. The authors pointed out to a few efficiency issues such as the time required to build such model especially in the presence of a large test suit, the cost of incrementing the software to gather more information without incurring a significant overhead and how quickly the model can track the execution of the software.

Overall, semi-supervised learning techniques performed reasonably well in the area of bug localisation and reverse engineering. However, they deal with different problems to the one in this thesis. In traditional semi-supervised learning techniques, a small proportion of the test data is labelled as passing or failing and used along with unlabelled data to build a classifier (the same scenario used in this research). On the other hand, some of proposed approaches under this category combined supervised learning technique (Markov model) with unsupervised learning technique (clustering technique) which can be considered as semi-supervised learning approach. The most important factor is the generalizability of the

experimental results for all proposed approaches as small subject programs were used during the evaluation phase.

Unsupervised Learning Techniques: Similar to semi-supervised learning techniques, unsupervised learning techniques have not been intensively investigated to construct automated test oracles (according to the existing literature [9, 77]). This subsection presents related work that uses unsupervised learning techniques to support other software engineering tasks (e.g. software estimation reliability and test case selection) and they did not aim for anomaly detection or automated oracle creation.

Podgurski et al. proposed a new approach to reducing the manual labour required to estimate software reliability [79]. The approach combined the usage of automatic clustering analysis along with a stratified sampling strategy. The proposed approach was divided into two stages. In the first stage, automatic cluster analysis techniques (partitioning and hierarchical clustering methods with Euclidean distance and Manhattan distance metrics) were used to group execution profile data with similar features (the execution counts of conditional branches). In other words, the clustering was aimed at grouping similar executions together under the assumption that erroneous executions would be clustered separately from the correct ones. In the second stage, a stratified sampling strategy was used to reduce the sample size necessary to estimate reliability with a given degree of precision; the sampling here was used to identify which clusters represented failing executions, by selecting a data point from each cluster and determining if they were errors or not. The same study was replicated by Podgurski et al. by performing additional experiments with larger subject programs [80]. The aim of both studies was reduce the number of program executions that must be checked manually for conformance to requirements. The main assumption in both studies was that it is often possible to profile a population of executions so that a significant number of failures (failed executions) exhibit unusual profiles that can be revealed by multivariate data analysis, provided the software does not fail too frequently. The experimental results in both studies showed that the proposed approach in the first study was able to find clusters that exhibit unusual profiles with an average performance 58.681% for all subject programs, and 41.557% for all subject programs in the second study. In addition,

they also showed that stratified sampling was more efficient than simple sampling in most cases.

Dickinson et al. performed two different studies to investigate the use of clustering techniques for finding failures in software [24, 25]. In the first study, their technique involved profiling the executions induced by the original test cases and then applying an automatic agglomerative clustering algorithm. The sets of feature vectors were obtained by using different dissimilarity metrics (binary, proportional, standard deviation, histogram, linear regression, count binary and proportional binary metrics). Executions were clustered based on function caller/callee profiles. Execution samples after clustering were selected based on different sampling strategies (simple random, one-per-cluster, adaptive and n-per-cluster sampling strategies). The sampling strategies were used to choose the best samples which may contain failures from the whole cluster population. The experimental results showed that the percentage of failures found in the smallest clusters is significantly higher than 50%. The results suggested that one-per-cluster, adaptive and n-per-cluster sampling were more effective than simple random sampling in terms of finding failures. However, adaptive sampling was the most effective sampling strategy. In the second study, the previous work was extended by performing more experiments. The aim of the study was to confirm the experimental results for previous study, and also to investigate the distribution of failures in all clusters. The study proposed a new sampling strategy to find such areas called failure-pursuit sampling. The analysis of nearest-neighbour distance and Gaussian influence were the main idea behind of failure-pursuit sampling. In this sampling strategy, once a failure is detected in the initial selected sample of executions profiles additional executions are selected in the surrounding area of any failures found in the initial sample; this process is repeated until no failures are found. These additional executions were selected based on the density of surrounding neighbours. The experimental results for failure-pursuit sampling showed that failures were typically isolated from the other executions.

Masri et al. [60] presented an empirical study of several test case filtering techniques that are based on exercising various types of information flows. Coverage based and distribution based techniques were the main test case filtering approaches employed. A coverage based

approach selects test cases to maximize coverage of particular program elements, whereas a distribution based approach selects test cases to span the range of behaviors exhibited by the original test cases. In both approaches, test cases are characterized by execution profiles. Both approaches also were compared to other filtering techniques based on exercising simpler program elements such as basic blocks, branches, function calls and call pairs with respect to their effectiveness for revealing defects. Their empirical study showed that coverage maximization and distribution based filtering techniques were more effective overall than simple random sampling. In addition. distribution based filtering techniques did not perform significantly better than coverage maximization overall.

Yoo et al. also used a clustering approach to the problem of regression test optimisation [110] where test cases were clustered based on their dynamic runtime behavior (execution traces). Their work showed that the clustering approach outperformed the coverage based approach in terms of fault detection rate.

Yan et al. proposed a dynamic test cluster sampling strategy called execution spectra based sampling (ESBS) [104]. In their sampling strategy, a suspiciousness value is calculated for each test case. If the suspiciousness value is larger than the predefined threshold then the corresponding test is considered to be a possible failed test. Otherwise, it is considered to be a possible passed test. After that, the sampling strategy selects a possible failed test with maximum suspiciousness value from the cluster. Then, it uses the inspection result (the real pass or fail information) and the execution spectra information of the selected test to update the suspiciousness values for the remaining tests in the same cluster. The selection process is continued until no failed test in the cluster remain. The authors claim that their sampling strategy is more effective than existing test cluster sampling strategies [24, 25].

Generally, unsupervised learning techniques (mainly clustering algorithms with sampling strategies) were widely used in the area of software estimation reliability and test case selection. This has a strong influence on the work presented in this thesis on constructing automated test oracles using unsupervised learning techniques (mainly clustering algorithms), but an intensive experimental evaluation on this aspect is needed with deep analysis to test their general applicability as test oracles.

2.4.2 Test Oracles Based on Invariant Detection

Program behaviours can be automatically checked against the given invariants for violations. Therefore, invariants can be used as test oracles to distinguish between the correct and incorrect output. Invariants are often inserted into the code by the developers but this again can be a costly exercise and an additional burden at the time coding. The Daikon tool can be used to learn and infer invariants from program executions dynamically by using a collection of inputs (test cases), monitoring key values (class attribute, method entry and exit points, loop invariants etc.) and then making inferences from this large set of observations [29, 30]. Pacheco et al. [73] developed the Eclat tool which used Daikon to infer operational models from a set of correct executions and derive test oracles based on properties of the operational models. The input for Eclat tool is a set of classes to test and an example program execution (a passing test suite). The output for Eclat tool is a set of JUnit test cases, each containing a potentially fault-revealing input and a set of assertions at least one of which fails. The experimental results showed that Eclat successfully generated inputs which exposed fault revealing behavior.

Hangal and Lam used dynamic invariant detection to find program errors (the DIDUCE tool)[42]. Similar to Daikon, DIDUCE tries to extract invariants dynamically from program executions. However, instead of presenting the user with numerous invariants found after a program execution, DIDUCE continually checks the program behavior against the invariants hypothesized up to that point in the program run(s) and reports all detected violations. When a dynamic invariant violation is detected, the invariant is relaxed to allow for the new behavior and program execution is resumed. Their experimental evaluation showed that DIDUCE is effective in detecting hidden errors and finding the root causes of complex programming errors. Their experimental results also showed that DIDUCE is able to find bugs that result from algorithmic errors in handling corner cases, errors in inputs, and developers misconceptions of the APIs. The authors claimed that DIDUCE can be used to help programmers to locate bugs in unfamiliar code and, sometimes even in code that have not been instrumented.

Similarity to Pacheco et al. [74] and Hangal et al. [42], Brun and Ernst [17] proposed a new approach for finding program properties that indicate errors. The technique used a combination of machine learning algorithms (Support Vector Machines and Decision Trees) and dynamic invariant detection (mainly Daikon [30]). Daikon was used to obtain the program properties and machine learning algorithms to classify those properties. The technique comprised two stages: training and classification. In the training stage, machine learning algorithms were trained on properties of erroneous programs and fixed versions of them. As consequence, model fault revealing properties are generated which are true of incorrect code and not true of correct code. In the classification stage, the user supplies the generated model with properties of his or her code, then the model selects those properties that are most likely to be fault revealing. The main goal of the technique was to help programmers in locating errors in code by automatically providing the programmers with properties of code that are likely to be fault revealing. Their experimental evaluation showed that the proposed approach (Faults Invariant Classifier) effectively classifies properties as fault revealing. Ranking and selecting the top properties was more advantageous than selecting all properties considered fault revealing by the machine leaner. For C programs, on average 45% of the top 80 properties were fault revealing; for Java programs, 59% of the top 80 properties were fault revealing.

A second class of approaches under this category are the ones that focused on finite state model inference. The Synoptic tool developed by Beschastnikh et al.[13, 86] helps developers by inferring from log files a concise and accurate system model focusing on generating invariant-constrained models. The main component of Synoptic tool is the use of three types of mined temporal invariants to guide the model space exploration. Their evaluation showed that the Synoptic tool always makes progress and always finds a model that satisfies the mined invariants. Their case studies showed that Synoptic graphs improved developer confidence in the correctness of their systems, and were useful for finding bugs.

Lorenzoli et al.[55] combined the ideas of invariant detection and temporal property mining and developed a dynamic analysis algorithm (the algorithm called GK-tail) for extracting software behavioral models. GK-tail algorithm constructs an Extended Finite State

Machine (EFSM) from a set of dynamic traces. The transitions in these extended models include called functions or methods and a set of constraints on the parameters or environment. However, the main threat to validity is that the performance of proposed algorithm depends on the quality of the test suites used to produce EFSM models (all dynamic models have a similar problem).

Selar et al. [87] proposed an approach to learn Finite State Automata (FSA) by using sequences of systems call. Their approach deals with system security and aimed at detecting anomalous sequences of system calls which are likely to point to intrusion attempts and malware. Their experiments showed that the training periods needed for FSA based approach are shorter compared to the existing learning algorithm (N-gram algorithm) that have been used in the same area (intrusion detection). Moreover, the FSA based approach can detect a wide range of attacks and also produces much fewer false positives than the N-gram algorithm.

A third class of approaches under this category are the ones that exploited temporal dependencies between events to mine control flows under a high amount of interleaved traces [12]. Perracotta, developed by Yang et al.[106], infers temporal properties (in the form of pre-defined templates involving two API calls) from program executions. They applied Perracotta to infer temporal rules for the Windows kernel APIs where it successfully inferred 56 interesting properties. The authors also used the ESP verifier (a validation tool for typestate properties [97]) to check the inferred properties and found many previously undetected bugs in Windows.

Gabel and Su proposed an approach which used a sliding-window queue method over dynamic sequences of API method calls during the program execution to mine two-letter regular expressions [35]. Their approach was evaluated on a set of commonly used java programs and found that it learns and fully verifies a large set of temporal properties with an acceptable overhead. Their evaluation showed that the approach was able to reveal previously unknown defects and code smells. The approach maintained a high degree of precision.

Nguyen et al. [69] investigated the existing related work under this category and classified the main approaches that can be used as automated test oracles to three types: (1) *Data*

Invariants; (2) *Temporal Invariants*; (3) *Finite State Automation (FSA)*. They selected an oracle from each type based on their characteristics, popularity and supporting tools (Daikon [30] for data invariants, KLFA [59] for FSA and Synoptic [13] for temporal invariants). They studied the false positive rate and fault detection of those selected oracles. Their empirical results showed that automated test oracles can detect several real faults. However, the fault detection capability of those oracles comes at the price of a quite high false positive rate (30% on average). The high false rate makes the balance between practical benefits (revealed faults) and costs for manual assessment of the false alarms unclear.

Generally, most of test oracles under this category assumed that the fault free version of the SUT is available in order to train the oracle before using the oracle to detect faults (often difficult to obtain in practice). It must be stressed that the research presented in this thesis is not dealing with test oracles in this category (invariant detection) but they have been included as they may provide a useful basis for empirical comparison.

2.5 Discussion

In terms of their application in practice, the most important properties that test oracles need to demonstrate are scalability, fault detection ability, false positive rate and cost effectiveness. Each of those properties is explained further below:

- Scalability means the ability of any technique to handle any size of software (with corresponding increases in the volume of data). In other words, a technique has to be potentially usable at an industrial level.
- Fault detection ability refers to the effectiveness with which new (unseen) faults occurring in running application are identified.
- False positive rate is the rate of false alarms reported by test oracles. This can be considered as the biggest issue with automated oracles. When such a rate is intolerably high, any problem reported by automated oracles will be deemed unreliable and ignored by developers.

- Cost effectiveness takes into consideration the effort and resources required to create an oracle in relation to its ability to reveal subtle semantic failures.

Generally, all those properties are complementary to each other and can affect the usability of any test oracle in practice. The ultitmate goal of the software testing community is to find a test oracle that can be used to test any system, and is able to find all failures with a low false positive rate at an acceptable cost.

Test oracles based on supervised learning techniques have been widely used to build an automated test oracles. They have shown that they are able to test any system with any size which makes them scalable. They also tend to display a powerful ability to detect failures with a very low false positive and false negative rate (in other words a high classification accuracy). However, their effectiveness depends on the availability of a fully labelled training data set (each instance in the training data set has to be labelled as a pass or failing test execution) to construct the oracle. Labelling each instance in the training data set can be an expensive process and typically relies on using a reference version of the software (which is difficult to obtain in practice), making them prohibitively expensive and not cost-effective.

Test oracles based on invariant detection have also been discussed on the related work section because they have machine learning at their core. Those approaches can be used to test any system with any size but they require a fault free version of the software under test to construct the oracle which is difficult to obtain in real world testing scenario. As with supervised learning, this can affect the cost-effectiveness of these approaches. Test oracles based on invariant detection are effective at finding failures but tend to suffer from quite a high false positive rate.

2.6 Conclusions

This chapter discussed several approaches to build automated test oracles based on anomaly detection techniques (machine learning and data mining techniques), and also based on invariant detection. Some other approaches were also presented only briefly because they are

out of the scope for this research. The review has identified out to the following principal findings:

- Anomaly detection approaches are potentially applicable for automated test oracles but different machine learning and data mining techniques with different set of features (dynamic execution data) give mixed results in terms of detection accuracy. Classification strategies (supervised) are reported to be more effective compared to clustering (unsupervised) but have the disadvantage of relying on the availability of accurate labels and a large training set for various normal and anomaly classes, which is often not feasible.

- Different types of dynamic execution data have been used as a set of features to build anomaly detection models including the execution count of conditional branches, function caller/callee profile, execution count for functions or methods, profiles of event transitions, predicate state information, inputs/outputs, throw counts, catch counts and execution traces. The experimental results in some studies showed that the execution count of conditional branches and of functions or methods is more suitable to build an effective model compared to throw count and catch counts.

- Different approaches have been used to transform dynamic execution data to a suitable set of feature vectors for anomaly detection such as binary metric, proportional metric, standard deviation (SD) metric, histogram metric, linear regression metric, count binary metric and proportional binary metric. The binary metric is the most commonly used approach.

- Test oracles based on invariants detection can easily generate many false alarms.

The review showed that unsupervised and semi-supervised learning techniques have not been investigated intensively for building test oracles for automated software fault detection but do show elements of promise in this respect, thereby justifying the choice of research area. The review suggests that more empirical work is required to explore their general application. Therefore, the next two chapters will present an empirical investigation for both

semi-supervised and unsupervised learning techniques to build automated test oracles by using input/output pairs and execution traces. Both approaches will also be compared to test oracles based on invariant detection techniques.

Chapter 3

Automatically Classifying Test Results by Semi-Supervised Learning

3.1 Introduction

The literature review in the area of test oracles motivates the need for further experimentation on semi-supervised machine learning techniques used to build test oracles. Self-training, co-training and co-EM (Expectation Maximisation) have been chosen for further investigation as those are the techniques that are sparking popular interest amongst academic researchers in the area of document classification and web classification. Furthermore, self-training, co-training and co-EM (expectation maximisation) were also reported to be better for prediction in the area of document classification and web classification.

The main purpose of the work is to investigate the use of semi-supervised learning techniques to classify passing and failing tests. A small proportion of the test data is labelled with a large proportion of unlabelled test data and the learning algorithms use this (both labelled and unlabelled data) to build a classifier which is then used to label the remaining data (i.e. classifying it as being either a passing or failing test). A range of learning algorithms are investigated using several versions of three systems along with varying types of data (initially just input/output pairs and then input/output pairs with their corresponding execution traces) and different labelling strategies (both failing and passing tests, and just passing tests

alone). Moreover, the work explores the optimal number of labelled data (i.e. a subset of the input/output pairs labelled as correct) to employ in relation to the program domain to generate an effective oracle. In addition, any other practical issues that arise when applying semi-supervised learning to the test oracle area are observed.

3.2 Methodology

3.2.1 Semi-Supervised Learning

The principle of semi-supervised learning is that the learning algorithm is fed a labelled subset of the data - instances for which the correct classification is known - and from this builds a model which is then used to classify the remaining (unlabelled) data. There is a clear trade-off between the accuracy of the classifier and the volume of data used in training, and the challenge is to build the most effective classifier from the smallest amount of data. For techniques that operate in semi-supervised approach there are two possible scenarios [18]. The first scenario is that the training data has a small set of labelled instances from both the abnormal/anomaly class and the normal class [22] [21], and the second is that the training data has labelled instances for only the normal class.

At first sight it may appear unusual to try and employ the first scenario to identify software failures. After all, if testers are able to label a failure then why bother looking for it? However, there are several scenarios where this approach could be employed. For example, failures may be available from a previous version of the software, or there may be many faults in the system and an initial subset which has been observed and classified and may then be used to build a model to detect the remainder, or the failures may be artificially created by seeding the SUT with faults in much the same way as mutation testing.

For the second scenario the training data has labelled instances for only the normal class - this would correspond to a subset of passing test cases in this work. Since such techniques do not require labels for the abnormal class they are therefore more widely applicable. This approach has been successfully used to detect faults in space craft where an abnormal class would signify an accident which is not easy to model [34]. Therefore, the model was built

for the class corresponding to normal behaviour only and used to identify anomalies in the test data.

There are two types of semi-supervised learning categorised according to the prediction goal: inductive learning where the goal is to predict unseen data which were not available during the training; and transductive learning where the goal is to predict the unlabelled data which were available during the training. This research is focused in transductive learning – trying to predict whether unlabelled data items correspond to a passing or failing output.

3.2.2 Semi-Supervised Learning Algorithms

Three approaches to semi-supervised learning are explored in this research: *self-training*, *co-training* and *co-EM (expectation maximisation).*

Self-Training

The basic principle behind self-training is firstly to train a classifier on the small set of available labelled data and then use this to classify the large remaining set of unlabelled ones. This process may be performed iteratively as outlined in Algorithm 1 [68] - at each stage the classifier that has been bootstrapped with the labelled data (training set) labels the remaining data and adds those in which it has the most confidence to the training set. This updated training set is then used to build a new classifier which is then applied to the remaining unlabelled data, and so on... The self-training algorithm can be thought of as a wrapper algorithm, as it itself takes an algorithm as a parameter which it uses to build the classifier at each stage. A popular and robust approach (which has shown to perform well in the domain of document classification for example [71]) is to combine the Naïve Bayes and EM (expectation-maximization) algorithms: the initial classifier is built using Naïve Bayes and the extension to the unlabelled data at each stage of the iteration is handled by EM.

Algorithm 1 Pseudo code for the self-training algorithm

Input: Labelled data D_L, unlabelled data D_U, and a supervised learning algorithm A.

Step 1: Train a classifier c using labelled data D_L with A.
Step 2: Label unlabelled data D_U with c.
Step 3: For each class C, select an example which c labels as C with high confidence, and add it to the labelled data D_L.
Step 4: Repeat 1–3 until it converges or no more unlabelled data D_U left.

Output: c

Co-Training and co-EM

Approaches based on co-training assume that the features describing the object can be divided in two independent subsets: perspectives that individually are sufficient to train a good classifier. Two classifiers - one for each perspective - are created using the initial set of labelled data and then iteratively trained as described by Algorithm 2 [68]. At every iteration each classifier contributes the newly labelled data with the associated highest confidence to the labelled set which is then used as training data for the next iteration. In this way the two classifiers teach each other with a respective subset of unlabelled data and their highest confidence predictions [15].

Co-EM also operates with two perspectives but takes a different approach at each stage of the iteration. The first classifier is trained on the labelled data and then used to probabilistically label all the unlabelled data (not just the those elements in which it has the highest confidence). The second classifier is trained on both labelled data and the unlabelled data which has been tentatively labelled by the first classifier, and it in turn probabilistically relabels all the data for the first classifier to use. The process iterates until the classifiers converge [70].

In this research two implementations of the co-training algorithm were explored: one using Naïve Bayes and the other a Support Vector Machine with a Radial Basis Function (RBF) kernel. One instance of the co-EM algorithm was used which also employed the Support Vector Machine with a RBF kernel. Those implementations represent the most

successful technology for text/web categorization which is the main reason for using them in this research.

Algorithm 2 Pseudo code for the co-training algorithm

Input: Labelled data D_L, unlabelled data D_U, and a supervised learning algorithm A.

Step 1: Train a classifier c_F using the feature set F of each example with A.
Step 2: For each class C, pick the unlabelled data D_U which classifier c_F labels as class C with highest confidence, and add it to labelled data D_L.
Step 3: Train classifier c_E using the feature set E of each example with A.
Step 4: For each class C, pick the unlabelled data D_U which classifier c_E labels as class C with highest confidence, and add it to the collection of labelled data D_L.
Step 5: Repeat 1–4 until it converges or no more unlabelled data D_U left.

Output: Two classifiers, c_F and c_E

Co-training has been shown to perform well if the two assumptions about the splitting of the feature set are true [15]: each feature should be sufficient by itself to build a good classifier and the two features are conditionally independent of each other. These two assumptions often may not be satisfied in real world application but it has been demonstrated empirically [40, 70] that co-training can still be effective in such a case if the feature set is split randomly (although not as effective as if independent perspectives are employed). In this research the data in the second phase of experiments is a combination of input-output pairings and method execution traces which could be regarded as distinct perspectives. However, there are questions about their independence (the path taken by a program is a function of its input). Therefore, a random split for the data into two subsets for both co-training and co-EM was chosen. This allows for a more direct comparison between experiments.

As mentioned earlier, both self-training and co-training make use of the confidence of prediction as selection criterion during the labelling process to decide which of the unlabelled samples should move to the labelled set. However, as well at the prediction confidence this step should also make use of the class distribution - the split between positive and negative examples (passing and failing outputs in this work), and try and maintain this proportion during the selection process. For instance, if the positive to negative class ratio in the labelled data set is 3:1, after the unlabelled data are classified by the classifier, then 3 “positive”

examples and 1 "negative" example with the highest predicted posterior probabilities will be selected in each iteration.

3.3 Experiment Design

A series of experiments was conducted to assess the effectiveness of the algorithms and scenarios presented in the previous section in terms of accuracy of classification of test data (unlabelled data). Two different studies were performed with two different types of execution data: in the first study a set of input/output pairs was used as input to classifiers; in the second study input/output pairs were augmented with their associated execution traces. Along with this the two common scenarios employed by semi-supervised learning algorithms was explored: labelling both normal and abnormal data, and labelling normal data alone. For both these scenarios different proportions of labelled data were investigated. This section describes the framework used to design these experiments.

3.3.1 Subject Programs

Versions of three subject programs were used in this research: the NanoXML XML parser, Siena (Scalable Internet Event Notification Architecture) and Sed (stream editor). All systems are available from the Software Infrastructure Repository (SIR)[1], are non-trivial systems, have several versions with well-documented faults, and also come with test suites – an important factor as having sets of good with representative coverage of operation profile, but comprehensive and also independently created, tests is vital for this experiment.

NanoXML

NanoXML is a non-GUI based XML parser written in Java. NanoXML contains a component library and an application, JXML2SQL, which takes as input a XML file and either transforms it into a html file and showing the contents in tabular form or into a SQL file. NanoXML has 24 classes, 5 versions (although the fourth version was excluded as it contains no faults),

[1] http://sir.unl.edu/portal/index.php

each containing multiple faults – 7 in each of versions 1-3 and 8 in version 5 – and 214 test cases. The error rates in all faulty versions ranged from 31% to 39% (the error rate is the proportion of the supplied test cases which will fail due to the seeded faults).

Siena

Siena (Scalable Internet Event Notification Architecture) is an Internet-scale event notification middleware for distributed event-based applications deployed over wide-area networks. Siena is responsible for selecting notifications that are of interest to clients (as expressed in client subscriptions) and then delivering those notification to the clients via access points. Siena contains 26 classes (9 in its core and 17 which constitute an application), 567 test cases and 7 faulty versions: 3 with single, and 4 with multiple ones. Versions with multiple faults (V1,V3,V5 and V7) have been excluded from this experiment for the time being because of the absence of a fault matrix (a simple way of establishing which test cases are responsible for revealing which fault). Therefore, only V2, V4 and V6 are included in the experiment, each having a single fault and an error rate of 17%.

Sed

Sed (stream editor) is a Unix utility that parses and transforms text by using a simple compact programming language. Sed takes text input in the form of commands and a text file, performs some operation (or set of operations) on the text file, and outputs the modified text. Sed is typically used for extracting part of a file using pattern matching or substituting multiple occurrences of a string within a file. Sed is written in C and has 225 functions, 370 test cases and 7 versions with multiple faults. However, only one version was included in the experiment (version 5 which has 4 faults and an error rate of 18%).

3.3.2 Experiment Set-up

The main components of the experiments were: a set of programs with known failures, a set of test inputs for each program, a way to determine whether an execution of each test was

successful or not (passed or failed), and a mechanism for recording the execution trace taken through the program by each test. Each of these steps is described in more detail below.

Input/Output Pair Collection

All subject programs come with Test Specification Language (TSL) test suits and tools to run these automatically (details are available from the SIR repository and the article by Do, Elbaum and Rothermel [26]). Test cases which failed to produce any output were discarded. Failure to produce an output occurred in a small number of cases where the input file was missing from the test suite, and consequently no output file was produced: 7 out of 214 for NanoXML, 73 out of 567 for Siena, and 7 out of 370 for Sed, giving final test case numbers of 207, 494 and 363.

Execution Trace Collection

Daikon[2][30] was used to instrument the subject programs in order to collect the execution traces. For all subject programs, each test case was executed along with its input to produce one trace with one output. Daikon allows programs to be monitored and traced at varying levels of granularity, but for this work sequences of method invocations (entry points) and method exits in the order they occurred during test execution were extracted.

Identification of Failures

Both NanoXML and Sed systems come with matrices which map test cases to failures corresponding to faults and makes the identification of faults effectively automatic. Siena has no such fault matrix so the test outputs of the original version was compared with that of the faulty ones to find the failing tests.

Data Transformation

To be acceptable to the various machine learning algorithms, the data requires processing before it can be analysed. The processing procedures differ from one data type to another – for

[2] http://plse.cs.washington.edu/daikon/

instance numeric data sometimes requires normalisation. All systems used in the experiment work with textual input and produce textual output. Very often there is little semantic information in such data and a lot of noise, so to minimise the content (and redundancy) but still retain any uniqueness, the data (input/output pairs) was transformed by a simple process of tokenisation. The tokenisation method is widely used in the area of text mining to produce a suitable set of attribute vectors to build a classification model (a problem not dissimilar to the one this thesis is dealing with), and is also suggested by Witten and Frank [103]. Several transformation methods such as hash coding, Huffman coding and others were examined, but tokenisation turned out to be the most suitable one, and also performed well with clustering algorithms for a similar problem [82]. Table 3.1 shows an example of this for both NanoXML and Siena. Notice that the parameters for Siena commands were all encoded as "1" as they remained unchanged between input and output.

The Sed data (input/output pairs) is a command line which contains 2 main parts with very specific information: parameters (operations to perform) and a text file (input / modified text file as output). Therefore, the data was transformed in slightly different way compared to NanoXML and Siena. For instance, all input components remained unchanged except the text file (../inputs/default.in) encoded as "<1>" as the file has only the text to be modified. For the output part, the diff utility (a data comparison tool) was used to calculate and display the differences between the original text file and the modified form (this process reports how to change the first file to make it match the second file with specific operations that need to be performed such as "a" for add and "c" for change). Table 3.1 shows an example of this coding stratgy.

Each input/output pair was augmented with their associated execution traces in the second study. Sequence traces for the Java systems are often very long, and each entry in a sequence is often a full Java method signature including package name, class name, method name, and parameters (along with their respective long signatures). Sequence traces for the C system are similar to the one generated by Java system where each entry in a sequence is often a full C function/procedure signature including function/procedure name and parameters. This required more compression than could be provided by simple tokenisation so the trace

Table 3.1 Example Coding of Input/Output Pairs

	Input	Output
Nanoxml	**Flower colour="Red" smell="Sweet" name="Rose" season="Spring"**	xml element name is: **Flower**
Encoding	**FCRSSNRSS**	**F**
Siena	Filter senp{x=0}**filter**{**x=20 y=30 z=10**} Event senp{x=0}**event**{**x=20**} senp{x=0}**event**{**y=30 z=10**}	**subscribing** for **filter**{**x=20 y=30 z=10**}**publishing** for **event**{**x=20**}**publishing** for **event** {**y=30 z=10**}
Encoding	**F111E1E11**	**SF111PE1PE11**
Sed	**sed -e 's/dog/cat/' ../inputs/default.in**	the modified text file (change and add operations)
Encoding	**sed-es/dog/cat/<1>**	**114a36c34c29c26c3l4c0a**

compression algorithm developed by Nguyen et al. [69] was used. The algorithm replaces the collections of method sequence entry and exit values with their hash keys, consisting usually of just 1 or 2 characters. It takes into account the occurrence frequency to assign shorter hash keys for entries that are most frequent. Table 3.2 shows a sample of sequences for one of collected traces and their hash key values (for space reasons, just 3 sequences are included rather than all sequences of that trace). The obtained trace from the example in this table is **0LA37**.

Finally all the data items were used in two different studies. In the first study, the input to the classifier was input/output pairs only as a set of vectors. This vector is built from two components: input and output, so for the NanoXML example in Table 3.1 the vector would be :<**FCRSSNRSS, F**>. In the second study, the vector is built from three components: input, output and execution trace, so for the NanoXML example from Table 3.1 above and the generated trace shown in Table 3.2, the vector would be: <**FCRSSNRSS, F, 0LA37**>. Appendix B and C provide more detail on the encoding scheme.

Table 3.2 Example Coding of Sequence Traces

Sequence Traces	Hash Keys Values
net.n3.nanoxml.XMLElement. getFullName():::EXIT283	0L
net.n3.nanoxml.XMLUtil.skipWhitespace (net.n3.nanoxml.IXMLReader,char, java.lang.StringBuffer, boolean[]):::ENTER	A
net.n3.nanoxml.StdXMLReader. getEncoding(java.lang.String):::ENTER	37

Selection of Labelled and Unlabelled Data

Cross validation was used during the selection of labelled *(D_L)* and unlabelled *(D_U)* data to avoid bias in the choice of data. Different values were set for the size of *(D_L)* in the experiments ranging from 10% to 50% of data (based on a percentage of the number of subject program test cases). The process was repeated so that every input/output pair will appear once in *(D_L)* during training process. Two semi-supervised learning scenarios are explored in this research: the first is where labels are drawn from both the normal and abnormal classes (i.e. passing and failing tests) and is termed *(Scenario 1)*, the second is where labels are drawn from just the normal (passing) class *(Scenario 2)*. To try and avoid biasing the results and also to maintain a more realistic scenario, the set of labelled failing cases for scenario 1 was kept deliberately small. Tables 3.3, 3.4 and 3.5 show the labelled data size and class distribution that were used during the experiments in both studies for scenario 1 for all versions of all subject programs.

In practice, it is often difficult to obtain a training data set which covers every possible abnormal behavior class that can occur in the data and in this case a small subset of 5 failing executions was randomly chosen (and these in turn may appear quite distinct, as one fault may transform the output in many different ways depending on the input). The failures to which these correspond for the different versions of NanoXML and Sed are shown in Tables 3.6 and 3.7, and this same abnormal labelled set was used as the size of the normal labelled set grew. For Siena there is just one failure (which again has different manifestations) so all 7

abnormal labels related to this. As mentioned earlier, there may be other ways of obtaining abnormal data such as from previous versions of the software or via seeded faults.

Table 3.3 Labelled training data set sizes and class distribution for NanoXML (all versions) for scenario 1

Labelled size %	Normal data	Abnormal data	Unlabelled data
10% (25 labelled instances)	20	5	182
20% (45 labelled instances)	40	5	162
30% (65 labelled instances)	60	5	142
40% (85 labelled instances)	80	5	122
50% (103 labelled instances)	98	5	104

Table 3.4 Labelled training data set sizes and class distribution for Siena (all versions) for scenario 1

Labelled size %	Normal data	Abnormal data	Unlabelled data
10% (50 Labelled instances)	43	7	444
20% (100 Labelled instances)	93	7	394
30% (153 Labelled instances)	146	7	341
40% (200 Labelled instances)	193	7	294
50% (247 Labelled instances)	240	7	247

Table 3.5 Labelled training data set sizes and class distribution for Sed (version 5) for scenario 1

Labelled size %	Normal data	Abnormal data	Unlabelled data
10% (39 Labelled instances)	34	5	331
20% (69 Labelled instances)	64	5	301
30% (104 Labelled instances)	99	5	266
40% (141 Labelled instances)	136	5	229
50% (180 Labelled instances)	175	5	190

Also of particular interest in this research is the way the data separates into distinct groups based on input, output and trace combinations, which may have an impact on the effectiveness of the machine learning algorithms employed. Table 3.8 shows for each system the number of test cases followed by the number of distinct inputs, outputs, input-output combinations, traces, and input-output-trace information. This table will be used in the discussion of the results.

Table 3.6 Abnormally Labelled data items for NanoXML studies using scenario 1

Version No.	Labelled failures (number of instances)
Version 1	F1 (5)
Version 2	F6 (3) and F7 (2)
version 3	F6 (3) and F7 (2)
Version 5	F1 (2) and F2 (3)

Table 3.7 Abnormally Labelled data items for Sed studies using scenario 1

Version No.	Labelled failures (number of instances)
Version 5	F3 (5)

Table 3.8 Domain size for the three systems

System	NanoXML (V1)	NanoXML (V2, V3 and V5)	Siena	Sed
Total No. Tests	207	207	494	363
Distinct Inputs	57	57	37	206
Distinct Outputs	70	70	60	179
Distinct I/O Combinations	120	120	104	287
Distinct Traces	105	2	2	295
Distinct I/O/Trace Combinations	120	120	104	341

3.3.3 Evaluation

The performance of semi-supervised learning algorithms was evaluated by using the *F-measure* – a combination measure of *Precision* and *Recall* (widely used measures in information science domain). These measures in turn rely on the concepts of true positives (TP), false positives (FP) and false negatives (FN) which are defined in this context as follows:

TP: A failing test result classified as failing test

FP: A passing test result classified as failing test

FN: A failing test result classified as passing test

Precision is defined as the ratio of correctly classified failures to the total number of true positive (correctly classified failures) and false positive (incorrectly classified passing tests):

$$Precision(PR) = \frac{(TP)}{(TP+FP)} \tag{3.1}$$

Recall is the ratio of correctly classified failures to the total number of true positive (correctly classified failures) and false negative (incorrectly classified failing tests):

$$Recall(RE) = \frac{(TP)}{(TP+FN)} \tag{3.2}$$

The F-measure - the harmonic mean of precision and recall - combines these two as follows:

$$F - measure = 2\frac{(PR \times RE)}{(PR+RE)} \tag{3.3}$$

In all cases these values are calculated purely on the *unlabelled* data instances - (D_U).

Generally, the F-measure was chosen because it is a well understood and commonly used measure which also combines important elements of a classifier – precision and recall – into one measure. Precision is a reflection of the false positive (FP) rate (a passing test result classified by the oracle as failing test) and recall reflects the false negative (FN) rate (a failing test result classified by oracle as passing test). The false positive and false negative rates are

the most important aspect to assess the test oracle performance and those aspects are captured by F-measure to provide a balanced view of oracle performance. The Error Rate is also a useful measure but captures only one aspect the classifier – it's accuracy (see the equation below) and furthermore does not distinguish between the source of the errors - whether they are attributable to false positive or false negatives. That said, the Error Rate was used later on to examine the results and the significance level of the classifiers.

$$ErrorRate(ER) = \frac{(FP+FN)}{(TP+TN+FP+FN)} \tag{3.4}$$

3.3.4 Comparison with Daikon

To try and provide some meaningful comparison regarding the effectiveness of the approach presented in this research, a comparison with Daikon [30] is also performed. Daikon is a popular tool in the specification mining area that can also be used as a form of automated test oracle to identify failing outputs. Daikon is dynamic analyser that is able to infer likely program invariants from the synthesis of program properties (e.g. key variables and relationships) observed over several program traces. An invariant is a property that holds at a certain point or points in a program; these are often used in assert statements, documentation and formal specifications. Daikon instruments and runs a program, observes the values that the program computes, and then builds assertions from inference over the collective values of the properties observed in the traced executions. These assertions can then act as a form of specification and subsequent executions of the program can be checked against these to determine if the assertions still hold or are violated at some point. It must be stressed that Daikon assumes that the system under test has a fault-free version with a large and complete test suite on which to train the assertions – something which is difficult to obtain in reality. In contrast semi-supervised learning has a slightly different assumption (the training data has labelled instances for a subset of passing and failing test cases or passing test cases only - see both scenario in section 3.2.1), and also it does not require a large and complete test suite on which to train the model (only a small proportion of the test data is labelled as either

passed and failed executions or only passed executions with a large proportion of the test data remaining unlabelled).

To build the initial set of assertions Daikon was run on the *non-faulty* version of each system using the same supplied set of test cases. Following this, Daikon was run on the various versions of the program containing the seeded faults (again using the supplied test cases) and used to establish whether there were any violations of the initially established assertions. To confirm that Daikon detects a seeded fault (true positive), the output reports produced by Daikon were manually inspected to find if there is any direct link between the reports (the violated invariants) and the seeded faults (information about the seeded faults can be found for each subject program in the SIR). A true positive is noted if the Daikon assertions reported an alarm that is verified to point to the corresponding faults. A false positive is recorded if the Daikon assertions reported an alarm but which does not relate to any of the corresponding faults. If the assertions do not report any alarm when one was expected (i.e. in response to a failure-inducing input) then this can be considered as False Negative. These values are then used to compute the F-measure which provides a useful point of comparison.

3.3.5 Tools and Configuration

A collective classification package (release 2015.2.27)[3] for semi-supervised learning in WEKA[4] (release 3-6-12) was used in the experiments. The maximum number of iterations in self-training and co-training is set to 80, and to 30 for co-EM in the experiments (as used in [40]). Default values for all other parameters (except the iteration parameter) were used as given in their implementation. The Daikon configuration employed was the most recent version and the same as that used in other experiments [30, 69].

[3]https://github.com/fracpete/collective-classification-weka-package/releases
[4]http://www.cs.waikato.ac.nz/ml/weka/downloading.html

3.4 Experimental Results and Discussion

This section presents the results obtained from using semi-supervised learning approaches (self-training, co-training and co-EM) on the three subject programs described in Section 3.3, using two kinds of execution data: firstly just input/output pairs, and secondly input/output pairs augmented with execution traces. In addition the two semi-supervised labelling scenarios were explored (section 3.2.1).

3.4.1 Study 1: Test Result Classification Based on Input/Output Pairs

Scenario 1: Labelling subset of both passing (normal) and failing (abnormal) tests: For the first study the classifiers were built using just the test case inputs and outputs from which a subset of instances of the normal behavior class (in this case, passing execution) were labelled along with a few instances for abnormal behaviour (failing executions). The rest of the data were unlabelled instances of test case input/output pairs which the classifiers iteratively categorised during the learning process.

Tables 3.9 and 3.10 show, for NanoXML and for Siena and Sed respectively, the results of applying semi-supervised learning with increasing numbers of Labelled samples. The figures reported are the average values obtained from employing cross validation to reduce the potential bias created by the choice of the Labelled data items (see section 3.3.2). The first column (size) defines the number of Labelled data items in the training sets as a percentage of the total number of test cases. The subsequent columns refer to the version number of the subject programs with (P, R, F) referring to the average values of Precision, Recall and F-measure (see section3.3.3) with the best results in each column highlighted in bold. A full experimental results over different cross validation can be found in Appendix D.

For NanoXML the self-training method (Naïve Bayes with EM) performed well over all versions, achieving an average F-measure of 0.5 when only 10% of the data (just 25 items) was Labelled. However, to achieve the better results it would be necessary to label between 30-50% of the data. Co-training using Naïve Bayes performed far less well, and more notably did not really improve as the size of the Labelled training set increased. Even

more disappointing are the results for co-EM and co-training with SVM: the performance for version 1 of NanoXML is identical for both algorithms but let down by poor recall values, but for versions 2, 3 and 5 the recall was zero most of the time as no failures were detected (indicated by a '-' n the results tables). Some learning algorithms are severely affected by the imbalanced training data set problem such as SVM. This could explain the poor performance for co-EM and co-training with SVM.

The performance of Naïve Bayes with EM on NanoXML is encouraging, particularly considering the number of input and output combinations that need to be classified (see Table 3.8). Given that there are 120 distinct input-output pairs it is also to be expected that the performance improves when the 30-50% bracket is reached – at this point between 60-100 data items will have been labelled and chances are that this will have covered the majority of the distinct combinations. It might be expected that as more cases get labelled so the accuracy would increase but this is not always the case, especially for some of the other algorithms. Looking at these cases it appears that initially many results were classified as fail (some correctly, some not). As more data gets labelled several of these fail results were turned into passes - some correctly but some not - due to the influence of one label (they may match a labelled passing input for example). This impacts on precision and recall as the FP value will drop as will the TP value which means that precision increases but recall decreases. By adding in more data in the form of traces it is anticipated that this undue influence from one component will diminish.

A similar pattern of results can be seen from the data for Siena – the semi-supervised learning techniques did not perform well in all versions. Remember that the versions of Siena contain just the one fault, so all failures correspond to the same fault (but which will have different manifestations). As with NanoXML the best F-measure values are achieved with self-training (Naïve Bayes with EM) but unlike NanoXML there was no increase in performance as the proportion of labelled data increased. This feature is something of a surprise as in comparison to NanoXML the number of input-output combinations for Siena is relatively small (104 - see Table 3.8). With nearly 500 test cases for the system the expectation would be for the majority of these combinations would be included by the time

that 30% of the data was labelled but this seemed to have no impact. This may be down to either the choice of abnormal cases or the fact that the increase in the number of tests means the data becomes very imbalanced. The results for the other approaches are generally poor, and when not zero are too low to be considered usable.

The results for Sed in some ways also reflect those for the other two systems. The best results are achieved using Naïve Bayes with EM but at the low level of labelling these are quite weak. It is only when around 30% of the data is labelled that these begin to become acceptable. This is perhaps to be expected as Sed had the most fragmented input-output profile of all the systems (287 in total - see Table 3.8).

Figures 3.1 and 3.2 present the relationship between the size of labelled data on the training set and the best classifier performance (the average F-measure for self-training and co-training methods with Naïve Bayes). The figures are essentially a graphical summary of the data that appears in Tables 3.9 and 3.10.

Scenario 2: Labelling subsets of only passing (normal) tests: In the second scenario classifiers were trained on a small set of labelled instances for normal behaviour class (passing executions) alone, with the remaining data being unlabelled instances (an unknown class for the classifier during the training process). The same set of learning strategies were explored but none of approaches performed well enough to warrant reporting in more detail. The trained classifiers were only able to classify all passing (normal) execution data correctly but miss-classified all failing (abnormal) execution data, labelling it as normal data instead.

3.4.2 Study 2: Test Result Classification Based on Input/Output Pairs Augmented with Execution Traces

Scenario 1: Labelling subset of both passing (normal) and failing (abnormal) tests: For this second study the input data for the semi-supervised learning strategies consisted of the input/output pairs used in the first study augmented with their execution traces. In both cases the data are encoded as described in section 3.3.2 to reduce them to a manageable size (the trace data in particular). This change aside, all other aspects - semi-supervised learning

Table 3.9 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs and Training on Normal and Abnormal Cases

Self-training (EM-Naïve):

	NanoXML Version			
Labelled Data Size	V1 (P, R, F)	V2 (P, R, F)	V3 (P, R, F)	V5 (P, R, F)
10%	(0.47, 0.40, 0.43)	(0.57, 0.46, 0.51)	(0.65, 0.60, 0.62)	(0.42, 0.46, 0.44)
20%	(0.57, 0.40, 0.47)	(0.74, 0.53, 0.62)	(0.72, 0.45, 0.56)	(0.52, 0.40, 0.45)
30%	(0.74, 0.46, 0.56)	**(0.83, 0.63, 0.72)**	(0.86, 0.71, 0.78)	(0.52, 0.40, 0.45)
40%	**(0.80, 0.80, 0.80)**	(0.83, 0.63, 0.72)	**(0.85, 0.78, 0.82)**	(0.68, 0.66, 0.67)
50%	(0.77, 0.80, 0.78)	(0.83, 0.63, 0.72)	(0.79, 0.78, 0.79)	**(0.73, 0.76, 0.75)**

Co-training (Co-Naïve):

	NanoXML Version			
Labelled Data Size	V1 (P, R, F)	V2 (P, R, F)	V3 (P, R, F)	V5 (P, R, F)
10%	(0.52, 0.29, 0.37)	**(0.65, 0.18, 0.28)**	**(0.63, 0.17, 0.27)**	**(1, 0.21, 0.35)**
20%	**(0.48, 0.49, 0.48)**	(1, 0.08, 0.15)	(1, 0.08, 0.15)	(1, 0.21, 0.35)
30%	(0.90, 0.23, 0.37)	(1, 0.08, 0.15)	(1, 0.08, 0.15)	(1, 0.21, 0.35)
40%	(1, 0.16, 0.27)	(1, 0.08, 0.15)	(1, 0.08, 0.15)	(1, 0.06, 0.11)
50%	(1, 0.16, 0.27)	(1, 0.08, 0.15)	(1, 0.08, 0.15)	(1, 0.06, 0.11)

Co-EM (EM-SVM):

	NanoXML Version			
Labelled Data Size	V1 (P, R, F)	V2 (P, R, F)	V3 (P, R, F)	V5 (P, R, F)
10%	(0.55, 0.23, 0.33)	(-, -, -)	(-, -, -)	(-, -, -)
20%	(0.55, 0.23, 0.33)	(-, -, -)	(-, -, -)	(-, -, -)
30%	**(0.82, 0.23, 0.36)**	(-, -, -)	(-, -, -)	(-, -, -)
40%	(1, 0.16, 0.27)	(-, -, -)	(-, -, -)	(-, -, -)
50%	(1, 0.16, 0.27)	(-, -, -)	(-, -, -)	(-, -, -)

Co-training (Co-SVM):

	NanoXML Version			
Labelled Data Size	V1 (P, R, F)	V2 (P, R, F)	V3 (P, R, F)	V5 (P, R, F)
10%	(0.55, 0.23, 0.33)	(-, -, -)	(-, -, -)	(-, -, -)
20%	(0.55, 0.23, 0.33)	(-, -, -)	(-, -, -)	(-, -, -)
30%	**(0.82, 0.23, 0.36)**	(-, -, -)	(-, -, -)	(-, -, -)
40%	(1, 0.16, 0.27)	(-, -, -)	(-, -, -)	(-, -, -)
50%	(1, 0.16, 0.27)	(-, -, -)	(-, -, -)	(-, -, -)

Table 3.10 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs and Training on Normal and Abnormal Cases

Self-training (EM-Naïve):

Siena Version		Sed Version	
Labelled Data Size	V2 (P, R, F)	Labelled Data Size	V5 (P, R, F)
10%	**(0.19, 0.60, 0.28)**	10%	(1, 0.10, 0.19)
20%	(0.10, 0.23, 0.13)	20%	(1, 0.10, 0.19)
30%	(0.09, 0.20, 0.12)	30%	**(0.39, 0.86, 0.54)**
40%	(0.09, 0.20, 0.12)	40%	(0.39, 0.86, 0.54)
50%	(0.09, 0.20, 0.12)	50%	(0.39, 0.86, 0.54)

Co-training (Co-Naïve):

Siena Version		Sed Version	
Labelled Data Size	V2 (P, R, F)	Labelled Data Size	V5 (P, R, F)
10%	(0.09, 0.21, 0.13)	10%	(1, 0.04, 0.08)
20%	**(0.37, 0.10, 0.16)**	20%	(1, 0.04, 0.08)
30%	(0.16, 0.03, 0.05)	30%	(1, 0.04, 0.08)
40%	(0.10, 0.23, 0.14)	40%	**(0.85, 0.36, 0.51)**
50%	(0.09, 0.21, 0.13)	50%	(0.85, 0.36, 0.51)

Co-training (Co-SVM):

Siena Version		Sed Version	
Labelled Data Size	V2 (P, R, F)	Labelled Data Size	V5 (P, R, F)
10%	**(0.11, 0.15, 0.13)**	10%	(-, -, -)
20%	(-, -, -)	20%	(-, -, -)
30%	(-, -, -)	30%	(-, -, -)
40%	(-, -, -)	40%	(-, -, -)
50%	(-, -, -)	50%	(-, -, -)

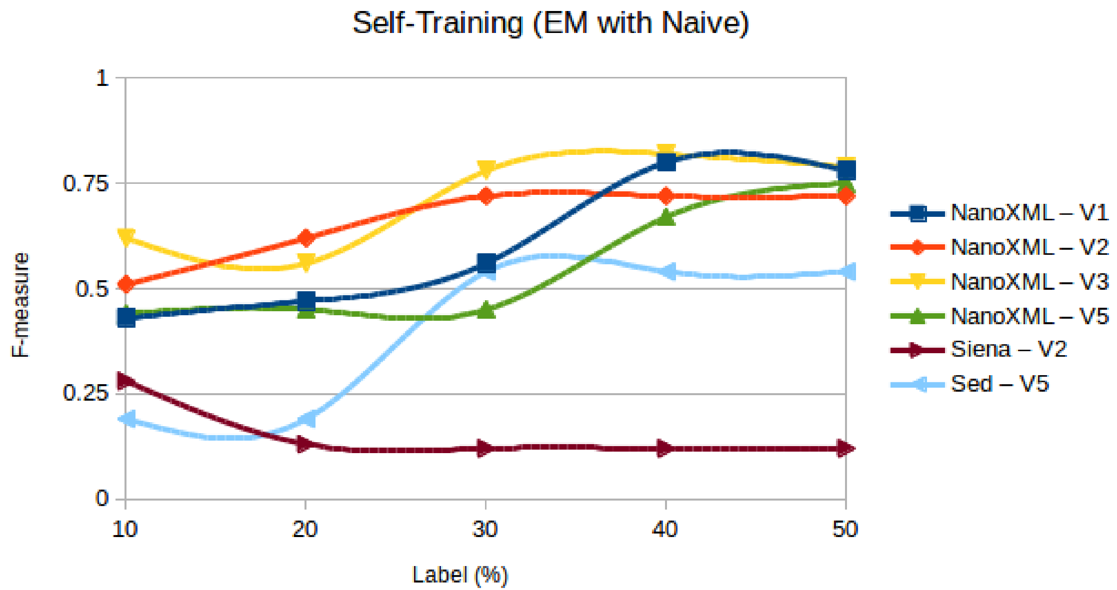

Fig. 3.1 The Average F-measure (F) Values vs. Labelled Data Size for Self-Training (EM-Naïve) Using Input/Output Pairs and Training on Normal and Abnormal Cases

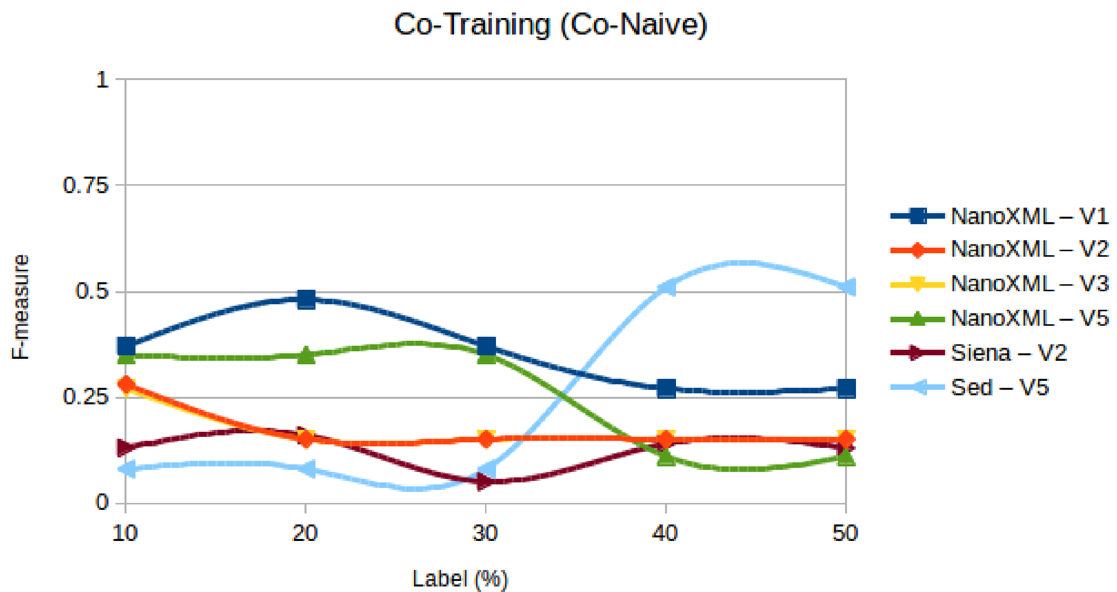

Fig. 3.2 The Average F-measure (F) Values vs. Labelled Data Size for Co-Training (Co-Naïve) Using Input/Output Pairs and Training on Normal and Abnormal Cases

algorithms, subject programs, proportions of Labelled data explored - are identical to the first study.

The results of this study for all versions of the subject programs are shown in Tables 3.11 and 3.12 which have the same format as Tables 3.9 and 3.10. The data shows that for versions 2, 3 and 5 of NanoXML the self-training method (Naïve Bayes with EM) has performed particularly well, producing an F-measure of 1 (in other words, correctly classifying all passing and failing executions) based on labelling only 10% of the data items. The reason for this is both encouraging and also slightly disappointing. The trace data perfectly separates the results (there are only two distinct traces - see Table 8): all passing tests follow one route through the program and all failing ones follow the other route. This demonstrates the useful information that execution trace information can bring but is also not a scenario that is likely to be observed that frequently. Performance on version 1 is not strong but becomes acceptable when around 30% of the data has been Labelled. This may be a consequence of the faults that lie within version 1 or possibly due to having just the one abnormal case Labelled (see Table 3.6), or the fact that its profile is very different to the other versions in having 105 distinct traces. What is notable is that once around 30% of the data is labelled then the performance is better than the results achieved from classifying the input-output data alone without the execution trace information.

Co-training using Naïve Bayes displays a very similar pattern although the improvement observable in version 1 for self-training is absent. Co-training with SVM did not perform well on any versions with the exception of version 3 (it is unclear why similar results should be have been achieved for versions 2 and 5). The results for co-EM were particularly poor and have been omitted from the table. The poor performance for co-EM and co-training with SVM could be attributable to the imbalanced training data sets problem which SVM algorithm always suffers from.

The results for Siena for all techniques (with the exception again of co-EM) are universally good, accurately classifying the passing and failing executions with 100% accuracy based on just the smallest labelling proportion. Again, this is for exactly the same reasons as versions 2,3 and 5 of NanoXML – the passing and failing executions are perfectly separable based

on the execution traces alone. This trace pattern was entirely unknown at the time the two systems were selected and not something that was either expected or planned for.

Sed displays a very similar pattern of results to those achieved using input-output pairs alone. Indeed the inclusion of execution traces appear to have practically no impact on the results. It has already been observed that Sed has the most fragmented input-output combination, and together with its 295 distinct traces gives 341 distinct input-output-trace combinations. For cases such as this it is clear that other information needs to be introduced to the classifier algorithms such as execution time or summary information relating to traces (e.g. number of unique methods, nesting pattern etc.) in order to achieve better results.

Figures 3.3 and 3.4 illustrate the relationship between the size of labelled data on the training set and the best classifier performance (the average F-measure for self-training and co-training methods with Naïve Bayes). The figures are essentially a graphical summary of the data that appears in Tables 3.11 and 3.12.

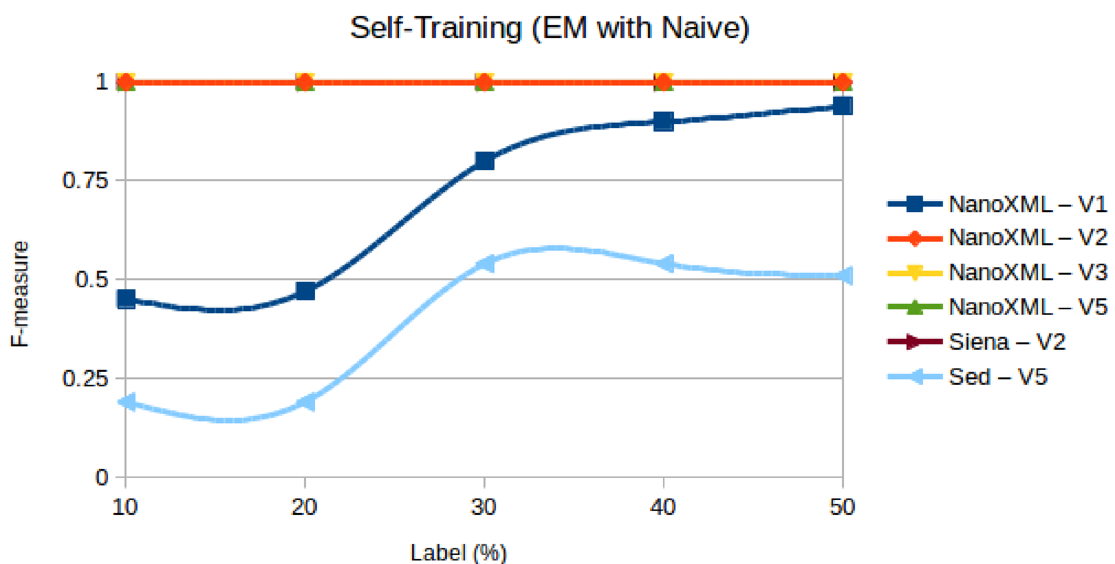

Fig. 3.3 The Average F-measure (F) Values vs. Labelled Data Size for Self-Training (EM-Naïve) Using Input/Output Pairs Augmented with Execution Traces and Training on Normal and Abnormal Cases

Scenario 2: Labelling subset of only passing (normal) tests: For this part of the study the data used was as for the first scenario - input/output pairs augmented with their execution

Table 3.11 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs Augmented with Execution Traces and Training on Normal and Abnormal Cases

Self-training (EM-Naïve):

	NanoXML Version			
Labelled Data Size	V1 (P, R, F)	V2 (P, R, F)	V3 (P, R, F)	V5 (P, R, F)
10%	(0.50, 0.41, 0.45)	**(1, 1, 1)**	**(1, 1, 1)**	**(1, 1, 1)**
20%	(0.47, 0.47, 0.47)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
30%	(0.70, 0.94, 0.80)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
40%	(0.86, 0.94, 0.90)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
50%	**(0.94, 0.94, 0.94)**	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)

Co-training (Co-Naïve):

	NanoXML Version			
Labelled Data Size	V1 (P, R, F)	V2 (P, R, F)	V3 (P, R, F)	V5 (P, R, F)
10%	**(0.50, 0.40, 0.44)**	**(1, 1, 1)**	**(1, 1, 1)**	**(1, 1, 1)**
20%	(0.50, 0.40, 0.44)	(1, 1, 1)	(1, 0.98, 0.99)	(1, 1, 1)
30%	(0.83, 0.28, 0.42)	(1, 1, 1)	(1, 0.98, 0.99)	(1, 1, 1)
40%	(0.90, 0.28, 0.43)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
50%	(0.90, 0.28, 0.43)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)

Co-training (Co-SVM):

	NanoXML Version			
Labelled Data Size	V1 (P, R, F)	V2 (P, R, F)	V3 (P, R, F)	V5 (P, R, F)
10%	(-, -, -)	(-, -, -)	**(1, 1, 1)**	(-, -, -)
20%	(-, -, -)	(-, -, -)	(1, 1, 1)	(-, -, -)
30%	(-, -, -)	(-, -, -)	(1, 1, 1)	(-, -, -)
40%	(-, -, -)	(-, -, -)	(1, 1, 1)	(-, -, -)
50%	(-, -, -)	(-, -, -)	(1, 1, 1)	(-, -, -)

Table 3.12 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs Augmented with Execution Traces and Training on Normal and Abnormal Cases

Self-training (EM-Naïve):

Siena Version		Sed Version	
Labelled Data Size	V2 (P, R, F)	Labelled Data Size	V5 (P, R, F)
10%	**(1, 1, 1)**	10%	(1, 0.10, 0.19)
20%	(1, 1, 1)	20%	(1, 0.10, 0.19)
30%	(1, 1, 1)	30%	**(0.39, 0.86, 0.54)**
40%	(1, 1, 1)	40%	(0.39, 0.86, 0.54)
50%	(1, 1, 1)	50%	(0.39, 0.86, 0.54)

Co-training (Co-Naïve):

Siena Version		Sed Version	
Labelled Data Size	V2 (P, R, F)	Labelled Data Size	V5 (P, R, F)
10%	**(1, 1, 1)**	10%	(1, 0.07, 0.14)
20%	(1, 1, 1)	20%	(1, 0.07, 0.14)
30%	(1, 1, 1)	30%	(1, 0.07, 0.14)
40%	(1, 1, 1)	40%	**(0.39, 0.84, 0.53)**
50%	(1, 1, 1)	50%	(0.39, 0.84, 0.53)

Co-training (Co-SVM):

Siena Version		Sed Version	
Labelled Data Size	V2 (P, R, F)	Labelled Data Size	V5 (P, R, F)
10%	**(1, 1, 1)**	10%	(-, -, -)
20%	(1, 1, 1)	20%	(-, -, -)
30%	(1, 1, 1)	30%	(-, -, -)
40%	(1, 1, 1)	40%	(-, -, -)
50%	(1, 1, 1)	50%	(-, -, -)

Co-EM (EM-SVM):

Siena Version		Sed Version	
Labelled Data Size	V2 (P, R, F)	Labelled Data Size	V5 (P, R, F)
10%	(-, -, -)	10%	(1, 0.07, 0.14)
20%	(-, -, -)	20%	(1, 0.07, 0.14)
30%	(-, -, -)	30%	(1, 0.07, 0.14)
40%	(-, -, -)	40%	(1, 0.07, 0.14)
50%	(-, -, -)	50%	(1, 0.07, 0.14)

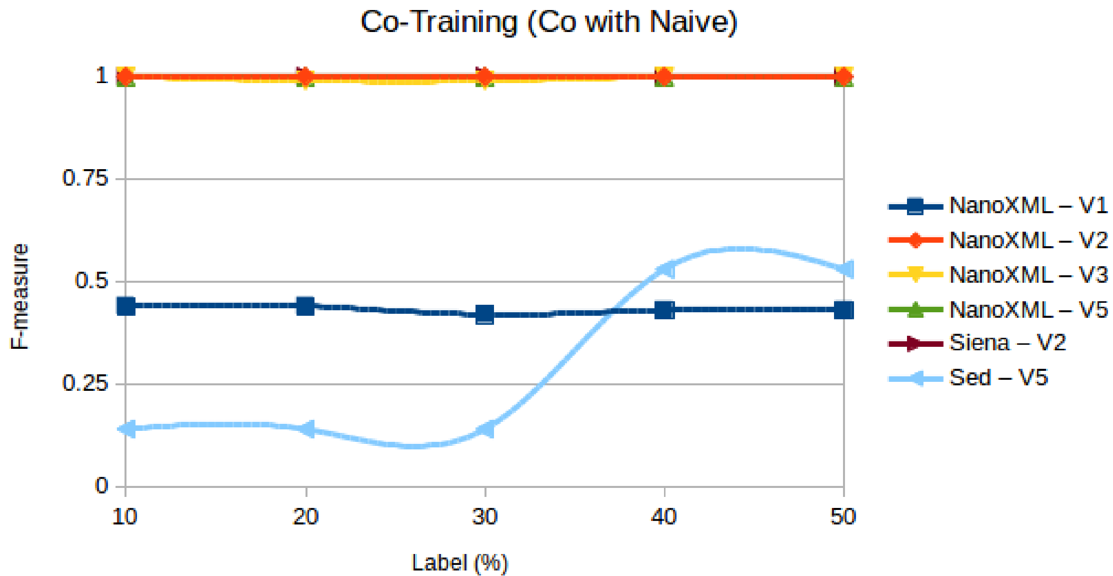

Fig. 3.4 The Average F-measure (F) Values vs. Labelled Data Size for Co-Training (Co-Naïve) Using Input/Output Pairs Augmented with Execution Traces and Training on Normal and Abnormal Cases

traces - but only a normal (passing execution) subset of the data was Labelled. The results are shown in Tables 3.13 and 3.14 and display a similar, but slightly less effective, pattern to scenario 1.

Again self-training (Naïve Bayes with EM) has performed well and has correctly classified almost all the data for versions 2, 3 and 5 of NanoXML when only a small proportion of the data is Labelled. For version 1, very much like in scenario 1, the results are not as impressive and even though they improve as the proportion of labelled data increases, the overall performance as indicated by the F-measure is held back by the low recall. The reasons for this can be explained by the trace information in much the same way as for the previous scenario (there are only two distinct traces for version 2, 3 and 5 whereas 105 distinct traces for version 1 - see Table 3.8). Co-training using Naïve Bayes also shows a similar trend, performing well on versions 2 and 3 and also on version 5 except at the lowest level of labelling. For version 1 though co-training failed to distinguish between the passing and failing executions, producing recall values of 0 most of the time. Co-training with SVM and

co-EM both produced results that were close to zero the majority of the time and have been omitted from the results.

The results for Siena (Table 3.14) show that both self-training (Naïve Bayes with EM) and co-training (Naïve Bayes) methods have extremely performed well, again due to the very informative execution traces. Self-training was able to detect all failures with all Labelled sample sizes on all versions and co-training produced a similar set of results with the exception of the smallest (10%) labelling proportion. Co-training with SVM and co-EM again did not perform well for Siena either and the results for these have been omitted.

Again Sed proved to be the most challenging system and for Naïve Bayes with EM did not display any acceptable results until at least 30% of the data was labelled (again attributable to the fragmentation). Co-training (Naïve Bayes) displayed a similar performance but was able to produce results even with the smallest level of labelling. Even though the results for this approach are not as accurate as the previous scenario, the absence of any labelled failing inputs makes this an interesting outcome.

Figures 3.5 and 3.6 show the relationship between the size of labelled data on the training set and the best classifier performance (the average F-measure for self-training and co-training methods with Naïve Bayes). The figures are essentially a graphical summary of the data that appears in Tables 3.13 and 3.14.

Table 3.13 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs Augmented with Execution Traces and Training on Normal Cases Alone

Self-training (EM-Naïve):

	NanoXML Version			
Labelled Data Size	V1 (P, R, F)	V2 (P, R, F)	V3 (P, R, F)	V5 (P, R, F)
10%	(0.45, 0.12, 0.20)	(1, 0.96, 0.98)	(1, 0.89, 0.94)	(1, 0.92, 0.96)
20%	(0.37, 0.12, 0.19)	(1, 0.96, 0.98)	**(1, 0.97, 0.98)**	(1, 0.92, 0.96)
30%	(0.40, 0.15, 0.22)	(1, 0.96, 0.98)	(1, 0.97, 0.98)	(1, 0.92, 0.96)
40%	(0.66, 0.20, 0.30)	**(1, 1, 1)**	(1, 0.97, 0.98)	(1, 0.96, 0.98)
50%	**(0.89, 0.24, 0.38)**	(1, 1, 1)	(1, 0.97, 0.98)	**(1, 1, 1)**

Co-training (Co-Naïve):

	NanoXML Version			
Labelled Data Size	V1 (P, R, F)	V2 (P, R, F)	V3 (P, R, F)	V5 (P, R, F)
10%	(-, -, -)	**(1, 0.96, 0.98)**	**(1, 1, 1)**	(0, 0, 0)
20%	(-, -, -)	(1, 0.80, 0.88)	(1, 0.81, 0.89)	(1, 0.63, 0.74)
30%	(-, -, -)	(1, 0.80, 0.88)	(1, 0.81, 0.89)	(1, 0.63, 0.74)
40%	(-, -, -)	(1, 0.80, 0.88)	(1, 0.81, 0.89)	**(1, 0.64, 0.78)**
50%	(-, -, -)	(1, 0.80, 0.88)	(1, 0.81, 0.89)	(1, 0.64, 0.78)

Table 3.14 The Average Precision (P), Recall (R) and F-measure (F) values vs. Labelled Data Size for Semi-Supervised Learning Techniques Using Input/Output Pairs Augmented with Execution Traces and Training on Normal Cases Alone

Self-training (EM-Naïve):

Siena Version		Sed Version	
Labelled Data Size	V2 (P, R, F)	Labelled Data Size	V5 (P, R, F)
10%	**(1, 1, 1)**	10%	(-, -, -)
20%	(1, 1, 1)	20%	(-, -, -)
30%	(1, 1, 1)	30%	(0.39, 0.45, 0.42)
40%	(1, 1, 1)	40%	(0.33, 0.56, 0.42)
50%	(1, 1, 1)	50%	**(0.41, 0.68, 0.51)**

Co-training (Co-Naïve):

Siena Version		Sed Version	
Labelled Data Size	V2 (P, R, F)	Labelled Data Size	V5 (P, R, F)
10%	(1, 0.81, 0.89)	10%	(0.38, 0.28, 0.33)
20%	**(1, 1, 1)**	20%	(0.36, 0.25, 0.30)
30%	(1, 1, 1)	30%	(0.43, 0.39, 0.41)
40%	(1, 1, 1)	40%	(0.40, 0.45, 0.42)
50%	(1, 1, 1)	50%	**(0.42, 0.48, 0.45)**

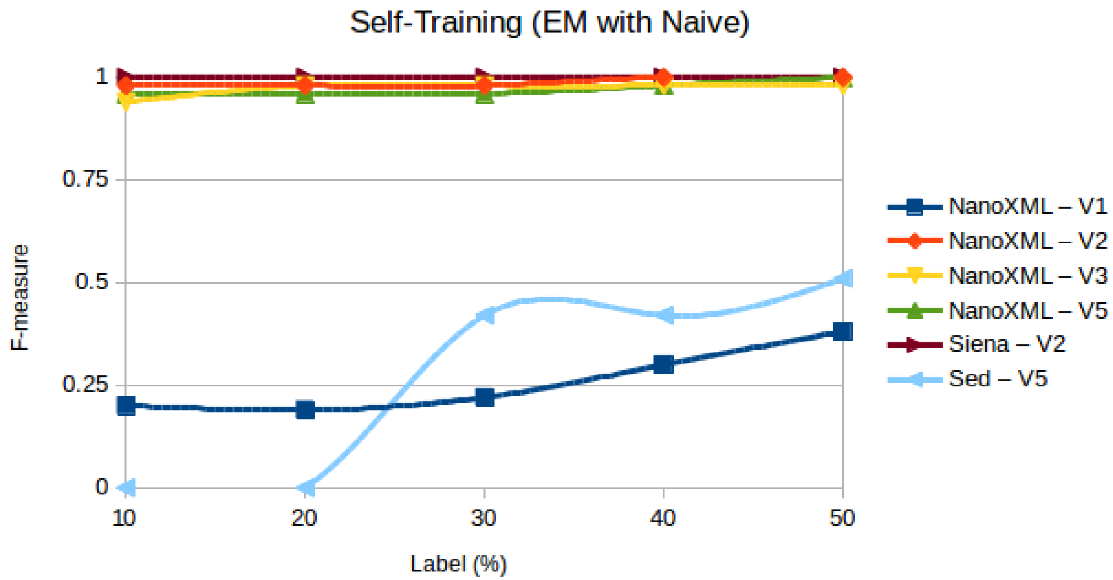

Fig. 3.5 The Average F-measure (F) Values vs. Labelled Data Size for Self-Training (EM-Naïve) Using Input/Output Pairs Augmented with Execution Traces and Training on Normal Cases Alone

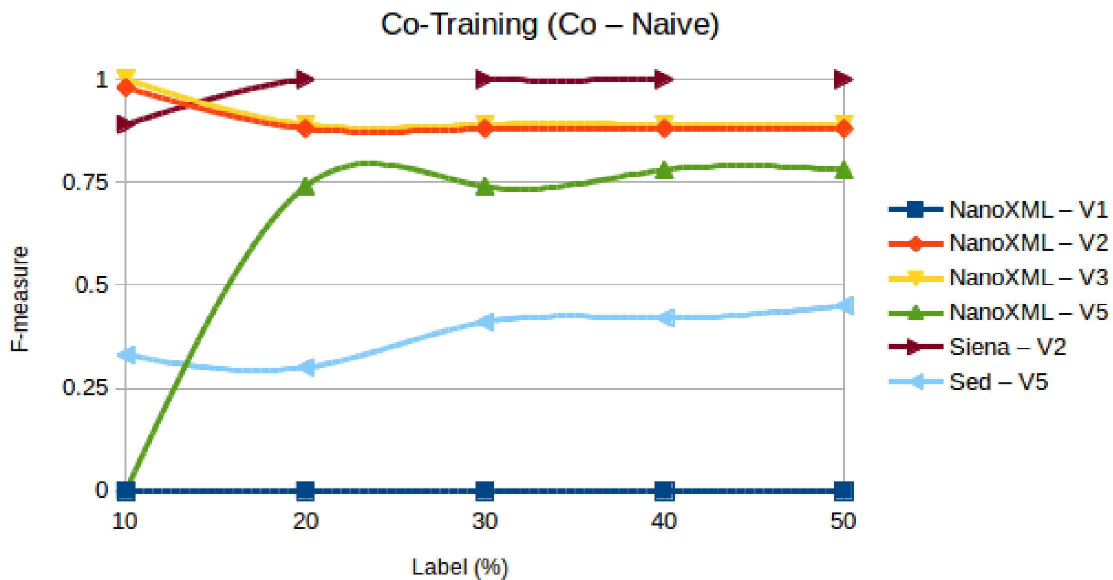

Fig. 3.6 The Average F-measure (F) Values vs. Labelled Data Size for Co-Training (Co-Naïve) Using Input/Output Pairs Augmented with Execution Traces and Training on Normal Cases Alone

3.5 Statistical Test for semi-supervised learning Hypothesis

The experimental hypothesis for test oracle based on semi-supervised learning techniques can be formulated in the following way *"Test oracles based on semi-supervised learning techniques are able to accurately classify a significant majority of unlabelled (or unseen) data"*. The Binomial test will be used to test the experimental hypothesis on the experimental data results for all subject programs used in this thesis (see Table 3.15). A null hypothesis, alternative hypothesis, equation parameters and significance level are stated as follows:

- The Null hypothesis (H_0) $P <= 0.5$. Where P is the classification probability compared against random chance classification probability obtained by (0.5). In other words, it is compared against a random oracle where a random oracle is a classifier which classifies data by random chance.
- The Alternative hypothesis (H_1) $P > 0.5$.
- (N) is the number of errors for the classifier (FP+FN) under a particular labelled data size, (K) is the number of tests which represent the number of unlabelled data that the classifier attempts to classify.
- The significance level is 0.05, and Z-Test method with a right one tailed test is used.

The null hypothesis is rejected in all cases. It was observed that the results are statistically significant in all cases where $p-value$ lower than the significance level (0.05) - (see Table 3.16). This can be generalised to the rest of experimental data.

3.6 Semi-supervised Learning Techniques versus Daikon

The performance of Daikon is compared with the best results from each of the semi-supervised approaches under both scenarios for all systems with both data types. Also included in the comparison are the results from using two base classifiers (Naïve Bayes and

Table 3.15 Random Tested F-measure from Several Experiments

Test	System	F-m	Number of Errors (FP+FN) (N)	Number of tests (K)	Labelled Data Size
Test 1	NV1	0.80	32	122	40%
Test 2	NV2	0.72	35	142	30%
Test 3	NV3	0.82	24	122	40%
Test 4	NV5	0.75	33	104	50%
Test 5	SV2	0.28	242	444	10%
Test 6	SeV5	0.54	46	266	30%

Table 3.16 The Results of Z-Test and p-value on Tested F-measure

Test Number	z	$p-value$	Note
Test 1	37.299883	3.0762×10^{-303}	(H_0) rejected - significant results
Test 2	41.919651	0	(H_0) rejected - significant results
Test 3	44.703188	0	(H_0) rejected - significant results
Test 4	30.289512	2.3888×10^{-200}	(H_0) rejected - significant results
Test 5	41.46217	0	(H_0) rejected - significant results
Test 6	71.509349	0	(H_0) rejected - significant results

Support Vector Machines) built in fully supervised mode where *all* of the data for the training set is labelled and the classifier is built and evaluated using 10-fold cross-validation (where the classifier is trained on 90% of the dataset and evaluated on the remaining 10% until all items have appeared in the training and test sets).

The results of this study are summarised in Tables 3.17 and 3.18 for all systems, input types and scenarios, and show the classifier type - self-training using EM clustering algorithm with Naïve Bayes (‘Self’), co-training using Naïve Bayes (‘Co-Naïve’), co-training using Support Vector Machines classifiers (‘Co-SVM’), co-expectation-Maximisation using Support Vector Machines (‘Co-EM’) and the base classifiers Naïve Bayes (‘Base Naïve’) and Support Vector Machines (‘Base SVM’) - along with the version number of the systems and the 3 values corresponding to Precision *(P)*, Recall *(R)* and the F-measure *(F)*. Any technique that performed particularly poorly has been omitted from the results.

The results for semi-supervised learning techniques using input/output pairs under scenario 1 (a proportion of both normal and abnormal data are labelled) for NanoXML are shown in table 3.17. The data for NanoXML shows that self-training outperformed Daikon on

version 1 and 5 but Daikon performed better on versions 2 and 3 (recall that the comparison is with the *best* results from semi-supervised learning, so in this case the self-training results are based on labelling 40% and 50% of the data respectively). Daikon also outperformed all other semi-supervised learning methods with the exception of version 1 where Daikon was beaten by co-Naïve. The results for Siena and Sed (Table 3.18) show that Daikon outperformed all semi-supervised learning methods. Generally, the base classifier always beat the semi-supervised learning techniques on all versions for NanoXML and Siena systems and Daikon too, but this is not surprising as it required a fully labelled training set for its construction.

The second part of table 3.17 presents the results of using semi-supervised learning techniques on input/output pairs augmented with their execution traces for NanoXML under scenario 1 (both normal and abnormal data are Labelled). From the experimental results, it can be observed that practically all the semi-supervised learning approaches outperformed Daikon with the exception of base SVM on version 1 and co-SVM on versions 1, 2 and 5 (which raises serious questions about the applicability of Support Vector Machines for this type of problem and data set). Again the values for self-training are based on a large proportion of labels (50% in this case) but even if this was dropped down to 10% self-training would still have outperformed Daikon over all versions. The results for Siena (the second part of table 3.18) show that semi-supervised learning methods outperformed Daikon and are able to perform on a par with the base classifiers even after being built using just 10% of the Labelled data - quite a surprising accomplishment which also indicates the value of including additional data. On the other hand, from the Sed results (the second part of table 3.18), Daikon outperformed semi-supervised/supervised learning techniques. Moreover, self-training and co-training using with Naïve Bayes performed on a par with the base Naïve Bayes classifier.

The final part of table 3.17 compares the results of using input/output pairs augmented with their execution traces along with scenario 2 for NanoXML (labelling a proportion of the normal results only). Self-training again outperformed Daikon in all cases except version 1 of NanoXML where they performed on a par. Had a smaller proportion of the data been used

for the self-training then it would have been beaten by Daikon (although only for this version NanoXML). The results for Siena (the final part of table 3.18) show the self-training and co-training methods beat Daikon, and also they are able to perform on a par with the base classifier. However, the results for Sed (the final part of table 3.18) show that Daikon beat the semi-supervised/supervised learning techniques. Furthermore, self-training outperformed co-training but the base Naïve classifier performed slightly much better than semi-supervised learning approaches (Naïve Bayes with EM and co-training using Naïve Bayes).

Overall the semi-supervised learning techniques, and especially the self-training method (using Naïve Bayes with EM), performed well in comparison to Daikon in the case where the data consists of input/output pairs augmented with their execution traces (regardless of whether the training used both normal and abnormal data labels or solely normal labels). The relatively poor performance of Daikon may be attributable to the size of the test suites - in all cases Daikon suffered from a high false positive rate and it may have been that the suites were all too small to adequately train the system. However, the same data sets were used for training the semi-supervised learning approaches which may be an advantage for these techniques if they are able to perform well even with a small test suite size.

3.7 Discussion

Test oracles based on semi-supervised learning techniques are more less expensive in comparison to those based on supervised learning techniques as they require a smaller set of labelled training data (as opposed to the large data set required by supervised techniques or the fault-free version employed by invariant detectors). However, oracles based on semi-supervised learning techniques have a lower accuracy in comparison to those based on supervised learning techniques (they have slightly higher false positive rate, and also slightly lower fault detection ability) which is to be expected as the training of the algorithms uses far less labelled data. Semi-supervised approaches are a classic demonstration of the cost-benefit trade-off: a larger set of labelled data is likely to yield a more accurate classifier, and while these techniques are significantly more cost-effective (and practicable) than supervised ap-

Table 3.17 Daikon Versus Semi-supervised Learning Techniques

I/O

Scenario 1	NanoXML Version			
	V1	V2	V3	V5
Classifiers	(P, R, F)	(P, R, F)	(P, R, F)	(P, R, F)
Self	(0.80, 0.80, 0.80)	(0.83, 0.63, 0.72)	(0.85, 0.78, 0.82)	(0.73, 0.76, 0.75)
Co-EM	(0.82, 0.23, 0.36)	(-, -, -)	(-, -, -)	(-, -, -)
Co-SVM	(0.82, 0.23, 0.36)	(-, -, -)	(-, -, -)	(-, -, -)
Co-Naïve	(0.48, 0.48, 0.48)	(0.65, 0.18, 0.28)	(0.63, 0.17, 0.27)	(1, 0.21, 0.35)
Base Naïve	(0.92, 0.79, 0.85)	(0.99, 1, 0.99)	(0.99, 1, 0.99)	(0.90, 0.92, 0.916)
Daikon	(0.40, 0.37, 0.38)	(0.80, 0.75, 0.77)	(0.94, 0.91, 0.92)	(0.63, 0.53, 0.57)

I/O+Traces

Scenario 1	NanoXML Version			
	V1	V2	V3	V5
Classifiers	(P, R, F)	(P, R, F)	(P, R, F)	(P, R, F)
Self	(0.94, 0.94, 0.94)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
Co-SVM	(-, -, -)	(-, -, -)	(1, 1, 1)	(-, -, -)
Co-Naïve	(0.50, 0.40, 0.44)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
Base Naïve	(0.94, 0.94, 0.94)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
Base SVM	(0.94, 0.22, 0.36)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
Daikon	(0.40, 0.37, 0.38)	(0.80, 0.75, 0.77)	(0.94, 0.91, 0.92)	(0.63, 0.53, 0.57)

I/O+Traces

Scenario 2	NanoXML Version			
	V1	V2	V3	V5
Classifiers	(P, R, F)	(P, R, F)	(P, R, F)	(P, R, F)
Self	(0.89, 0.24, 0.38)	(1, 1, 1)	(1, 0.97, 0.98)	(1, 1, 1)
Co-Naïve	(-, -, -)	(1, 0.96, 0.98)	(1, 1, 1)	(1, 0.64, 0.78)
Base Naïve	(0.94, 0.94, 0.94)	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)
Daikon	(0.40, 0.37, 0.38)	(0.80, 0.75, 0.77)	(0.94, 0.91, 0.92)	(0.63, 0.53, 0.57)

Table 3.18 Daikon Versus Semi-supervised Learning Techniques

I/O

Scenario 1	Siena	Sed	
	V2		V5
Classifiers	(P, R, F)	Classifiers	(P, R, F)
Self	(0.19, 0.60, 0.28)	Self	(0.39, 0.86, 0.54)
Co-SVM	(0.11, 0.15, 0.13)	Co-SVM	(-, -, -)
Co-Naïve	(0.37, 0.10, 0.16)	Co-Naïve	(0.85, 0.36, 0.51)
Base Naïve	(0.99, 0.97, 0.93)	Base Naïve	(0.35, 0.86, 0.50)
Daikon	(0.75, 0.71, 0.72)	Daikon	(0.66, 0.60, 0.62)

I/O+Traces

Scenario 1	Siena	Sed	
	V2		V5
Classifiers	(P, R, F)	Classifiers	(P, R, F)
Self	(1, 1, 1)	Self	(0.39. 0.86, 0.54)
Co-SVM	(1, 1, 1)	Co-SVM	(-, -, -)
EM-SVM	(-, -, -)	EM-SVM	(1, 0.07, 0.14)
Co-Naïve	(1, 1, 1)	Co-Naïve	(0.39, 0.84, 0.53)
Base Naïve	(1, 1, 1)	Base Naïve	(0.39, 0.84, 0.53)
Base SVM	(1, 1, 1)	Base SVM	(-, -, -)
Daikon	(0.75, 0.71, 0.72)	Daikon	(0.66, 0.60, 0.62)

I/O+Traces

Scenario 2	Siena	Sed	
	V2		V5
Classifiers	(P, R, F)	Classifiers	(P, R, F)
Self	(1, 1, 1)	Self	(0.41, 0.68, 0.51)
Co-Naïve	(1, 1, 1)	Co-Naïve	(0.42, 0.48, 0.45)
Base Naïve	(1, 1, 1)	Base Naïve	(0.39, 0.84, 0.53)
Daikon	(0.75, 0.71, 0.72)	Daikon	(0.66, 0.60, 0.62)

proaches, there is still work to be done in establishing the ideal ratio of labelled to unlabelled data.

3.8 Conclusions

It was found that self-training (Naïve Bayes with EM) and co-training (Naïve Bayes) classifiers are substantially more effective for detecting failures than co-training (Support Vector Machines) and co-EM (Support Vector Machines) classifiers. Furthermore, in most cases they perform far better under both scenarios as an automated test classifier than Daikon even with small test suites (Daikon usually requires a fault free version of the system with a large and complete test suite to perform well).

It was also found that adding execution trace data can help enormously. This is not really unexpected given that more information is being supplied to the algorithm, but even when very fragmented they improve the accuracy once a fair proportion of the data is labelled. In extreme cases the impact is dramatic (leading to a perfect classifier on from a very small subset of labelled data) but these are relatively rare instances (based on author's assumption).

The results do not give a clear consistent conclusion regards to the size of labelled data on the training set. In some cases (especially in the case that input/output pairs were augmented with their execution traces) labelling just a small proportion of the test cases - as low as 10% - was sufficient to build a classifier that is able to correctly categorise the large majority of the remaining test cases. This has an important implications for the practical use of this technique: when checking the test results from a system a developer need only examine a small proportion of these and use this information to train a learning algorithm to classify the remainder. However, in some cases a large proportion of labelling data was required to achieve higher classification results.

The next chapter describes an investigation of using unsupervised learning techniques (mainly clustering algorithms) where the testers do not have labelled data at all (a real scenario).

Chapter 4

Separating Passing and Failing Test Executions by Clustering Anomalies

4.1 Introduction

The previous chapter showed that semi-supervised learning techniques are useful in many circumstances, but they might be fall short as a solution when labelled execution data is not available (i.e. a small proportion of test data is labelled as a passing or failing test in semi-supervised learning techniques). Fortunately, unsupervised learning techniques can help in the case that no labelled execution data available.

The main focus of this chapter is to investigate the use of clustering based anomaly detection techniques to support the construction of a test oracle by performing two different empirical studies along with varying types of dynamic execution data. In the first empirical study, a range of clustering algorithms are applied to just the input-output pairs of three systems with the primary aim of exploring the feasibility of this approach. The second empirical study extends the first empirical study by augmenting the input/output pairs with their associated execution traces with the aim of improving the performance and accuracy of the approach.

The chapter also explores the optimal number of clusters to employ in relation to the program domain to generate an effective oracle. In addition, the chapter includes observations

on any practical implications that can be found when using unsupervised learning (mainly clustering techniques) in the test oracles area.

4.2 Methodology

4.2.1 Clustering Analysis

Clustering aims to partition a population of objects, each containing various attributes, into groups in such way that objects with similar values are placed in the same cluster, whereas those with dissimilar ones are placed in different clusters. The similarity of objects can be decided by using different distance metrics (discussed in more detail in section 4.2.3). In this thesis the objects of interest are observations from program executions - test inputs and outputs and execution traces - and the aim of clustering is to separate the passing and failing executions.

The main clustering hypothesis is that *"Normal data instances belong to large and dense clusters, while anomalies either belong to small or sparse cluster"* [18]. Clustering techniques based on this hypothesis report objects belonging to clusters whose size *and/or* density is below a threshold as anomalies. By applying clustering techniques under this assumption on a population of execution data that has failures with a non-conformant pattern, it could be possible to cluster the population so as to separate a significant proportion of the failures in small clusters. This would enable the failures to be found by checking the small clusters.

Constructing test oracles via clustering analysis depends in particular on the following decisions: (1) The choice of particular clustering techniques with suitable distance metrics; (2) The number of clusters to be determined; (3) The size of small clusters to be decided. Each of these steps will be discussed in the following sections.

4.2.2 Clustering Algorithms

There is a very large variety of approaches towards clustering and so far this thesis has explored the use of the following algorithms: agglomerative hierarchical clustering, DBSCAN clustering (Density Based Spatial Clustering of Application with Noise) and EM clustering (Expectation-Maximization). Agglomerative hierarchical clustering has been used by other researchers for some similar types of problem and shown to perform reasonably well (e.g. [24], [25], [105], [110]) and also recommended by Witten and Frank [103] as the most suitable solution for nominal and string data (which the coding systems produce for three subject programs section 3.3.2). In contrast, DBSCAN and EM were chosen because of their ability to determine the number of clusters automatically rather than specify them at the outset (one of the limitation of agglomerative hierarchical clustering algorithm) [41].

The following subsections give a brief description of each approach. For further details on the techniques the reader is referred to the work of Han et al. [41] or Witten and Frank [103] for example.

Agglomerative Hierarchical Clustering Algorithm

The agglomerative hierarchical algorithm is an example of a clustering approach that aims to build a hierarchy of objects. The core principle of this type of clustering method is that the objects are more related to nearby objects (as defined by the distance metric) than to objects farther away. A hierarchical clustering method can be either agglomerative or divisive, depending on whether the hierarchical decomposition is formed in a bottom-up (merging) or top-down (splitting) fashion.

Agglomerative hierarchical clustering initially assigns each object to its own cluster, calculates the distance between two clusters, and combines the most similar ones. This process is repeated, building larger and larger clusters at higher levels of the hierarchy, until no close similarity or dissimilarity between two clusters can be found.

Divisive hierarchical clustering operates in the opposite fashion, initially assigning all objects into one cluster and then dividing this main cluster into smaller ones based on object dissimilarity until no further splits can be made.

In both approaches the user has to specify the desired number of clusters as a termination condition.

DBSCAN Clustering Algorithm

DBSCAN is an example of density based clustering approach, grouping together those objects that are close neighbours which allows it to find arbitrarily shaped clusters. Unlike agglomerative hierarchical clustering the number of clusters can be determined automatically (after specifying two key parameters: the minimum number of points in a cluster and the distance between them) and the approach also supports the notion of an outlier (objects not belonging to any cluster). A cluster is defined as containing at least a minimum number of points (MinPts), every pair of points of which either lies within a user specified distance (ε) of each other or is connected by a series of points that each lie within distance of (ε) the next point in the chain. Smaller values of (ε) yield denser clusters because instances must be closer to another to belong to the same cluster. Based on the value of (ε) and the minimum cluster size, it is possible that some objects will not belong to any cluster (these outliers are considered as noise).

EM (Expectation-Maximization) Clustering Algorithm

The *EM* clustering algorithm is an example of probability based clustering approach. In contrast to an approach such as *k-means* clustering, in which a fixed number of clusters (k) is given at the outset and objects are assigned to those clusters so that the means across clusters (for all objects) are as different from each other as possible, *EM* works purely from the set of objects without any a priori information to find the most likely set of clusters from a probabilistic perspective. *EM* operates iteratively to assign data objects to clusters and update the parameters of the probability distributions governing the various clusters until the best model is found.

4.2.3 (Dis)similarity Measures

A range of distance measures were explored such as Euclidean distance, Minkowski distance, Manhattan distance and edit distance in order to establish the most suitable measure for the experiments proper. The first three were similar in terms of the performance and principle. However, edit distance did not perform well and agglomerative hierarchical clustering consistently assigned all input/output pairs into one cluster even when the cluster count was increased. After exploring these various alternatives Euclidean distance was settled on as the measure of (dis)similarity between two objects for agglomerative hierarchical clustering and DBSCAN. The WEKA toolkit [1] used in this thesis computes this by converting all nominal attributes into binary numeric attributes. So, an attribute with *(k)* values is transformed into *(k)* binary attributes (using the one-attribute-per-value approach) [103]. Thus, all attributes values are binary: being either a numeric attribute or a synthetic binary attribute that is treated as numeric. The squared Euclidean distance sums the squared differences between these attributes: a zero sum indicates agreement (similarity), but a non-zero sum suggests a dissimilarity.

The consequence of choosing Euclidean distance is that nominal or categorical data (such as the inputs, outputs and traces used in this thesis) are only considered equal if they are identical. Any form of difference, no matter how small or large, causes them to be considered unequal. This means that two traces may differ in just one method call out of thousands but are considered as different as two that had no method calls in common. This might seem an odd decision but the rationale behind this is that even a slight difference in an execution trace may be indicative of an error. Using other measures would mean such a difference was hardly perceptible and could easily be missed.

4.2.4 Linkage Metrics

In addition to a similarity metric, agglomerative hierarchical clustering requires a linkage metric which is used to determine when clusters should be merged or split. There are three

[1] http://www.cs.waikato.ac.nz/ml/weka/

approaches: *Single Linkage* calculates the minimum distance between an object in one cluster and an object in another, *Average Linkage* computes the mean distance between objects in the two clusters, and *Complete Linkage* is based on the maximum distance between objects. All three are explored in this thesis.

4.2.5 Number of Clusters

For agglomerative hierarchical clustering the number of clusters needs to be provided as parameter. This can clearly have a significant impact: too many clusters results in fragmentation and too few in overgeneralisation. Therefore, a number of different cluster counts were explored based on a percentage of the number of subject program test cases: 1%, 5%, 10%, 15%, 20% and 25%.

The number of clusters for EM is determined automatically by cross validation, a technique often used in classification [103]. A given data set is firstly divided into *(m)* parts. Next, *(m-1)* parts are used to build a clustering model, and the remaining part used to test the quality of the clustering. This process is repeated *(m)* times to derive clusterings of *(k)* clusters by using each part in turn as the test set. The average of the quality measure is taken as the overall quality measure. Then, the overall quality measure with respect to different values of *(k)* is compared to find the best number of clusters that fits the data.

The DBSCAN algorithm uses two specified parameters (ε: the radius parameter, and MinPts: the neighbourhood density threshold – see section 4.2.2) to determine the number of clusters automatically. For this thesis, the parameters which gave the best results are reported in the results and discussion sections (section 4.4.2 and section 4.5.2).

4.2.6 Small Cluster Size

One of the key elements of this thesis is the hypothesis that failures tend to congregate in small clusters. But what is a small cluster? For this thesis, small is defined as less than or equal to the mean of the cluster size (the remainder being considered as large). For the purposes of the experimental evaluation all clusters were examined to determine the

proportion of failures contained therein, but in practice is it envisaged that only small clusters would be inspected and larger ones ignored.

4.3 Experimental Evaluation

Two main experiments were run to evaluate the effectiveness of clustering techniques in separate failing and passing tests.

Experiment 1: In the first experiment the input to the clustering algorithms consisted of just the test case inputs along with their associated outputs.

Experiment 2: The second experiment extended this by adding to the input/output pairs with their corresponding execution trace.

The main hypothesis under investigation being: *"Normal data instances belong to large and dense clusters, while anomalies (failures) either belong to small or sparse clusters"*. In other words, is the execution data which falls outside the clusters or in small (sparse) clusters indicative of bugs? Data about the distribution of failures over clusters, the impact of the number of clusters, the density of clusters, and the number of faults revealed per cluster were analysed to examine this hypothesis. This section gives a brief overview of the experimental set-up and evaluation procedures.

4.3.1 Experimental Set-up

The main components of the experiment were: a set of programs with known failures, a set of test inputs for each program, a way to determine whether an execution of each test was successful or not (passed or failed), and a mechanism for recording the execution trace taken through the program by each test. The seeded versions of the subject programs were run on the inputs to produce the associated outputs, and Daikon [30] was used to obtain the execution traces. The resulting set of input/output pairs was augmented with their associated execution traces, transformed to reduce the volume of data (traces are often very large), and then

analysed using several clustering algorithms. Knowing which data objects corresponding to failed test cases enabled author to determine how well the clustering algorithms performed. Each of these steps along with subject programs were described in more details in the previous chapter (see section 3.3.1 and section 3.3.2).

4.3.2 Evaluation of Clustering Techniques

The performance of the clustering algorithms can be assessed by looking at the way that failures are distributed over the small clusters (the definition of "small" is flexible so what follows is a general definition). To capture more accurately for this thesis, the F-measure was used – a combination measure of Precision and Recall (formulae were introduced in previous chapter section 3.3.3). These measures in turn rely on the concepts of true positives (TP), false positives (FP) and false negatives (FN) which are defined in this context as follows:

TP: A failing test result that appears in a small cluster.

FP: A passing test result that appears in a small cluster.

FN: A failing test result that appears in a large (i.e. not small) cluster.

For this context, Precision is defined as the ratio of "correctly clustered" failures (i.e. failures that appear in small clusters) to the sum of all the entries in the small clusters. Recall is the ratio of "correctly clustered" failures to the total number of true failures (failures appearing in both small and large clusters).

In this thesis, the small clusters were defined as those being of average size or less (i.e. the total number of passing and failing outputs divided by the number of clusters).

To illustrate the process of the evaluation, a small example is introduced which shows how the small cluster size, precision, recall and F-measure are computed. Assume that a system under test generates 21 data points during execution of its set of test cases. The system contains 3 faults (referred to as F1, F2 and F3) which cause failures which appear in the output 4, 4 and 2 times respectively. The remaining 11 test outputs were all passes (there is no need to distinguish amongst these). Again assume that after applying clustering,

6 clusters were created which grouped the outputs as follows: (f1, f2, f3, p, p, p), (p, p, p, p, p), (f1, f2, p, p), (f1, f2, p), (f2, f3), (f1), where fn corresponds to a failure associated with fault n and p corresponds to pass execution. This can be illustrated graphically as shown in Figure 4.1 (where the clusters are sorted in increasing order of size on the y-axis and the “cluster count” legend is just an arbitrary value allocated to a cluster). This representation allows the reader to see the distribution of failures over the clusters.

The key values are computed as follows:

- Small clusters are those of average size or less (i.e. (number of data points)/(number of clusters)). In the above example the average cluster size is (21/6) = 3.5, so the small clusters are all of these containing $\leq$ 3 data points (i.e. clusters 1, 2 and 3).

- Precision: Five of the outputs in the 3 small clusters are failures (TPs) and one is a pass (FP), so
  PR = 5/(5 + 1) = 0.83

- Recall: Five of the outputs in the 3 small clusters are failures (TPs) but 5 failures also ended up being allocated to the “large” clusters (TNs), so
  RE = 5/(5 + 5) = 0.5

- The F-measure is then 2 × (0.83 × 0.5)/(0.83 + 0.5) = 0.62

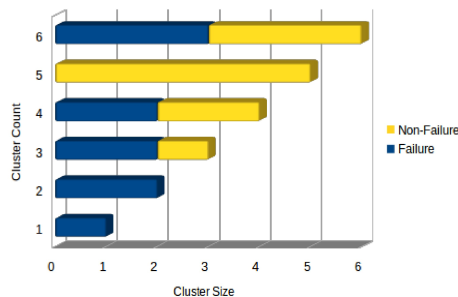

Fig. 4.1 Evaluation Example

4.4 Experiment 1 (Clustering Test Input/Output Pairs): Results And Discussion

This first experiment explored the use of clustering algorithms to cluster data composed of test case inputs and their associated outputs.

4.4.1 Distribution of Failures

The first question to explore is whether failures are distributed in a random pattern or whether they tend to congregate in the smaller clusters as hypothesised. Figures 4.2 – 4.7 show bar charts representing the cluster size and composition for all versions of NanoXML, Siena (faulty version 2), and Sed (faulty version 5) using agglomerative hierarchical clustering with average linkage. The results are interesting and in several cases (NanoXML versions 2 and 3 and Siena version 2) it can be seen that failures in the test input/output pairs population tend to cluster together and these clusters tend to be the smaller ones. This effect is less pronounced in NanoXML versions 1 and 5 where the smallest clusters also tend to contain more of the passing cases. The pattern for Sed is quite different – there are a very large number of small clusters rather than a gradually increasing distribution as in the other cases, and these contain a mixture of both passing and failing cases. Overall there is some support for the main hypothesis behind this work, that failures tend to gravitate towards the smaller clusters but it is by no means universal. The following sections examine this in more detail.

4.4.2 Failures Found verses Cluster Counts and Cluster Sizes

To investigate this observation further the population of input/output pairs that were in small clusters (defined as being of average size or less) and corresponded to failures was examined. Tables 4.1 and 4.2 show, for varying numbers and sizes of clusters over all systems and for the three different linkage metrics that may be used with agglomerative hierarchical clustering (Average, Single and Complete), the percentage of all data points corresponding to failures.

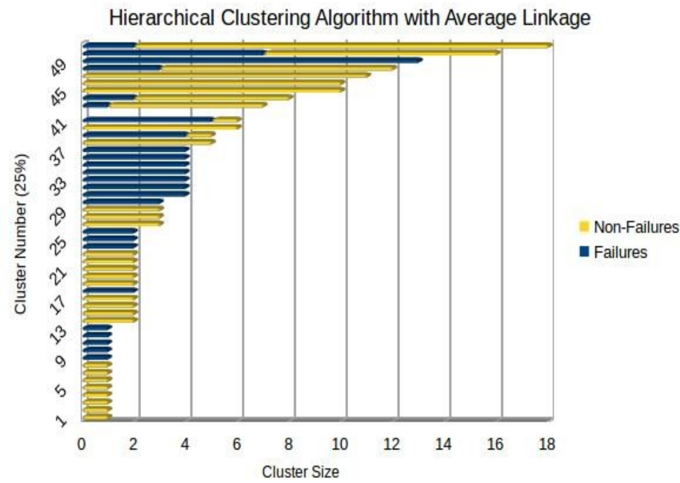

Fig. 4.2 Hierarchical Clustering Algorithm with Average Linkage for NanoXML (Version 1) Using Input/Output Pairs only.

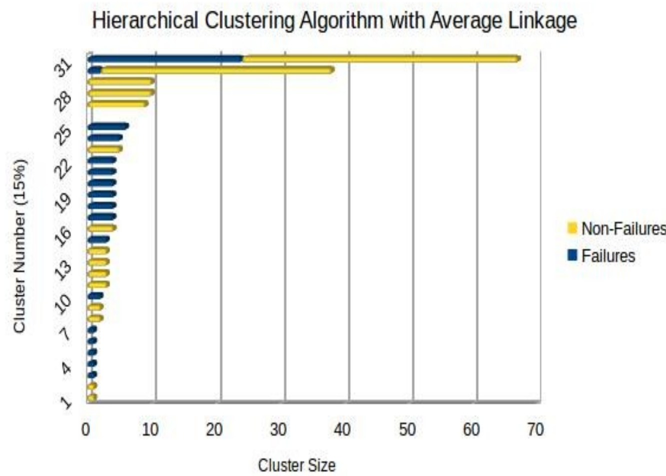

Fig. 4.3 Hierarchical Clustering Algorithm with Average Linkage for NanoXML (Version 2) Input/Output Pairs only.

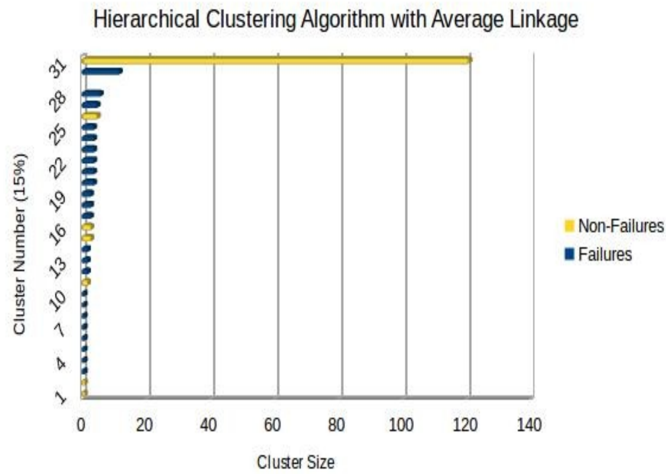

Fig. 4.4 Hierarchical Clustering Algorithm with Average Linkage for NanoXML (Version 3) Input/Output Pairs only.

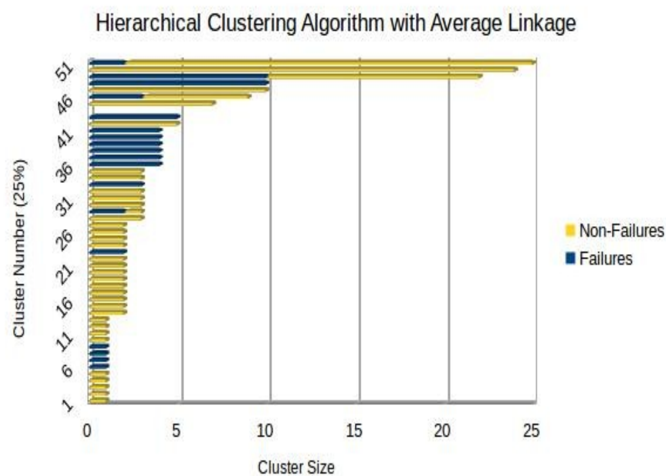

Fig. 4.5 Hierarchical Clustering Algorithm with Average Linkage for NanoXML (Version 5) Input/Output Pairs only.

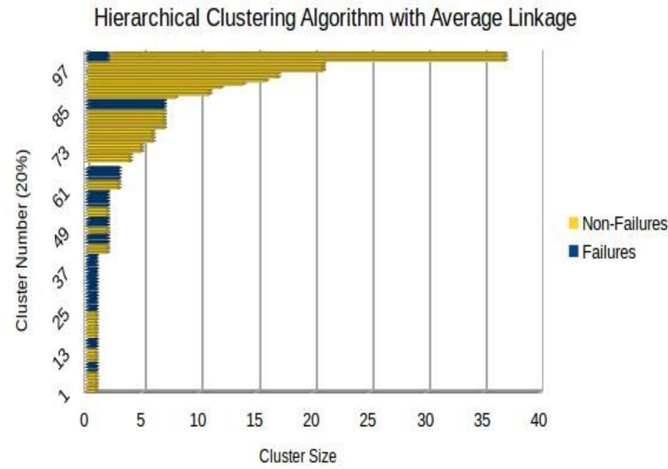

Fig. 4.6 Hierarchical Clustering Algorithm with Average Linkage for Siena (Version 2) Input/Output Pairs only.

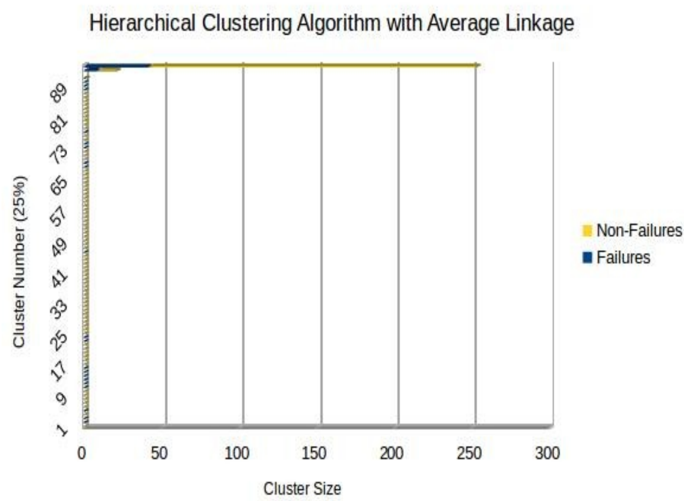

Fig. 4.7 Hierarchical Clustering Algorithm with Average Linkage for Sed (Version 5) Input/Output Pairs Only.

The first column (Cluster Count %) defines the number of clusters the algorithm is charged with creating expressed as a percentage of the number of test cases. So, for NanoXML a value of 10 in the Cluster Count % corresponds to 21 as it has 207 tests, for Siena this would be 50 as it has 494 test cases, and for Sed which has 363 tests it would be 37. The second column (Cluster Size %) is the average size of the clusters created by the algorithms, again expressed in terms of the number of tests. So as the values in Cluster Count % column increase, so do the number of clusters created which leads to a corresponding decrease in the average size of the clusters. The subsequent columns refer to the version number of the program. Note that the faults in Siena changed the same output data in all versions, even though they are distinct faults, so only the results from one version are considered since there is nothing to be gained from examining the other versions.

Considering the results for NanoXML (Table 4.1) the data shows that when the cluster counts are between 15% to 25% of the number of test cases (corresponding to cluster sizes of around 3% of the number of test cases - i.e. around 6 data points for NanoXML), well over 55% of the data points are failures irrespective of which linkage metric is used, and over 60% when the average linkage metric is employed. For Siena (Table 4.2) a similar pattern emerges but the best results are at the higher cluster count levels (20-25%, possibly due to the larger number of test cases which gives an average cluster size of around 4) and tend to be over 70%. The results for Sed (Table 4.2) are less dramatic and although a similar trend is displayed the failure density never reaches 50%, peaking at just over 40% when the complete linkage metric is used with an average cluster size of about 3. From the graphs shown earlier (Figure 4.7) it was observed that Sed contained a very large number of small clusters and only one large cluster, rather than a steadily increasing cluster size which suggests that the data is very fragmented and the algorithm is clearly struggling to form larger groups of data items. This might be addressed by augmenting the data with other types of features such as execution traces or code coverage information. This also could be solved by applying one of the pre-processing/filtering algorithms in the WEKA toolkit to the data in order to remove any unobserved noise.

Even with the results from Sed the findings lend support to the main hypothesis of this work: As the number of clusters increases and their average size decreases, so the failure density of the small (less than average) sized clusters tends to increase. One case where this is not quite true is version 3 of NanoXML where the smallest clusters contained the most failures: the input-output pairs corresponding to failures are so distinct from the rest that they were all grouped into one cluster (an impressive but probably unusual case!).

Tables 4.3 and 4.4 show the results of clustering test inputs and outputs using the Expectation Maximisation and DBSCAN algorithms respectively. Unlike Agglomerative Hierarchical Clustering, neither of these algorithms require the number of clusters to be specified in advance. The results show that EM performs well with all versions of NanoXML but less so with Siena and very poorly with Sed. Interestingly, for NanoXML the number of clusters created is close to the best number when specified for Agglomerative Hierarchical Clustering. The results for DBSCAN are weaker for NanoXML and very poor for Siena but extremely encouraging for Sed, generating both a very high failure density in the smallest clusters and a reasonable F score. In the case of Sed DBSCAN has generated a very large number of small clusters (matching the pattern observed earlier in figure 4.7) – almost twice the number that was explored using Agglomerative Hierarchical Clustering, which confirms the earlier observations about the data being very fragmented.

Although general pattern is for failure intensity to increase as the cluster size decreases, a trend which can also be observed in Figures 4.8 - 4.12 which present the percentage of failures found in the small clusters with different cluster counts in the subject programs (essentially a graphical summary of the data that appears in Tables 4.1 and 4.2), there are cases where the failure intensity peaks and then begins to drop (although not substantially) as the clusters are forced to fragment. An important lesson from this study is that the number of clusters is crucial: too few clusters may be ineffective for the technique but too many may cause the failure intensity to diminish as the clusters are forced to fragment. Identifying the ideal number of clusters (or similarly, the best parameters for algorithms such as DBSCAN) is something which needs further empirical investigation to establish.

Table 4.1 Recall (Failures Found) and F-measure vs. Cluster Size for Hierarchical Clustering with Different Linkage Metrics Using Input/Output Pairs Only.

Single Linkage:

Cluster Details		NanoXML Version			
(% Tests)		V1	V2	V3	V5
Count	Size	(%, F)	(%, F)	(%, F)	(%, F)
1%	50%	(0, 0)	(0, 0)	(0, 0)	(0, 0)
5%	10%	(14, 0.23)	(18, 0.27)	(7, 0.11)	(26, 0.13)
10%	3.5%	(53, 0.63)	(63, 0.71)	(50, 0.59)	(40, 0.45)
15%	3%	(56, 0.62)	(63, 0.65)	(65, 0.71)	(61, 0.60)
20%	2.5%	(56, 0.56)	(63, 0.60)	(72, 0.75)	(66, 0.57)
25%	2%	(56, 0.53)	(63, 0.55)	(74, 0.68)	(66, 0.54)

Average Linkage:

Cluster Details		NanoXML Version			
(% Tests)		V1	V2	V3	V5
Count	Size	(%, F)	(%, F)	(%, F)	(%, F)
1%	50%	(7, 0.07)	(28, 0.03)	(100, 1)	(10, 0.09)
5%	10%	(56, 0.60)	(63, 0.63)	(34, 0.46)	(26, 0.29)
10%	6.25%	(56, 0.58)	(63, 0.63)	(45, 0.57)	(61,0.59)
15%	3.25%	(56, 0.56)	(63, 0.62)	(82, 0.81)	(52, 0.53)
20%	2.5%	(51, 0.51)	(54, 0.51)	(75, 0.70)	(52, 0.43)
25%	2.25%	(65, 0.55)	(61, 0.55)	(75, 0.66)	(61, 0.48)

Complete Linkage:

Cluster Details		NanoXML Version			
(% Tests)		V1	V2	V3	V5
Count	Size	(%, F)	(%, F)	(%, F)	(%, F)
1%	50%	(12, 0.12)	(28, 0.04)	(100, 1)	(10, 0.08)
5%	10%	(12, 0.16)	(29, 0.26)	(20, 0.33)	(26, 0.28)
10%	6.25%	(35, 0.36)	(17, 0.49)	(67, 0.80)	(44, 0.39)
15%	3.12%	(59, 0.56)	(46, 0.41)	(84, 0.81)	(55, 0.44)
20%	2.5%	(51, 0.51)	(54, 0.51)	(75, 0.65)	(52, 0.43)
25%	2.25%	(54, 0.51)	(64, 0.56)	(75, 0.66)	(53, 0.43)

Table 4.2 Recall (Failures Found) and F-measure vs. Cluster Size for Hierarchical Clustering with Different Linkage Metrics Using Input/Output Pairs Only.

Single Linkage:

Cluster Details (% Tests) Count	Size	Siena Version V2 (%, F)	Sed Version Cluster Count	Size	V5 (%, F)
1%	19.8%	(0, 0)	1%	19.8%	(14, 0.26)
5%	4%	(3, 0.03)	5%	6.4%	(23, 0.29)
10%	2%	(40, 0.34)	10%	2.685%	(23, 0.29)
15%	1.21%	(48, 0.44)	15%	1.69%	(23, 0.24)
20%	0.79%	(72, 0.65)	20%	1.22%	(27, 0.23)
25%	0.6%	(60, 0.53)	25%	1%	(36, 0.27)

Average Linkage:

Cluster Details (% Tests) Count	Size	Siena Version V2 (%, F)	Sed Version Cluster Count	Size	V5 (%, F)
1%	19.8%	(0, 0)	1%	19.8%	(0, 0)
5%	4%	(16, 0.13)	5%	6.46%	(9, 0.16)
10%	2%	(41, 0.39)	10%	2.628%	(12, 0.16)
15%	1.21%	(41, 0.37)	15%	1.563%	(18, 0.0.20)
20%	0.79%	(67, 0.61)	20%	1.08%	(25, 0.24)
25%	0.6%	(75, 0.62)	25%	0.808%	(29, 0.25)

Complete Linkage:

Cluster Details (% Tests) Count	Size	Siena Version V2 (%, F)	Sed Version Cluster Count	Size	V5 (%, F)
1%	19.80%	(0, 0)	1%	20%	(9, 0.16)
5%	4%	(33, 0.18)	5%	6.466%	(22, 0.26)
10%	2%	(47, 0.34)	10%	2.71%	(33, 0.38)
15%	1.21%	(66, 0.49)	15%	1.69%	(29, 0.29)
20%	0.79%	(72, 0.65)	20%	1.24%	(36, 0.30)
25%	0.6%	(60, 0.53)	25%	0.968%	(41, 0.30)

Table 4.3 Percentage of Failures and F-measure vs. Cluster Size for EM clustering Algorithm Using Input/Output Pairs Only.

	Cluster Details		EM
Systems	Count	Size	(%, F)
Nanoxml V1	1.94%	25%	(49, 0.65)
Nanoxml V2	2.42%	20.2%	(50, 0.29)
Nanoxml V3	2.42%	20%	(62, 0.43)
Nanoxml V5	1.45%	33%	(64, 0.77)
Siena V2	2.02%	16.66%	(35, 0.22)
Sed V5	2.71%	9.9%	(5, 0.06)

Table 4.4 Percentage of Failures and F-measure vs. Cluster Size for DBSCAN clustering Algorithm Using Input/Output Pairs Only. Note for NanoXML (Epsilon = 0.9 Minpoints = 2) and for Siena and Sed (Epsilon = 1.5 Minpoints = 1).

	Cluster Details		DBSCAN
Systems	Count	Size	(%, F)
Nanoxml V1	19.9%	2.68%	(25, 0.29)
Nanoxml V2	19.9%	2.68%	(22, 0.24)
Nanoxml V3	19.9%	2.70%	(25, 0.26)
Nanoxml V5	19.9%	2.60%	(16, 0.17)
Siena V2	4.45%	4.54%	(3, 0.03)
Sed V5	48.23%	1%	(83, 0.30)

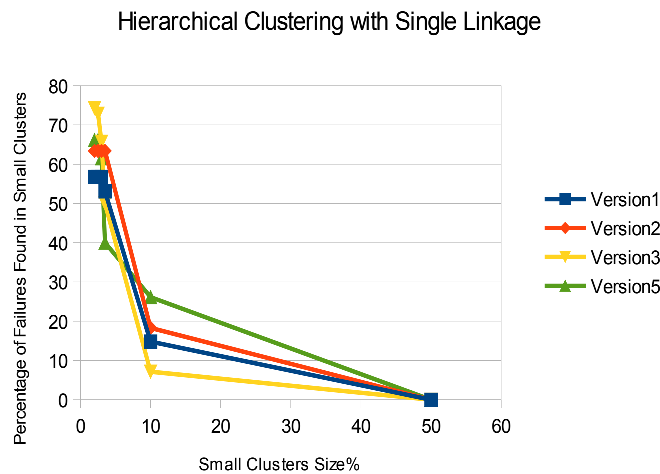

Fig. 4.8 Percentage of Failures Found Over the Smallest Clusters for all NanoXML Versions Using Single Linkage and Input/Output Pairs Only.

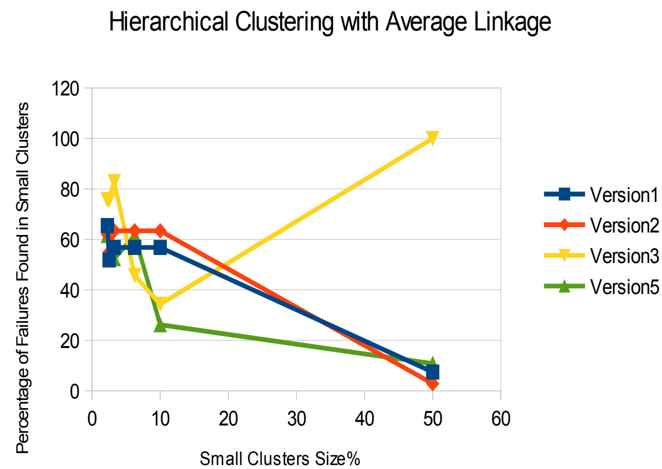

Fig. 4.9 Percentage of Failures Found Over the Smallest Clusters for all NanoXML Versions Using Average Linkage and Input/Output Pairs Only.

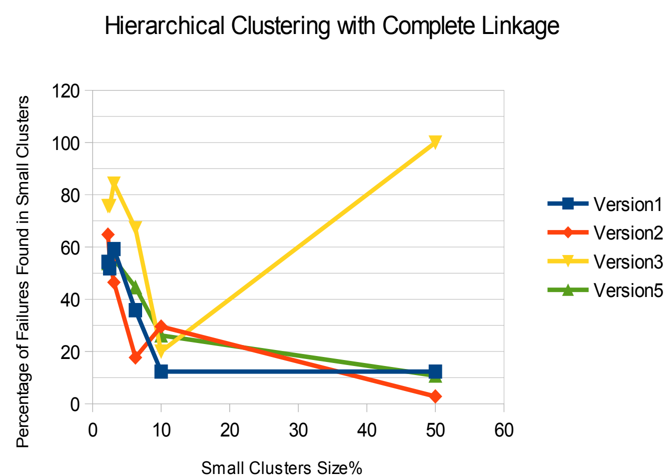

Fig. 4.10 Percentage of Failures Found Over the Smallest Clusters for all NanoXML Versions Using Complete Linkage and Input/Output Pairs Only.

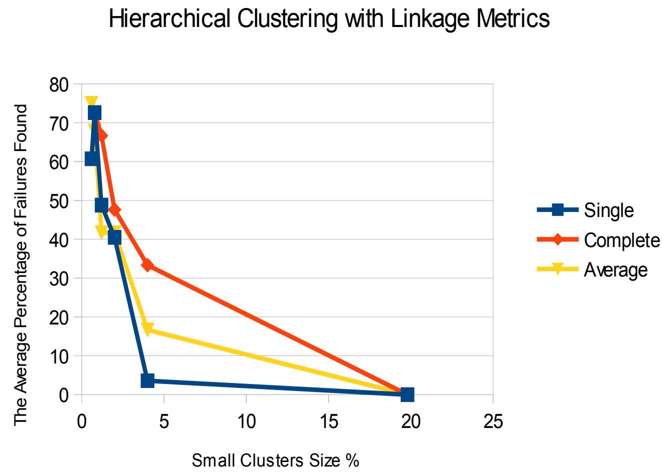

Fig. 4.11 Percentage of Failures Found Over the Smallest Clusters for Siena Version Using Linkage Metrics and Input/Output Pairs Only.

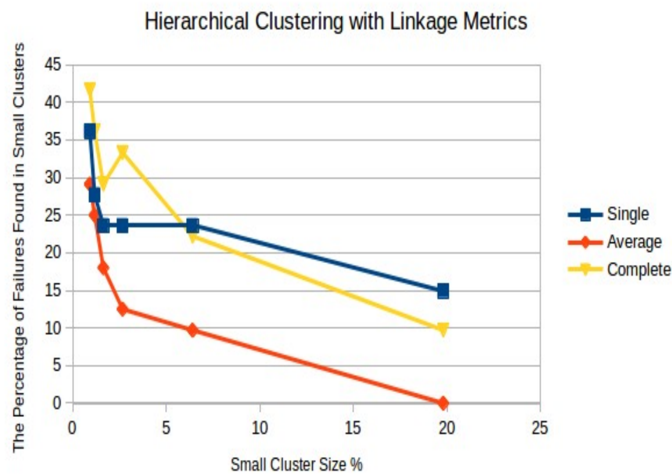

Fig. 4.12 Percentage of Failures Found Over the Smallest Clusters for Sed Version Using Linkage Metrics and Input/Output Pairs Only.

4.4.3 Failure Density of Smallest Clusters

From the perspective of supporting the practising software engineer in their work and also in the construction of a test oracle, the interesting question concerns the return on investment: how many outputs need to be examined before a reasonable number of failures are observed? To answer this the proportion of failing outputs appearing in the smallest sized clusters was examined in more detail. The absence of a fault matrix for Siena makes this very time consuming to compute, therefore only the results for the highest failure density clusters for NanoXML and Sed were calculated. The results of this are summarised in Tables 4.5 and 4.6 and show the cluster size (the 3 values correspond to the absolute size of the cluster, the number of clusters of that size, and the size of the cluster proportional to the test set size) and details of the failures found (the proportion, the actual failures indicated by 'Fn', and the number of occurrences of each failure). Failures associated with a new fault (i.e. not previously encountered) are indicated in bold font. The final column shows the cumulative count of unique faults observed (via their associated failures) over the total number of faults in the system. So, for instance, the first entry of Table 4.5 shows that for Version 1 using 25% of the number of test cases to define the number of clusters, there were 13 clusters each of size 1 corresponding to 0.48% of the number of test cases, containing failures 1 (3 times), 2 and 6 (once each), giving a cumulative count of 3 out of a total of 7.

Table 4.5 shows that on average over all four versions a fair proportion of the failures - 45% (13/29) - are contained within the very smallest clusters (formed from just one or two items). This is encouraging from a test oracle perspective:out of 43 outputs, 23 correspond to failures giving a failure density of 53%. This initially good rate tails off until the cluster size reaches 4 and additional failures appear in the outputs (except for version 5). By this point an average of 66% (19/29) of the failures have appeared in the clusters, albeit at the expense of having to examine more non-failing outputs and encountering duplicate failing outputs (but still giving a failure density of around 59%). This failure density figure, combined with the fact that clusters tend to contain outputs associated with the same failure, means that in practice less than half of the outputs from a small cluster need to be checked before a failing output is encountered.

The results for Sed (Table 4.6) are less impressive but nevertheless encouraging. Even though the failure density is lower than for NanoXML the failures are well represented in the smallest clusters: by examining these 3 out of the 4 failures would be encountered. On the downside the outputs of 62 small clusters (all of size 1) need to be checked but this is still far less work than examining all 370 test outputs.

Of course, there are still additional failing outputs embedded in the larger clusters which can not be ignored. This is clearly a weakness of the approach and one of the main topics of future work is to explore how these can be teased out into smaller clusters. A further feature of the clustering is that there is often number of independent clusters associated with the same failure (separated typically because the input/output pairs have different attribute values). This is also a challenge since finding the same failure appearing in several clusters can be quite frustrating for the individual charged with the task of checking outputs. Merging them together is not the answer as this will typically result in a larger cluster which may escape scrutiny, so some way of indicating similarity between them needs to be explored.

4.5 Experiment 2 (Clustering Test Input/Output Pairs and Execution Traces): Results And Discussion

A second experiment was run to investigate if collecting additional data in the form of the execution traces associated with each test case would improve the accuracy of the clustering performed in the first experiment by increasing in particular the failure density of the small clusters. Since this trace data can be quite extensive it was compressed as described in Section 3.3.2 in previous chapter. Apart from collecting and including this additional trace data in the clustering all other aspects of this experiment were identical to the previous experiment.

4.5.1 Distribution of Failures over Clusters

Again, the first major question to explore is whether failures are distributed in a random pattern over the clusters or whether they gravitate towards the small clusters as hypothesised.

Table 4.5 Failure Distribution over less than Average Sized Clusters for Nanoxml Using Input/Output Pairs Only.

Version 1 (25%)		
Cluster Size	Failures Found	Cumulative
1, 13, 0.48%	(**F1**:3, **F2**:1, **F6**:1)	3/7
2, 13, 0.97%	(F1:4, F2:2, F6:2)	3/7
3, 4, 1.45%	(F6:3)	3/7
4, 8, 1.94%	(F2:16, **F5**:8, **F7**:8)	5/7
Version 2 (15%)		
Cluster Size	Failures	Cumulative
1, 7, 0.48%	(**F1**:3, **F2**:2, **F6**:1)	3/7
2, 3, 0.97%	(F6:2)	3/7
3, 5, 1.45%	(F6:3)	3/7
4, 6, 1.94%	(F2:8, **F5**:8, **F7**:8)	5/7
5, 2, 2.42%	(F2:5)	5/7
6, 1, 2.91%	(F2:6)	5/7
Version 3 (15%)		
Cluster Size	Failures	Cumulative
1, 10, 0.48%	(**F1**:4, **F2**:1, **F3**:1, **F4**:2, **F6**:1)	5/7
2, 4, 0.97%	(F1:1, F4:2, F6:2)	5/7
3, 5, 1.45%	(F4:6, F6:3)	5/7
4, 6, 1.94%	(F2:8, **F5**:8, **F7**:8)	7/7
5, 2, 2.42%	(F2:5)	7/7
6, 1, 2.91%	(F2:6)	7/7
Version 5 (25%)		
Cluster Size	Failures	Cumulative
1, 13, 0.48%	(**F1**:3, **F2**:1)	2/8
2, 14, 0.97%	(F1:2, F2:2)	2/8
3, 8, 1.45%	(F2:3)	2/8
4, 7, 1.94%	(F2:28)	2/8

Table 4.6 Failure Distribution over less than Average Sized Clusters for Sed Version 5 Using Input/Output Pairs Only.

Complete Linkage (25%)		
Cluster Size	Failures Found	Cumulative
1, 62, 0.19%	(**F1**:7, **F2**:4, **F3**:7)	3/4
2, 16, 0.38%	(F1:2, F3:4)	3/4
3, 5, 0.57%	(F2:3, F3:3)	3/4
4, 2, 0.76%	(-)	-/4
5, 1, 0.96%	(-)	-/4

To examine this a sample of the results are shown visually – these are only a selection of the best results for all versions of the subject programs, but the full set is available online [2]. Figures 4.13 to 4.18 show bar charts of the cluster composition for NanoXML (all faulty versions), Siena (faulty version 2) and Sed (faulty version 5), where failing outputs are coloured blue and passing ones yellow. In these cases the cluster count for NanoXML is set at 15% of the number of test cases (producing approximately 30 clusters), 20% for Siena (producing just under 100 clusters) and 25% for Sed (producing just over 90 clusters). In all cases the results are using agglomerative hierarchical clustering (DBSCAN and EM clustering algorithms were also used but tended to perform relatively poorly – something which is explored in more detail later).

It can be seen from these results that as in experiment 1 the failure data do tend to cluster together and these clusters are the smaller ones in most cases. There are some exceptions to this: for example for NanoXML version 5 the very smallest clusters are dominated by non-failing outputs whereas the converse is true for the other versions, and in all cases of NanoXML some failures creep into the largest clusters. The results for Siena are more consistent with a clear tendency for failures to gravitate towards the small clusters and away from the larger ones. The results for Sed are similar to experiment 1 – many small clusters and one large cluster but this time with a few intermediate-sized ones. It must be stressed that these are selected, and very high-level, results (although others reflect a similar pattern) but it would seem that a substantial number of failures congregate in small clusters. The detailed composition of these small clusters is examined in more detail in the next section.

4.5.2 Failure Composition of Small Clusters

This apparent observed tendency for failures to gravitate towards the smaller clusters need to be explored in more detail: the precise degree to which it occurs; the impact of the different clustering algorithms and parameters (especially the number of clusters); and particularly the way that multiple failures are distributed (for example, in the case of several failures do they all appear in the small clusters or is one failure dominant?). To explore this principle further

[2] A complete sets of results can be found at: http://personal.strath.ac.uk/rafig.almaghairbe/

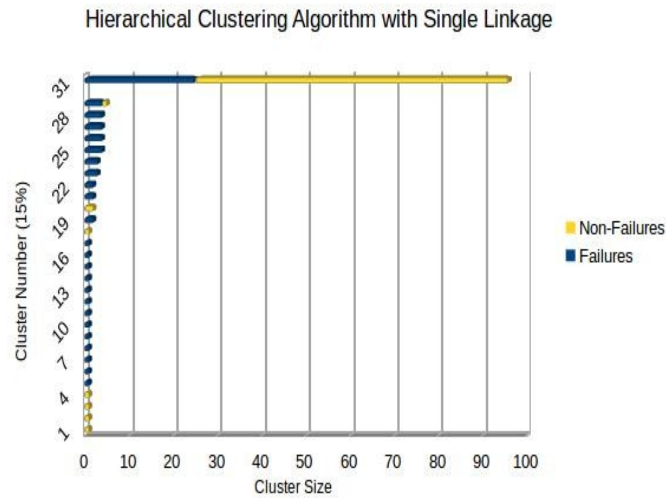

Fig. 4.13 Hierarchical Clustering Algorithm with Single Linkage for NanoXML (Version 1) Using Input/Output Pairs Augmented with Execution Traces.

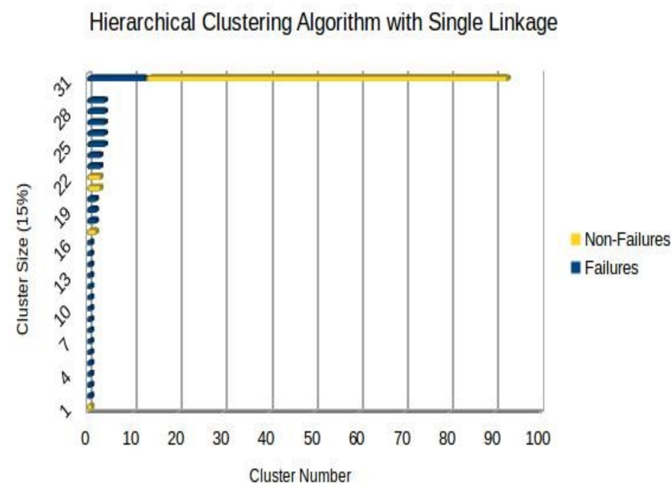

Fig. 4.14 Hierarchical Clustering Algorithm with Single Linkage for NanoXML (Version 2) Using Input/Output Pairs Augmented with Execution Traces.

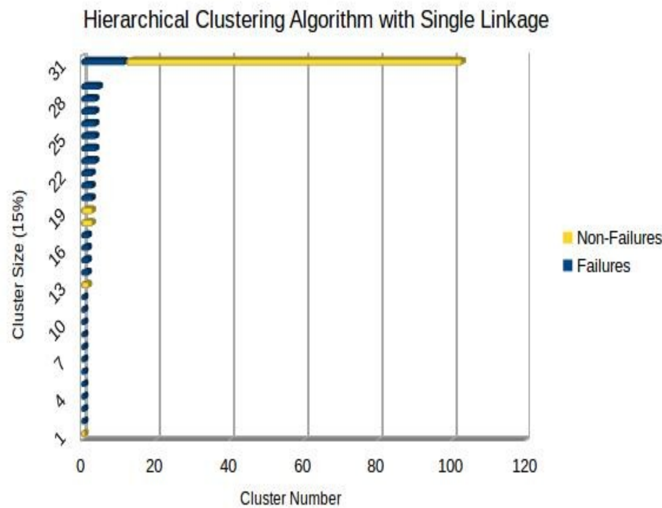

Fig. 4.15 Hierarchical Clustering Algorithm with Single Linkage for NanoXML (Version 3) Using Input/Output Pairs Augmented with Execution Traces.

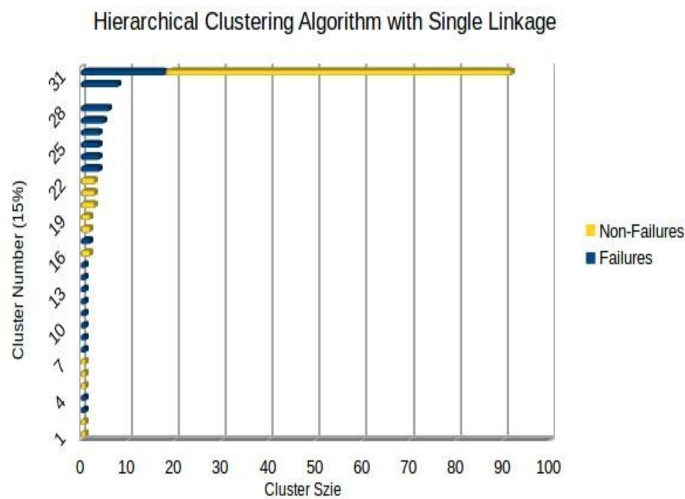

Fig. 4.16 Hierarchical Clustering Algorithm with Single Linkage for NanoXML (Version 5) Using Input/Output Pairs Augmented with Execution Traces.

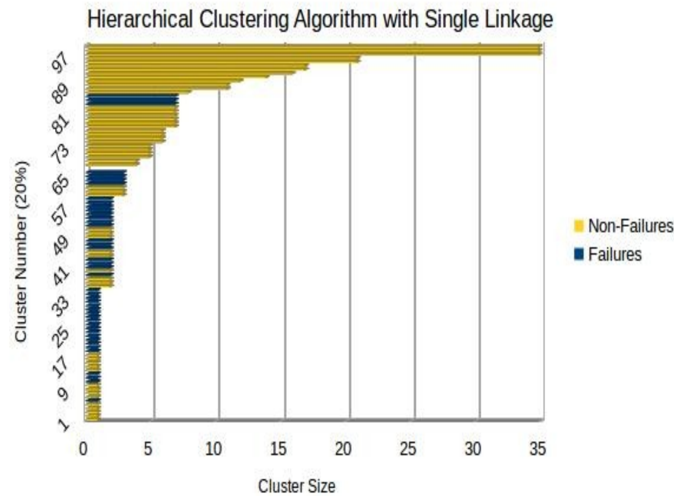

Fig. 4.17 Hierarchical Clustering Algorithm with Single Linkage for Siena (Version 2) Using Input/Output Pairs Augmented with Execution Traces.

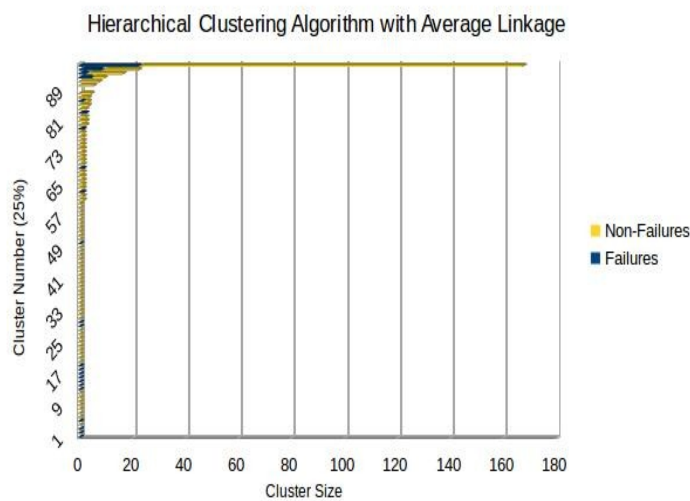

Fig. 4.18 Hierarchical Clustering Algorithm with Average Linkage for Sed (Version 5) Using Input/Output Pairs Augmented with Execution Traces.

the population of the small clusters (defined as being of average size or less) was examined for each of the algorithms, identified the percentage of these clusters that correspond to failures, and also used the F-measure to answer the point about the way that multiple failures are clustered.

Tables 4.7, 4.8 and 4.9 show, for NanoXML, Siena and Sed respectively, the results of applying agglomerative hierarchical clustering for different linkage metrics with varying numbers of clusters. The tables show the percentage of all data points in small (less than average-sized) clusters that correspond to failures, and the F-measure for the small clusters. The percentage figure gives an indication of the failure density and the F-measure adds to this by considering the range of faults that are revealed by failures that appear in the small clusters (for NanoXML there are 7 faults in versions 1-3 and 8 in version 5). The first column (Count) defines the number of clusters the algorithm is charged with creating expressed as a percentage of the number of test cases. The second column (Size) is the average size of the clusters again in terms of the number of test cases. The cluster count figure has to be supplied as a parameter whereas the size figure is a consequence of the number of clusters and is not controllable. The subsequent columns refer to the version number of the programs and % and F refer to the percentage of failures and the F-measure.

The data for NanoXML shows an interesting bi-modal response: the best results occur when there is either the smallest number of clusters (1% which corresponds to 2 clusters) or when the cluster counts range between 10% to 25% of the number of test cases (yielding between 20 and approximately 50 clusters). When the cluster count is very small the algorithm will generate two large clusters (these will be of similar size but the smaller one is always treated as the small cluster) and in some cases one of these is composed entirely of failures and the other of passing outputs (those where the F-measure has a value of 1) – in other words the algorithm has managed to perfectly separate the passing and failing executions (the reason behind this impressive clustering were investigated and discussed in more detail in section 3.4.2 in previous chapter - also see table 3.8).

As the cluster count increases so the results tend to drop quite dramatically until they pick up at around the 15% level (± 5%) before tailing off again. In this range the average

small cluster sizes are between 2.73% to 6.39% of the number of test cases – around 5 to 13 elements and it is worth noting that well over 60% – sometimes far more – of the data points are failures. This again lends support to the experimental hypothesis behind this work that failures tend to congregate in small clusters. Another notable point is the fact that the F-measure tends to vary in line with the percentage of failures (and in all but one case the highest F-measure is also the highest percentage of failures), indicating that the failures associated with the numerous faults are evenly distributed across the small clusters. This is important as it could have been the case that the small clusters were dominated by a small and unrepresentative number of failures. The exact composition of these clusters will be explored in more detail later. It is also notable that both the linkage metrics and the versions of the program have an impact on the results, but the best overall and most stable results are produced by using the single linkage metric with a cluster count set at 15% of the number of test cases.

The results for Siena (4.8) tend to follow a similar pattern: in some cases the smallest number of clusters (5) tend to perform well and again manage to perfectly separate the data (once again this result is down to the passing and failing outputs being completely separable by their traces), but in other cases (with the single linkage metrics) they perform very poorly. The data for Siena also supports the key hypothesis behind this work with the cluster counts between 5% to 25% of the number of test cases consisting of over 70% failures. As for NanoXML the linkage metrics influence the findings, with the single linkage producing the least consistent results and the complete linkage the best.

The picture for Sed is similar to that for experiment 1 – a gradual increase in failure density and F-measure as the cluster size drops but a much lower overall failure density value than was observed in the other two projects. Including the trace information has not produced any dramatic results as with NanoXML and Siena as there is no dominant pattern of traces arising from failing executions (Sed has 295 distinct traces - see table 3.8 in the previous chapter).

The results of using EM and DBSCAN to perform the clustering are shown in Tables 4.10 and 4.11. The first column (systems) defines the subject programs with their version

number. The second and third columns identify, as in the previous tables, the number of clusters and the average small cluster size again in terms of the percentage of test cases. The key difference in this case is that the cluster count is determined automatically by the algorithm. The final column shows the percentage of failures in the small clusters and the F-measure for each algorithm. With the exception of version 1, DBSCAN performed well on NanoXML: for version 2 the result was equal to the best found using agglomerative hierarchical clustering, and versions 3 and 5 were close to the best. It is also notable that the cluster count chosen was 15% – identified as the best compromise for agglomerative hierarchical clustering. The trace information in version 1 is far more diverse which may explain the less impressive performance in this case. The results for Siena are consistent but far inferior to those produced by most of the different cluster size parameters using agglomerative hierarchical clustering. Sed produced the most disappointing results for this algorithm – far worse than when it was operating on test input and outputs alone which suggests that the clustering seems to be fragmenting the data further. The findings for EM are very disappointing, with the odd exception of NanoXML Version 1. In the majority of cases the algorithm failed to apportion any of the failures into the smallest clusters and also elected to use a very small number of clusters.

4.5.3 Fault Density of Smallest Clusters

As in experiment 1 the practical utility of the approach and the return on investment was explored: how many outputs need to be examined before a reasonable number of failures and associated faults are observed? To answer this the precise composition of failing data appearing in the smallest sized clusters was examined in more detail – in other words which failing outputs appeared in which clusters.

The results of this analysis for NanoXML (with a clustering size of 15% using agglomerative hierarchical clustering) are shown in Table 4.12 (which takes the same form as Table 4.5 in Section 4.4.3). The NanoXML results show a number of failures appearing in the smallest clusters with additional ones appearing after examining just a few more clusters (with the exception of version 5). This is an important finding as it suggests that those failures which

Table 4.7 Percentage of Failures Found (Recall) and F-measure vs. Cluster Size for Hierarchical Clustering with Different Linkage Metrics Using Input/Output Pairs Augmented with Execution Traces.

Single Linkage:

Cluster Details		NanoXML Version			
(% Tests)		V1	V2	V3	V5
Count	Size	(%, F)	(%, F)	(%, F)	(%, F)
1%	50%	(64, 0.72)	(0, 0)	(2, 0.03)	(0, 0)
5%	11.95%	(5, 0.09)	(5, 0.08)	(34, 0.5)	(40, 0.53)
10%	6.37%	(47, 0.59)	(40, 0.43)	(66, 0.79)	(41, 0.53)
15%	5.03%	(64, 0.73)	(78, 0.84)	(82, 0.84)	(60, 0.63)
20%	3.31%	(57, 0.65)	(68, 0.72)	(73, 0.75)	(60, 0.6)
25%	2.76%	(58, 0.62)	(68, 0.70)	(69, 0.70)	(44, 0.44)

Average Linkage:

Cluster Details		NanoXML Version			
(% Tests)		V1	V2	V3	V5
Count	Size	(%, F)	(%, F)	(%, F)	(%, F)
1%	50%	(0, 0)	(100, 0.94)	(84, 0.91)	(100, 1)
5%	12.04%	(7, 0.11)	(6, 0.11)	(14, 0.24)	(9, 0.14)
10%	6.47%	(44, 0.56)	(48, 0.64)	(34, 0.47)	(26, 0.35)
15%	4.30%	(64, 0.73)	(78, 0.81)	(68, 0.74)	(60, 0.61)
20%	3.27%	(64, 0.58)	(68, 0.61)	(75, 0.71)	(60, 0.56)
25%	2.74%	(58, 0.55)	(70, 0.57)	(69, 0.64)	(46, 0.37)

Complete Linkage:

Cluster Details		NanoXML Version			
(% Tests)		V1	V2	V3	V5
Count	Size	(%, F)	(%, F)	(%, F)	(%, F)
1%	50%	(0, 0)	(100, 1)	(100, 0.98)	(100, 1)
5%	11.94%	(0, 0)	(3, 0.004)	(26, 0.40)	(43, 0.60)
10%	6.39%	(2, 0.02)	(3, 0.03)	(49, 0.65))	(70, 0.83)
15%	4.27%	(31, 0.36)	(10, 0.08)	(85, 0.88)	(70, 0.74)
20%	3.37%	(52, 0.55)	(45, 0.45)	(76, 0.72)	(61, 0.60)
25%	2.73%	(58, 0.57)	(70, 0.69)	(69, 0.67)	(46, 0.42)

Table 4.8 Percentage of Failures Found (Recall) and F-measure vs. Cluster Size for Hierarchical Clustering with Different Linkage Metrics Using Input/Output Pairs Augmented with Execution Traces.

Single Linkage:

Cluster Details		Siena Version
(% Tests)		V2
Count	Size	(%, F)
1%	19.8%	(0, 0)
5%	4%	(17, 0.21)
10%	1.99%	(34, 0.31)
15%	1.21%	(61, 0.47)
20%	0.79%	(75, 0.66)
25%	0.6%	(60, 0.48)

Average Linkage:

Cluster Details		Siena Version
(% Tests)		V2
Count	Size	(%, F)
1%	20.13%	(100, 0.96)
5%	4%	(23.80, 0.19)
10%	1.98%	(75, 0.65)
15%	1.20%	(75, 0.57)
20%	0.81%	(71, 0.60)
25%	0.6%	(75, 0.64)

Complete Linkage:

Cluster Details		Siena Version
(% Tests)		V2
Count	Size	(%, F)
1%	20%	(100, 1)
5%	4.04%	(100, 0.89)
10%	1.99%	(60, 0.56)
15%	1.27%	(71, 0.52)
20%	0.79%	(75, 0.66)
25%	0.6%	(75, 0.62)

Table 4.9 Percentage of Failures Found (Recall) and F-measure vs. Cluster Size for Hierarchical Clustering with Different Linkage Metrics Using Input/Output Pairs Augmented with Execution Traces.

Single Linkage:

Cluster Details (% Tests) Count	Size	Sed Version V5 (%, F)
1%	19.8%	(24, 0.31)
5%	6.4%	(27, 0.32)
10%	2.6%	(16, 0.21)
15%	1.65%	(24, 0.24)
20%	1.2%	(34, 0.28)
25%	1%	(34, 0.18)

Average Linkage:

Cluster Details (% Tests) Count	Size	Sed Version V5 (%, F)
1%	19.8%	(24, 0.31)
5%	7.6%	(13, 0.20)
10%	2.68%	(16, 0.20)
15%	1.67%	(25, 0.25)
20%	1.2%	(36, 0.29)
25%	1%	(39, 0.26)

Complete Linkage:

Cluster Details (% Tests) Count	Size	Sed Version V5 (%, F)
1%	20%	(12, 0.19)
5%	6.6%	(22, 0.28)
10%	2.71%	(33, 0.37)
15%	1.69%	(28, 0.27)
20%	1.22%	(31, 0.25)
25%	1%	(37, 0.29)

Table 4.10 Percentage of Failures Found (Recall) and F-measure vs. Cluster Size for DBSCAN clustering Algorithm Using Input/Output Pairs Augmented with Execution Traces.

	Cluster Details		DBSCAN
Systems	Count	Size	(%, F)
Nanoxml V1	50%	1.425%	(32, 0.31)
Nanoxml V2	15%	5.1%	(78, 0.81)
Nanoxml V3	15%	4.08%	(79, 0.81)
Nanoxml V5	15%	3.78%	(69, 0.67)
Siena V2	6%	3.33%	(53, 0.48)
Sed V5	8%	3.5%	(9, 0.10)

Table 4.11 Percentage of Failures Found (Recall) and F-measure vs. Cluster Size for EM clustering Algorithm Using Input/Output Pairs Augmented with Execution Traces.

	Cluster Details		EM
Systems	Count	Size	(%, F)
Nanoxml V1	2.41%	20%	(40, 0.42)
Nanoxml V2	1.93%	25.25%	0
Nanoxml V3	2.41%	19.8%	(5, 0.09)
Nanoxml V5	1.44%	33.33%	0
Siena V2	1.41%	14.28%	0
Sed V5	1%	33.33%	(9, 0.08)

are going to be observed tend to appear relatively early in the ordering of clusters. This has important practical implications: collectively these smallest clusters correspond to between 25% and 30% of the total output of the system, and the observed failures appear in an even smaller grouping, which means that the majority of failures in a system can be identified by looking at between one-fifth and one-quarter of the output – a substantial saving in effort for the developer.

The results for Siena are included in Table 4.13 although since Siena contains just the one fault the impact is less pronounced. However, it does show that the observed failures also tend to be concentrated early on in the small clusters and have the same implications as the NanoXML results. The findings for Sed are shown in Table 4.14. The pattern is similar to the first experiment but the number of clusters to be examined has dropped very slightly. Again there are clear practical benefits: 75% of the program's failures are concentrated in about 16% of its results.

4.5.4 Impact of Failure Density

One key factor in this study is the failure density. As mentioned in Section 3.3.1, this is between 31%-39% for NanoXML and 17% for Siena. This failure rate is a factor of the combination of test cases supplied for the two systems and the nature of the faults embedded within the systems. However, in practical terms this may be too high. The expectation is that this approach would be applied to a relatively mature system which may not have many obvious faults, and consequently a much smaller failure rate. Furthermore, an assumption behind anomaly detection is that anomalous events are relatively rare whereas in these experiments the failure rate has been fairly high, so may represent a difficult case for the successful application of clustering techniques. To explore the impact of this two versions of two of the systems were taken – NanoXML V3 and Siena V2 (Sed was ignored as it demonstrated a similar failure rate to Siena) – and randomly pruned out fault revealing test cases to systematically reduce the failure rates to 10%, 5% and 1% for each system.

The results for this part of the investigation are shown in Tables 4.15 and 4.16 which, for each system, shows the cluster size, again in terms of the percentage of test cases (but

Table 4.12 Failure Distribution over less than Average Sized Clusters for Nanoxml

Version 1 (15%)		
Cluster Size	Failures Found	Cumulative
1, 10, 0.67%	(**F1**:2, **F2**:2, **F6**:1)	3/7
2, 3, 1.34%	(F1:2, F6:2)	3/7
3, 2, 2.01%	(F2:3, F6:3)	3/7
4, 5, 2.68%	(F2:4, **F5**:8, **F7**:8)	5/7
5, 1, 3.35%	(F2:4)	5/7
6, 1, 4.02%	(F2:6)	5/7
Version 2 (15%)		
Cluster Size	Failures	Cumulative
1, 8, 0.67%	(**F1**:3, **F2**:2, **F6**:2)	3/7
2, 4, 1.34%	(F1:2, F6:4)	3/7
3, 3, 2.01%	(F2:3)	3/7
4, 5, 2.68%	(F2:4, **F5**:8, **F7**:8)	5/7
5, 1, 3.35%	(F2:5)	5/7
6, 1, 4.02%	(F2:6)	5/7
Version 3 (15%)		
Cluster Size	Failures	Cumulative
1, 8, 0.59%	(**F1**:3, **F2**:1, **F4**:2, **F6**:1)	4/7
2, 4, 1.18%	(F1:2, F4:2, F6:2)	4/7
3, 5, 1.77%	(F4:3, F6:6)	4/7
4, 6, 2.36%	(F2:8, **F5**:8, **F7**:8)	6/7
5, 1, 2.95%	(F5:5)	6/7
6, 1, 3.55%	(F2:6)	6/7
Version 5 (15%)		
Cluster Size	Failures	Cumulative
1, 7, 0.62%	(**F1**:1, **F2**:1)	2/8
2, 4, 1.25%	(F1:2)	2/8
3, 3, 1.88%	(-)	2/8
4, 6, 2.51%	(F2:24)	2/8
5, 1, 3.14%	(F1:5)	2/8
6, 1, 3.77%	(F2:6)	2/8

Table 4.13 Failure Distribution over less than Average Sized Clusters for Siena

Version 2 (5%)		
Cluster Size	Failures Found	Cumulative
1, 5, 0.20%	(-)	-/1
2, 1, 0.40%	(**F**:2)	1/1
3, 8, 0.60%	(F:24)	1/1
4, 1, 0.80%	(F:4)	1/1
6, 3, 1.21%	(F:12)	1/1
8, 1, 1.61%	(-)	1/1
9, 1, 1.82%	(F:9)	1/1
11, 3, 2.22%	(F:33)	1/1

Table 4.14 Failure Distribution over less than Average Sized Clusters for Sed Version 5

Complete Linkage (25%)		
Cluster Size	Failures Found	Cumulative
1, 59, 0.19%	(**F1**:8, **F2**:1, **F3**:7)	3/4
2, 17, 0.38%	(F2:2, F3:4)	3/4
3, 6, 0.57%	(F3:3)	3/4
4, 2, 0.76%	(-)	-/4
5, 1, 0.96%	(-)	-/4

note that the actual number of clusters will decrease as the failure rate decreases as test cases are being pruned from the suite), and the percentage of failures found and F-measure over the small clusters for failure rates of 10%, 5% and 1%. Both systems exhibit a similar distinctive pattern: as the failure rate decreases the recall (percentage of failures found) tends to remain high but the F-measure drops as the cluster count increases. The reason behind this is that with an increase in the number of clusters the false positive rate also increases as more passing tests become classified into the small clusters. This also has an important practical implication for this technique suggesting that if the system under investigation is expected to have a low failure rate then the cluster count (if specified as a parameter) should be very small, but as the expected failure rate increases then so should the number of clusters.

Table 4.15 NanoXML V3 with Reduced Failure Rate

	10%		5%		1%	
Cluster Count	Failures Found	F-measure	Failures Found	F-measure	Failures Found	F-measure
1%	100%	1	100%	1	100%	1
5%	100%	1	100%	1	100%	0.43
10%	100%	0.88	100%	0.60	100%	0.11
15%	100%	0.72	100%	0.43	100%	0.09
20%	100%	0.59	100%	0.33	100%	0.07
25%	100%	0.56	100%	0.34	100%	0.09

Table 4.16 Siena with Reduced Failure Rate

	10%		5%		1%	
Cluster Count	Failures Found	F-measure	Failures Found	F-measure	Failures Found	F-measure
1%	100%	1	100%	0.94	85%	0.91
5%	100%	0.78	0%	0	100%	0.14
10%	100%	0.69	100%	0.43	100%	0.13
15%	52%	0.43	100%	0.4	100%	0.14
20%	100%	0.64	100%	0.42	100%	0.11
25%	100%	0.58	100%	0.31	100%	0.07

4.6 Statistical Test for Clustering Hypothesis

The experimental results in the thesis gave an evidence which support the clustering hypothesis behind this work where in several cases small clusters (less than average sized) contained more than 60% of failures (and often a substantially higher proportion). The hypothesis test will be employed on several experimental data for all subject programs used in this thesis (see Table 4.17) to test and see the impact of clustering approach. Note that, if the clustering approach has no impact then the failures will be evenly distributed throughout the clusters irrespective of their size. A null hypothesis, alternative hypothesis and significance level are stated as follows:

- The Null hypothesis (H_0) $P <= (FN/TZ)$ (Proportion of failures found on small cluster is less than or equal to FN/TZ). Where FN is all failures on test suite, and TZ is the size of test suite.
- The Alternative hypothesis (H_1) $P > (FN/TZ)$ (Proportion of failures found on small cluster is more than FN/TZ).
- The significance level is 0.05, and Z-Test method with a right one tailed test is selected.

The hypothesis tests are rejected in all cases for first study when input/output pairs used as input to the clustering algorithm (Test 1 to Test 6 in Table 4.17). In addition, the hypothesis tests were rejected in all cases for second study when input/output pairs with execution traces used as input to the clustering algorithm (Test 7 to Test 12 in Table 4.17). It was observed the results statistically significant in all cases where $p-value$ lower than the significance level (0.05) - (see Table 4.18). This can be generalised to the rest of experimental data.

4.7 Clustering Algorithms versus Daikon

Although it is not fair to make a direct comparison between Daikon and the approach presented in this chapter as they address the oracle problem in quite different ways (Daikon by building up assertions using a “clean” reference model of the system and then automatically

Table 4.17 Tested Proportion of Failures Found on the Smallest Clusters on Several Experiments

Test Number	System	Failures Found	Population Size	Clusters Number
Test 1	NanoXML V1	65%	101	25%
Test 2	NanoXML V2	63%	72	15%
Test 3	NanoXML V3	82%	73	15%
Test 4	NanoXML V5	52%	99	25%
Test 5	Siena V2	67%	100	20%
Test 6	Sed V5	29%	92	25%
Test 7	NanoXML V1	64%	53	15%
Test 8	NanoXML V2	78%	56	15%
Test 9	NanoXML V3	82%	66	15%
Test 10	NanoXML V5	82%	59	15%
Test 11	Siena V2	82%	104	20%
Test 12	Sed V5	29%	130	25%

Table 4.18 The Results of Z-Test and p-value on Tested Failures Proportion

Test Number	z	$p-value$	Note
Test 1	5.531	9.07716×10^{-8}	(H_0) rejected - significant results
Test 2	5.272	3.67725×10^{-7}	(H_0) rejected - significant results
Test 3	9.074	5.26689×10^{-19}	(H_0) rejected - significant results
Test 4	4.666	7.46925×10^{-6}	(H_0) rejected - significant results
Test 5	13.51	9.27201×10^{-41}	(H_0) rejected - significant results
Test 6	2.195	0.035866519	(H_0) rejected - significant results
Test 7	3.787	0.000306705	(H_0) rejected - significant results
Test 8	6.984	1.0216×10^{-11}	(H_0) rejected - significant results
Test 9	8.576	4.26789×10^{-17}	(H_0) rejected - significant results
Test 10	8.5	8.16624×10^{-17}	(H_0) rejected - significant results
Test 11	18.055	6.52523×10^{-72}	(H_0) rejected - significant results
Test 12	2.647	0.012007269	(H_0) rejected - significant results

looking for violations, and clustering by working just with the possibly faulty version of the system and aiming to group the failing outputs into the smallest clusters to minimise the number of outputs that need to be checked), it can serve a valuable purpose as Daikon represents a viable alternative to automatically identifying failures. In both cases the effectiveness of the approach is defined using the F-measure (for details see section 3.3.4) which provides a useful point of comparison, so the findings for Daikon are compared with the best results from each of the clustering approaches for all systems in both experiments (results for Daikon are the same results that presented in the previous chapter section 3.6). The results of this are summarised in Tables 4.19, 4.20, 4.21 and 4.22 for NanoXML, Siena and Sed, and show the oracle type – Single, Average and Complete refer to the linkage metric used for agglomerative hierarchical clustering, and the version number – the 3 values correspond to cluster count (% test cases), cluster size (% test cases), and the F-measure in bold.

In the first experiment, table 4.19 shows that agglomerative hierarchical clustering algorithm with single linkage metric performed better than Daikon for NanoXML version 1 and 5. However, for version 2 and 3 Daikon performed better than agglomerative hierarchical clustering algorithm with all linkage metrics. The results for DBSCAN are very disappointed in comparison with Daikon where DBSCAN was beaten by Daikon in all NanoXML versions. On the other hand, EM clustering did outperform Daikon for NanoXML version 1 and 5 only. Table 4.20 shows that all clustering algorithms were not able to outperformed Daikon on both systems (Siena and Sed).

In the second experiment, the results for NanoXML (Table 4.21) shows that agglomerative hierarchical clustering with various linkage metrics outperformed Daikon (versions 1, 2 and 5). However, only for version 3 did Daikon perform better. Daikon was beaten by DBSCAN on version 2 and 5 but it performed much better than DBSCAN on version 1 and 3. EM, as we have already seen, fared badly except for version 1 where Daikon, curiously, also performed very poorly. For Siena (Table 4.22) the results show that agglomerative hierarchical clustering with complete metric is able to outperform Daikon. However, for this system Daikon performed much better than DBSCAN and EM. In contrast Daikon outperformed all clustering algorithms for Sed (Table 4.22).

Overall the clustering approaches (especially agglomerative hierarchical clustering algorithm with linkage metrics) using input/output pairs and execution traces together performed reasonably well compared to Daikon.

Table 4.19 F-measure for Failures Found for Clustering Algorithms Using Input/Output Pairs Versus Daikon in Nanoxml.

Oracles	NanoXML v1	NanoXML v2	NanoXML v3	NanoXML v5
Single	(15, 3, **0.62**)	(15, 3, **0.65**)	(20, 2.5, **0.75**)	(15, 3, **0.60**)
Average	(10, 6.25, **0.58**)	(10, 6.25, **0.63**)	(15, 3.25, **0.81**)	(10, 6.25, **0.59**)
Complete	(15, 3.12, **0.56**)	(25, 2.25, **0.56**)	(25, 2.25, **0.66**)	(15, 3.12, **0.44**)
DBSCAN	(19.9, 2.68, **0.29**)	(19.9, 2.68, **0.24**)	(19.9, 2.70, **0.26**)	(19.9, 2.60, **0.17**)
EM	(1.94, 25, **0.65**)	(2.42, 20.2, **0.29**)	(2.42, 20, **0.43**)	(1.45, 33, **0.77**)
Daikon	(-, -, **0.38**)	(-, -, **0.77**)	(-, -, **0.92**)	(-, -, **0.57**)

Table 4.20 F-measure for Failures Found for Clustering Algorithms Using Input/Output Pairs Versus Daikon in Siena Version 2 and Sed Version 5.

Oracles	Siena V2	Sed V5
Single	(20, 0.79, **0.65**)	(5, 6.4, **0.29**)
Average	(25, 0.6,**0.62**)	(25, 0.80,**0.25**)
Complete	(20, 0.79, **0.65**)	(10, 2.71, **0.38**)
DBSCAN	(4.45, 4.54, **0.03**)	(48.23, 1, **0.30**)
EM	(2.02, 16.66, **0.22**)	(2.71, 9.9, **0.06**)
Daikon	(-, -, **0.72**)	(-, -, **0.62**)

4.8 Discussion

Test oracles based on unsupervised learning techniques do not require the availability of labelled data or a fault free version of the system under test to construct test oracles which make them more scalable in comparison to test oracles based on supervised/semi-supervised learning techniques and test oracles based on invariant detection in terms of the provision of labelled data (other scalability issues may arise in the application of the algorithms but these a likely to be equally applicable to all approaches). In addition, the proposed approaches in this chapter are less expensive to obtain in comparison to test oracles based on supervised/semi-

Table 4.21 F-measure for Failures Found for Clustering Algorithms Using Input/Output Pairs Augmented with Execution Traces Versus Daikon in Nanoxml.

Oracles	NanoXML v1	NanoXML v2	NanoXML v3	NanoXML v5
Single	(15, 5.03, **0.73**)	(15, 5.03, **0.84**)	(15, 5.03, **0.84**)	(15, 5.03, **0.63**)
Average	(15, 4.30, **0.73**)	(15, 4.30, **0.81**)	(20, 3.27, **0.71**)	(15, 4.30, **0.61**)
Complete	(25, 2.73, **0.57**)	(25, 2.73, **0.69**)	(15, 4.27, **0.88**)	(10, 6.39, **0.83**)
DBSCAN	(50, 1.42, **0.31**)	(15, 5.1, **0.81**)	(15, 4.08, **0.81**)	(15, 3.78, **0.67**)
EM	(2.41, 20, **0.42**)	(1.93, 25.25, **0**)	(2.41, 19.8, **0.09**)	(1.44, 33.33, **0**)
Daikon	(-, -, **0.38**)	(-, -, **0.77**)	(-, -, **0.92**)	(-, -, **0.57**)

Table 4.22 F-measure for Failures Found for Clustering Algorithms Using Input/Output Pairs Augmented with Execution Traces Versus Daikon in Siena Version 2 and Sed Version 5.

Oracles	Siena V2	Sed V5
Single	(20, 0.79, **0.66**)	(20, 1.2, **0.28**)
Average	(10,1.98,**0.65**)	(25, 1, **0.26**)
Complete	(5, 4.04, **0.89**)	(25, 1, **0.29**)
DBSCAN	(6, 3.33, **0.48**)	(8, 3.5, **0.10**)
EM	(1.41, 14.28, **0**)	(1, 33.33, **0.08**)
Daikon	(-, -, **0.72**)	(-, -, **0.62**)

supervised learning techniques (again as no data labelling is necessary), but can be less accurate for the same reason.

4.9 Conclusion

This chapter has presented two empirical investigation for several clustering techniques such as agglomerative hierarchical with different linkage metrics, DBSCAN and EM clustering algorithms using dynamic execution data to build an automated test oracle. The input/output pairs only were used as input to the clustering algorithms in the first empirical investigation, and then they were augmented with execution traces in the second empirical investigation with the aim of improving the proportion of unique failures in the smaller clusters.

The experimental results gave an evidence which support the clustering hypothesis behind this work where in several cases small (less than average sized) clusters contained more than 60% of failures (and often a substantially higher proportion). As well as having a

higher failure density they also contained a spread of failures in the cases where there were multiple faults in the programs. The results provide us with some useful guidelines in terms of specifying the number of clusters as a parameter to the algorithms. Over both experiments Agglomerative Hierarchical Clustering produced the most consistently good results, although performance varied according to which linkage metric was used (and also varied with experiment). The results for DBCAN were also generally encouraging, particularly since the number of clusters does not need to be supplied as a parameters.

The results also demonstrate important practical consequences: the task of checking test outputs may potentially be reduced significantly to examining a relatively small proportion of the data to discover a large proportion of the failures. The approach has also been shown to be robust to a drop in the failure rate – all the way down to 1% of the output – and initial results suggest that when the failure rate is likely to be low then the number of clusters should also be small.

In addition, the clustering approach presented here performs favourably in comparison with Daikon in several cases especially in the second empirical investigation where input/output pairs augmented with execution traces. It must be stressed again that Daikon assumes that the system under test has fault-free version on which to train the oracle - something which is difficult to obtain in the reality. In contrast clustering approach makes no such assumption about a fault-free version (real practical scenario). The rational reason behind the poor performance of Daikon is that Daikon requires a large and complete test suite of the fault-free version of the system under test in order to train the oracle otherwise Daikon will be suffered of a high false positive rate [69] (all systems used in this work have a relatively small test suits). This gives an advantage for clustering approach in comparison to Daikon.

Chapter 5

Conclusions and Future Work

5.1 Summary of the Thesis

This research started by reviewing three extensive reviews of topics relating to test oracles [8, 9, 77]. The reviews showed that existing approaches to generate test oracles range from the inexpensive and ineffective to the effective but very costly [9]. The thesis was motivated by this finding and aimed to strike the balance between these approaches and develop a technique which can be effective and also inexpensive by using anomaly detection to automatically identify failing tests. As a result, the thesis begins by gathering background information on software testing and the test oracle problem in general. The thesis also identifies a variety of anomaly detection techniques that have been proposed to address the test oracle problem or to support other software engineering tasks such as reverse engineering and fault localisation.

It was found that a variety of machine learning and data mining techniques based on supervised learning strategy have been used to build an automated test oracles. It was also found that different types of features (dynamic execution data) were used to build the model. Another point to be noticed is that different approaches have been used to transform dynamic execution data to a suitable set of feature vectors to build the model. However, the evaluation on those techniques was relatively inadequate because of the subject programs' size (relatively small) which makes it difficult to generalise their applicability to build an

automated test oracles. In addition, the performance of supervised learning techniques depends on the availability of fully labelled training data set which is difficult to obtain on real software testing scenario. The most important finding was that machine learning and data mining techniques based on unsupervised learning and semi-supervised learning in particular have not been investigated intensively to build an automated test oracles.

Motivated by the finding of the review, the thesis continues by investigating semi-supervised learning techniques (self-training, co-training and co-EM) to support the construction of automated test oracles by classifying passing and failing tests. It presented two different studies with two practical scenarios associated with semi-supervised learning techniques - labelling both normal (passing) and abnormal (failing) tests, and labelling normal (passing) tests alone. The first study used the input/output pairs of the systems under test (NanoXML, Siena and Sed) as input to the learning algorithms, and then augmented these with execution traces in the second study. A comparison study between existing techniques from the specification mining domain (the data invariant detector Daikon [30]) and semi-supervised learning techniques was performed. The main findings from from this investigation may be summarised as follows:

- From the algorithms investigated Naïve Bayes with EM and co-training using Naïve Bayes proved to be the most consistent performers. These approaches have been shown to perform well in the area of document classification where all of the data sets are textual (as in this investigation) which may go some way towards explaining the reason for their performance in relation to the much poorer one of Support Vector Machines with the co-EM and co-training methods.

- Considering just input-output pairs with positive labels (passing cases) alone yielded very poor results.

- The results for input-output pairs with positive and negative labels (a subset of passing cases and a small number of failing ones) were variable: encouraging for NanoXML, acceptable for Sed but disappointing for Siena. This result was a surprise given that

the profile of the system is far less fragmented than Sed, but could be attributable to a data imbalance problem.

- Adding in execution trace data can help enormously. This is not really unexpected given that more information is being supplied to the algorithm, but even when very fragmented they improve the accuracy once a fair proportion of the data is labelled. In extreme cases the impact is dramatic (leading to a perfect classifier on from a very small subset of labelled data) but these were assumed to be relatively rare instances.

- Using input-output pairs and execution traces with just positive labels worked well when the number of traces is small but otherwise did not perform as well as when a small number of failing inputs are provided.

- Naïve Bayes with EM and co-training using Naive Bayes perform far better than Daikon as an automated test oracle even with small test suites (Daikon usually requires a fault free version of the system with a large and complete test suite to perform well).

The findings from this investigation have important implications for the practical use of this technique: when checking the test results from a system a developer need only examine a relatively small proportion of these and use this information to train a learning algorithm to classify the remainder. This has the potential to improve the efficiency and reduce the cost and tedium of manually checking large volumes of test results.

The thesis continues by adding an another investigation into unsupervised learning techniques (mainly Agglomerative Hierarchical, DBSCAN and EM clustering algorithms) to build an automated test oracle. Again it presented two different studies along with a comparison study to Daikon. The first study used input/output pairs of the systems under test (NanoXML, Siena and Sed) as input to the clustering algorithms, and then augmented these with execution traces in the second study. The main findings from from this investigation may be summarised as follows:

- Small clusters (less than average sized) contained more than 60% of failures (and often a substantially higher proportion). As well as having a higher failure density they

also contained a spread of failures in the cases where there were multiple faults in the programs.

- Over both studies Agglomerative Hierarchical Clustering produced the most consistently good results, although performance varied according to which linkage metric was used (and also varied with experiment). The results for DBSCAN were also generally encouraging, particularly since the number of clusters does not need to be supplied as a parameters.

- The approach has also been shown to be robust to a drop in the failure rate – all the way down to 1% of the output – and the results suggested that when the failure rate is likely to be low then the number of clusters should also be small.

- The clustering approaches used here performed favourably in comparison to Daikon in several cases especially in the second study where input/output pairs are augmented with execution traces.

The results also demonstrated important practical consequences: the task of checking test outputs may potentially be reduced significantly to examining an approximately just 25% of the data – a substantial reduction in effort and time.

5.2 Contributions of the Thesis

The thesis makes several contributions to the software testing area in general and to the test oracle problem in particular. This study has investigated the use of semi-supervised learning techniques to classify passing and failing test results which can be recognised as a first attempt to use such approaches to support the construction of test oracles. The experimental results showed that anomaly detection techniques based on semi-supervised learning have the potential to support the construction of automated test oracles. The results also showed an important implications for the practical potential of semi-supervised learning techniques: when checking the test results from a system a developer may need only examine a small

proportion of these and use this information to train a learning algorithm to automatically classify the remainder.

The study has also investigated the use of unsupervised learning techniques to separate passing and failing test executions by clustering anomalies. The experimental results gave an evidence which support the clustering hypothesis (*"Normal data instances belong to large and dense clusters, while anomalies either belong to small or spare cluster"*) behind this work [18]. The experimental results showed that anomaly detection techniques based on clustering techniques can be used to build an automated test oracles. The results of this investigation also have potentially important practical consequences: the task of scrutinising test outputs may potentially be reduced significantly to examining well under half the contents of the smaler clusters an order of magnitude reduction in effort and time.

The study has performed a comparison between existing techniques from specification mining domain (the data invariant detector Daikon [30]) and anomaly detection techniques based on semi-supervised and unsupervised learning strategies. In several cases the proposed approaches performed well in comparison to Daikon. It must be stressed that Daikon requires a fault free version of the system under test with complete and large test suite from which to build assertions - a luxury that the proposed approaches did not require.

5.3 Threats to Validity

The clear issue concerning the external validity of this work is the generalisability of the experimental results: the finding so far are limited to three subject programs which can not be said to from a representative set, even though they are non-trivial real-world Java and C systems of reasonable size containing real faults. The failure rates for all systems may also not be representative, as may be the test cases (although these were created independently via collaboration between the SIR and subject systems' developers). Note that, some investigation of the impact of reducing the failure rate has been undertaken mainly on unsupervised learning techniques investigation.

A potential construct validity for this work lies on the use of the coding scheme for both input/output pairs and execution traces (although the coding scheme for text/string data was suggested to be the most suitable by [103] which are similar to data type in this thesis). The coding scheme for execution traces was the same algorithm used by Nguyen et al. [69] and also has no information about whether a trace is associated with a passing or failing execution.

The author of this thesis suggests further replication of the study on additional subject programs to be able to generalise the findings and to reduce the external validity threat.

5.4 Future Work

In this thesis, author has proposed new approaches to construct test oracles from software data to guide automatic software testing in the absence of specifications. Despite of the initial achievements, there are number of barriers for the work to become practically usable which fall into the categories of scalability and accuracy. Both barriers is explained further below:

Improving the accuracy of the proposed approaches:

Accuracy in this context means the ability of the proposed approaches to identify failing and passing test results as correctly as possible (high true positive rate and low false positive/-false negative rates). The accuracy of semi-supervised and unsupervised learning techniques can be improved by augmenting the data sets (input/output pairs and execution traces) with more relevant information from the program execution (e.g. state information, execution time and code coverage etc.) to build more effective/accurate test oracles.

Adding more execution data to the data sets can help to reduce the size of labelled data used to train the learning algorithms in semi-supervised learning. This also relates to scalability but to make the approach practical the size of labelled data needs to be as low as possible which means improving the accuracy as well. The other point related to the size of labelled and then accuracy is that the predictions of semi-supervised learning approaches

should come with an estimate of confidence, possibly associated with the proportion of labelled data used.

The accuracy of unsupervised learning approaches can be improved by selecting the most appropriate similarity measures for clustering algorithms. In addition, the number of specified clusters for clustering algorithms is important and the accuracy can be improved by specifying the optimal number of cluster counts. Note that, the accuracy of unsupervised learning techniques (clustering algorithms) in this research means the separation between failing and passing test results. In other words, the failing test results should be grouped in small clusters with high failure density compared to large clusters which should have more passing test results. The definition of 'small' and 'large' is quite coarse in this context. Finding the appropriate definition of small and large clusters can help to improve accuracy in practice, as well as providing some guidelines to the tester on the point where is not worth exploring further clusters.

Increasing the scalability of the proposed approaches:

The test data transformation for the software under test is a very important issue which affects the scalability of the proposed approaches in this thesis. This clearly impacts on the volume of data that has to be processed but also has implications for accuracy too, so must be done in a way that does not compromise this. To be practically applicable it is necessary to find a generic automated approach for each type of test data. For instance, the input/output pairs for tested systems in this thesis were string/text type and it turned out that tokenization procedure worked reasonably well with the proposed approach but this may not be generally applicable for all input and output types.

An interesting approach to addressing the scalability (and cost-effectiveness) issue which will be explored in the future is to investigate the feasibility of using the cheap results from clustering in which there is the greatest confidence to generate the labelled set required by the semi-supervised techniques, and thereby reduce the cost of the semi-supervised learning.

Generally, further research will be devoted to overcome those barriers by conducting further empirical investigation of the effectiveness of proposed approaches corroborate the

findings and to increase their external validity, particularly by exploring a wider range of programs, faults and coding schemes for dynamic execution data (input/output pairs and execution traces etc.).

5.5 Closing remarks

As this thesis has shown that building an effective test oracle with low cost is a challenging as well as complex problem. Through the studies presented here, author has aimed to build an automated test oracle via anomaly detection techniques in order to improve the efficiency and reduce the cost and tedium of manually checking large volumes of test results. The feasibility of the approach is demonstrated and shown to have the potential to reduce by an order of magnitude the numbers of outputs that need to be examined following a test run.

References

[1] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 352–361, March 2013. doi: 10.1109/ICST.2013.11.

[2] K. K. Aggarwal, Yogesh Singh, A. Kaur, and O. P. Sangwan. A neural net based approach to test oracle. *SIGSOFT Softw. Eng. Notes*, 29(3):1–6, May 2004. ISSN 0163-5948. doi: 10.1145/986710.986725. URL http://doi.acm.org/10.1145/986710.986725.

[3] Bernhard K. Aichernig, Andreas Griesmayer, Einar Broch Johnsen, Rudolf Schlatte, and Andries Stam. Formal methods for components and objects. chapter Conformance Testing of Distributed Concurrent Systems with Executable Designs, pages 61–81. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-04166-2. doi: 10.1007/978-3-642-04167-9_4. URL http://dx.doi.org/10.1007/978-3-642-04167-9_4.

[4] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008. ISBN 0521880386, 9780521880381.

[5] J. H. Andrews and Yingjun Zhang. General test result checking with log file analysis. *IEEE Transactions on Software Engineering*, 29(7):634–648, July 2003. ISSN 0098-5589. doi: 10.1109/TSE.2003.1214327.

[6] A. Avizienis and J. C. Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986. ISSN 0018-9219. doi: 10.1109/PROC.1986.13527.

[7] George K. Baah, Alexander Gray, and Mary Jean Harrold. On-line anomaly detection of deployed software: A statistical machine learning approach. In *Proceedings of the 3rd International Workshop on Software Quality Assurance*, SOQUA '06, pages 70–77, New York, NY, USA, 2006. ACM. ISBN 1-59593-584-3. doi: 10.1145/1188895.1188911. URL http://doi.acm.org/10.1145/1188895.1188911.

[8] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. URL http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

[9] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *Software Engineering, IEEE Transactions on*, 41(5): 507–525, May 2015. ISSN 0098-5589. doi: 10.1109/TSE.2014.2372785.

[10] Antonia Bertolino. Software testing research and practice. In *Proceedings of the Abstract State Machines 10th International Conference on Advances in Theory and Practice*, ASM'03, pages 1–21, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00624-9. URL http://dl.acm.org/citation.cfm?id=1754749.1754751.

[11] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.25. URL http://dx.doi.org/10.1109/FOSE.2007.25.

[12] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, Arvind Krishnamurthy, and Thomas E. Anderson. Mining temporal invariants from partially ordered logs. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, SLAML '11, pages 3:1–3:10, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0978-3. doi: 10.1145/2038633.2038636. URL http://doi.acm.org/10.1145/2038633.2038636.

[13] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained

models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 267–277, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025151. URL http://doi.acm.org/10.1145/2025113.2025151.

[14] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

[15] Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, COLT' 98, pages 92–100, New York, NY, USA, 1998. ACM. ISBN 1-58113-057-0. doi: 10.1145/279943.279962. URL http://doi.acm.org/10.1145/279943.279962.

[16] James F. Bowring, James M. Rehg, and Mary Jean Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 195–205, New York, NY, USA, 2004. ACM. ISBN 1-58113-820-2. doi: 10.1145/1007512.1007539. URL http://doi.acm.org/10.1145/1007512.1007539.

[17] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 480–490, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0. URL http://dl.acm.org/citation.cfm?id=998675.999452.

[18] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009. ISSN 0360-0300. doi: 10.1145/1541880.1541882. URL http://doi.acm.org/10.1145/1541880.1541882.

[19] Jessica Chen and Suganthan Subramaniam. Specification-based testing for gui-based applications. *Software Quality Journal*, 10(3):205–224, 2002. ISSN 1573-1367. doi: 10.1023/A:1021634422504. URL http://dx.doi.org/10.1023/A:1021634422504.

[20] T.Y. Chen, T.H. Tse, and Z. Quan Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45(1):1 – 9, 2003. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/S0950-5849(02)00129-5. URL http://www.sciencedirect.com/science/article/pii/S0950584902001295.

[21] D. Dasgupta and N.S. Majumdar. Anomaly detection in multidimensional data using negative selection algorithm. In *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, volume 2, pages 1039–1044, 2002. doi: 10.1109/CEC.2002.1004386.

[22] D. Dasgupta and F. Nino. A comparison of negative and positive selection algorithms in novel pattern detection. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 1, pages 125–130 vol.1, 2000. doi: 10.1109/ICSMC.2000.884976.

[23] Marcio Eduardo Delamaro, Fátima de Lourdes dos Santos Nunes, and Rafael Alves Paes de Oliveira. Using concepts of content-based image retrieval to implement graphical testing oracles. *Software Testing, Verification and Reliability*, 23(3):171–198, 2013. ISSN 1099-1689. doi: 10.1002/stvr.463. URL http://dx.doi.org/10.1002/stvr.463.

[24] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, pages 339–348, May 2001. doi: 10.1109/ICSE.2001.919107.

[25] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: The distribution of program failures in a profile space. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 246–255, New York, NY, USA, 2001. ACM. ISBN 1-58113-390-1. doi: 10.1145/503209.503243. URL http://doi.acm.org/10.1145/503209.503243.

[26] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005. ISSN 1382-3256. doi: 10.1007/s10664-005-3861-2. URL http://dx.doi.org/10.1007/s10664-005-3861-2.

[27] Roong-Ko Doong and Phyllis G. Frankl. The astoot approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, April 1994. ISSN 1049-331X. doi: 10.1145/192218.192221. URL http://doi.acm.org/10.1145/192218.192221.

[28] D. D'Souza and M. Gopinatha. Computing complete test graphs for hierarchical systems. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 70–79, Sept 2006. doi: 10.1109/SEFM.2006.13.

[29] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 213–224, May 1999. doi: 10.1145/302405.302467.

[30] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2007.01.015. URL http://dx.doi.org/10.1016/j.scico.2007.01.015.

[31] Patrick Francis, David Leon, Melinda Minch, and Andy Podgurski. Tree-based methods for classifying software failures. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, pages 451–462, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2215-7. doi: 10.1109/ISSRE.2004.43. URL http://dx.doi.org/10.1109/ISSRE.2004.43.

[32] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE

'11, pages 416–419, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025179. URL http://doi.acm.org/10.1145/2025113.2025179.

[33] Kambiz Frounchi, Lionel C. Briand, Leo Grady, Yvan Labiche, and Rajesh Subramanyan. Automating image segmentation verification and validation by learning test oracles. *Information and Software Technology*, 53(12):1337 – 1348, 2011. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2011.06.009. URL http://www.sciencedirect.com/science/article/pii/S095058491100156X.

[34] Ryohei Fujimaki. An approach to spacecraft anomaly detection problem using kernel feature space. In *in Proc. PAKDD-2005: Ninth Pacific-Asia Conference on Knowledge Discovery and Data Mining*. ACM Press, 2005.

[35] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806806. URL http://doi.acm.org/10.1145/1806799.1806806.

[36] Marie-Claude Gaudel. Testing can be formal, too. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '95, pages 82–96, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-59293-8. URL http://dl.acm.org/citation.cfm?id=646619.697560.

[37] Dimitra Giannakopoulou, David H. Bushnell, Johann Schumann, Heinz Erzberger, and Karen Heere. Formal testing for separation assurance. *Annals of Mathematics and Artificial Intelligence*, 63(1):5–30, 2011. ISSN 1573-7470. doi: 10.1007/s10472-011-9224-3. URL http://dx.doi.org/10.1007/s10472-011-9224-3.

[38] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25 (5):30–37, Sept 2008. ISSN 0740-7459. doi: 10.1109/MS.2008.109.

[39] R. Guderlei, R. Just, and C. Schneckenburger. Benchmarking testing strategies with tools from mutation analysis. In *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, pages 360–364, April 2008. doi: 10.1109/ICSTW.2008.11.

[40] Yuanyuan Guo, Xiaoda Niu, and H. Zhang. An extensive empirical study on semi-supervised learning. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 186–195, Dec 2010. doi: 10.1109/ICDM.2010.66.

[41] Jiawei Han, Micheline Kamber, and Jian Pei. 10 - cluster analysis: Basic concepts and methods. In Jiawei HanMicheline Kamber and Jian Pei, editors, *Data Mining (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 443 – 495. Morgan Kaufmann, Boston, third edition edition, 2012. ISBN 978-0-12-381479-1. doi: http://dx.doi.org/10.1016/B978-0-12-381479-1.00010-1. URL http://www.sciencedirect.com/science/article/pii/B9780123814791000101.

[42] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. doi: 10.1145/581339.581377. URL http://doi.acm.org/10.1145/581339.581377.

[43] M. Haran, A. Karr, M. Last, A. Orso, A. A. Porter, A. Sanil, and S. Fouche. Techniques for classifying executions of deployed software to support software engineering tasks. *IEEE Transactions on Software Engineering*, 33(5):287–304, May 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.1004.

[44] Murali Haran, Alan Karr, Alessandro Orso, Adam Porter, and Ashish Sanil. Applying classification techniques to remotely-collected program execution data. *SIGSOFT Softw. Eng. Notes*, 30(5):146–155, September 2005. ISSN 0163-5948. doi: 10.1145/1095430.1081732. URL http://doi.acm.org/10.1145/1095430.1081732.

[45] D. M. Hoffman and P. Strooper. Automated module testing in prolog. *IEEE Transactions on Software Engineering*, 17(9):934–943, Sep 1991. ISSN 0098-5589. doi: 10.1109/32.92913.

[46] D. Hook and D. Kelly. Testing for trustworthiness in scientific software. In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 59–64, May 2009. doi: 10.1109/SECSE.2009.5069163.

[47] C. Hunter and P. Strooper. Systematically deriving partial oracles for testing concurrent programs. In *Computer Science Conference, 2001. ACSC 2001. Proceedings. 24th Australasian*, pages 83–91, 2001. doi: 10.1109/ACSC.2001.906627.

[48] T. Janssen, R. Abreu, and A. J. C. v. Gemund. Zoltar: A toolset for automatic fault localization. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 662–664, Nov 2009. doi: 10.1109/ASE.2009.27.

[49] A. Z. Javed, P. A. Strooper, and G. N. Watson. Automated generation of test cases using model-driven architecture. In *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07, pages 3–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2971-2. doi: 10.1109/AST.2007.2. URL http://dx.doi.org/10.1109/AST.2007.2.

[50] H. Jin, Y. Wang, N. W. Chen, Z. J. Gou, and S. Wang. Artificial neural network for automatic test oracles generation. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 2, pages 727–730, Dec 2008. doi: 10.1109/CSSE.2008.774.

[51] Yogesh Singh K. K. Aggarwal and Arvinder Kaur. A neural net based approach to test oracle. *Journal of Computer Science*, 1:341–345, 2005. ISSN 3.

[52] Dae S. Kim-Park, Claudio de la Riva, and Javier Tuya. An automated test oracle for xml processing programs. In *Proceedings of the First International Workshop on Software Test Output Validation*, STOV '10, pages 5–12, New York, NY, USA,

2010. ACM. ISBN 978-1-4503-0138-1. doi: 10.1145/1868048.1868050. URL http://doi.acm.org/10.1145/1868048.1868050.

[53] Daniel T. Larose. *Discovering Knowledge in Data*. John Wiley Sons, Inc., 2005. ISBN 9780471687542.

[54] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 557–566, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9. doi: 10.1145/1557019.1557083. URL http://doi.acm.org/10.1145/1557019.1557083.

[55] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 501–510, May 2008. doi: 10.1145/1368088.1368157.

[56] Ying Lu and Mao Ye. Oracle model based on rbf neural networks for automated software testing. In *Information Technology Journal*, volume 6, pages 469–474, 2007.

[57] Chengying Mao and Yansheng Lu. *Extracting the Representative Failure Executions via Clustering Analysis Based on Markov Profile Model*, pages 217–224. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31877-4. doi: 10.1007/11527503_26. URL http://dx.doi.org/10.1007/11527503_26.

[58] Ye Mao, Feng Boqin, Zhu Li, and Lin Yao. *Neural Information Processing: 13th International Conference, ICONIP 2006, Hong Kong, China, October 3-6, 2006. Proceedings, Part III*, chapter Neural Networks Based Automated Test Oracle for Software Testing, pages 498–507. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-46485-3. doi: 10.1007/11893295_55. URL http://dx.doi.org/10.1007/11893295_55.

[59] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 117–126, Nov 2008. doi: 10.1109/ISSRE.2008.48.

[60] W. Masri, A. Podgurski, and D. Leon. An empirical study of test case filtering techniques based on exercising information flows. *IEEE Transactions on Software Engineering*, 33(7):454–477, July 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.1020.

[61] P. McMinn, M. Shahbaz, and M. Stevenson. Search-based test input generation for string data types using the results of web queries. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 141–150, April 2012. doi: 10.1109/ICST.2012.94.

[62] Phil McMinn, Mark Stevenson, and Mark Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the First International Workshop on Software Test Output Validation*, STOV '10, pages 1–4, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0138-1. doi: 10.1145/1868048.1868049. URL http://doi.acm.org/10.1145/1868048.1868049.

[63] A. M. Memon and Qing Xie. Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for gui-based software. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 8–17, Sept 2004. doi: 10.1109/ICSM.2004.1357785.

[64] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL http://doi.acm.org/10.1145/96267.96279.

[65] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

[66] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011. ISBN 1118031962, 9781118031964.

[67] Paulo Augusto Nardi. *On test oracles for Simulink-like models*. Universidade de São Paulo, 2013.

[68] Andrew Ng. *Lectures on Machine Learning Course by Stanford University*, 2013. https://www.coursera.org/learn/machine-learning [Accessed:03/07/2013].

[69] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Automated oracles: An empirical study on cost and effectiveness. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 136–146, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491434. URL http://doi.acm.org/10.1145/2491411.2491434.

[70] Kamal Nigam and Rayid Ghani. Analyzing the effectiveness and applicability of co-training. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM '00, pages 86–93, New York, NY, USA, 2000. ACM. ISBN 1-58113-320-0. doi: 10.1145/354756.354805. URL http://doi.acm.org/10.1145/354756.354805.

[71] Kamal Nigam, Andrew Kachites McCallum, Sebastian Thrun, and Tom Mitchell. Text classification from labeled and unlabeled documents using em. *Mach. Learn.*, 39 (2-3):103–134, May 2000. ISSN 0885-6125. doi: 10.1023/A:1007692713085. URL http://dx.doi.org/10.1023/A:1007692713085.

[72] Rafael A. P. Oliveira, Upulee Kanewala, and Paulo A. Nardi. Automated Test Oracles: State of the Art, Taxonomies, and Trends. volume 95, pages 113–199. ELSEVIER ACADEMIC PRESS INC 525 B STREET, SUITE 1900, SAN DIEGO, CA 92101-4495 USA, 2014.

[73] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 504–527, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-27992-X, 978-3-540-27992-1. doi: 10.1007/11531142_22. URL http://dx.doi.org/10.1007/11531142_22.

[74] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA ’07, pages 815–816, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7. doi: 10.1145/1297846.1297902. URL http://doi.acm.org/10.1145/1297846.1297902.

[75] S. Parsa and S. A. Naree. Software online bug detection: applying a new kernel method. *IET Software*, 6(1):61–73, February 2012. ISSN 1751-8806. doi: 10.1049/iet-sen.2010.0057.

[76] Saeed Parsa, Somaye Arabi Nare, and Mojtaba Vahidi-Asl. *Early Bug Detection in Deployed Software Using Support Vector Machine*, pages 518–525. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-540-89985-3. doi: 10.1007/978-3-540-89985-3_64. URL http://dx.doi.org/10.1007/978-3-540-89985-3_64.

[77] Mauro Pezzè and Cheng Zhang. Automated test oracles: A survey. In *Advances in Computers*, volume 95, pages 1–48. Elsevier, 2015.

[78] M. Pezzè. Towards cost-effective oracles. In *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*, pages 1–2, May 2015. doi: 10.1109/AST.2015.7.

[79] Andy Podgurski and Charles Yang. Partition testing, stratified sampling, and cluster analysis. In *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT ’93, pages 169–181, New York, NY, USA, 1993. ACM. ISBN 0-89791-625-5. doi: 10.1145/256428.167076. URL http://doi.acm.org/10.1145/256428.167076.

[80] Andy Podgurski, Wassim Masri, Yolanda McCleese, Francis G. Wolff, and Charles Yang. Estimation of software reliability by stratified sampling. *ACM Trans. Softw. Eng. Methodol.*, 8(3):263–283, July 1999. ISSN 1049-331X. doi: 10.1145/310663.310667. URL http://doi.acm.org/10.1145/310663.310667.

[81] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 465–475, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X. URL http://dl.acm.org/citation.cfm?id=776816.776872.

[82] Almaghairbe Rafig and Marc Roper. Building test oracles by clustering failures. In *Proceedings of the 10th International Workshop on Automation of Software Test*, AST 2015. IEEE, 2015.

[83] Anthony Ralston, Edwin D. Reilly, and David Hemmendinger, editors. *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., Chichester, UK, 4th edition, 2003. ISBN 0-470-86412-5.

[84] A. Rauf, S. Anwar, M. Ramzan, S. ur Rehman, and A. Ali Shahid. Ontology driven semantic annotation based gui testing. In *Emerging Technologies (ICET), 2010 6th International Conference on*, pages 261–264, Oct 2010. doi: 10.1109/ICET.2010.5638479.

[85] Om Prakash Sangwan, Pradeep Kumar Bhatia, and Yogesh Singh. Radial basis function neural network based approach to test oracle. *SIGSOFT Softw. Eng. Notes*, 36(5):1–5, September 2011. ISSN 0163-5948. doi: 10.1145/2020976.2020992. URL http://doi.acm.org/10.1145/2020976.2020992.

[86] Sigurd Schneider, Ivan Beschastnikh, Slava Chernyak, Michael D. Ernst, and Yuriy Brun. Synoptic: Summarizing system logs with refinement. In *Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML '10)*, Vancouver, BC, Canada, October 3, 2010.

[87] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Security and Privacy, 2001. S P 2001. Proceedings. 2001 IEEE Symposium on*, pages 144–155, 2001. doi: 10.1109/SECPRI.2001.924295.

[88] K. I. Seo and E. M. Choi. Rigorous vertical software system testing in ide. In *5th ACIS International Conference on Software Engineering Research, Management Applications (SERA 2007)*, pages 847–854, Aug 2007. doi: 10.1109/SERA.2007.114.

[89] S. R. Shahamiri, W. M. N. W. Kadir, and S. Z. Mohd-Hashim. A comparative study on automated software test oracle methods. In *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, pages 140–145, Sept 2009. doi: 10.1109/ICSEA.2009.29.

[90] S. R. Shahamiri, W. M. N. Wan Kadir, and S. Ibrahim. A single-network ann-based oracle to verify logical software modules. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, volume 2, pages V2–272–V2–276, Oct 2010. doi: 10.1109/ICSTE.2010.5608808.

[91] S. R. Shahamiri, W. M. N. Wan Kadir, and S. Ibrahim. A single-network ann-based oracle to verify logical software modules. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, volume 2, pages V2–272–V2–276, Oct 2010. doi: 10.1109/ICSTE.2010.5608808.

[92] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, and Suhaimi bin Ibrahim. An automated oracle approach to test decision-making structures. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 5, pages 30–34, July 2010. doi: 10.1109/ICCSIT.2010.5563989.

[93] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, and Suhaimi bin Ibrahim. An automated oracle approach to test decision-making structures. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 5, pages 30–34, July 2010. doi: 10.1109/ICCSIT.2010.5563989.

[94] SeyedReza Shahamiri, WanM.N. Wan-Kadir, Suhaimi Ibrahim, and SitiZaitonMohd Hashim. Artificial neural networks as multi-networks automated test oracle. *Automated Software Engineering*, 19(3):303–334, 2012. ISSN 0928-8910. doi: 10.1007/s10515-011-0094-z. URL http://dx.doi.org/10.1007/s10515-011-0094-z.

[95] H. M. Sneed. State coverage of embedded realtime programs. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 245–, Jul 1988. doi: 10.1109/WST.1988.5384.

[96] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl. Automated oracle creation support, or: How i learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 870–880, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337326.

[97] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, January 1986. ISSN 0098-5589. doi: 10.1109/TSE.1986.6312929. URL http://dx.doi.org/10.1109/TSE.1986.6312929.

[98] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008. ISBN 1596932147, 9781596932142.

[99] G. Tassey. *The Economic Impact of Inadequate Infrastructure for Software Testing*. Number Planning Report 02-3. National Institute Of Standards & Technology, May 2002. URL http://www.nist.gov/director/planning/upload/report02-3.pdf.

[100] D. Tu, R. Chen, Z. Du, and Y. Liu. A method of log file analysis for test oracle. In *Scalable Computing and Communications; Eighth International Conference on Embedded Computing, 2009. SCALCOM-EMBEDDEDCOM'09. International Conference on*, pages 351–354, Sept 2009. doi: 10.1109/EmbeddedCom-ScalCom.2009.69.

[101] Meenakshi Vanmali, Mark Last, and Abraham Kandel. Using a neural network in the software testing process. *International Journal of Intelligent Systems*, 17(1):45–62, 2002. ISSN 1098-111X. doi: 10.1002/int.1002. URL http://dx.doi.org/10.1002/int.1002.

[102] Elaine J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4): 465–470, 1982. doi: 10.1093/comjnl/25.4.465. URL http://comjnl.oxfordjournals.org/content/25/4/465.abstract.

[103] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. ISBN 0120884070.

[104] Shali Yan, Zhenyu Chen, Zhihong Zhao, Chen Zhang, and Yuming Zhou. A dynamic test cluster sampling strategy by leveraging execution spectra information. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 147–154, April 2010. doi: 10.1109/ICST.2010.47.

[105] Shali Yan, Zhenyu Chen, Zhihong Zhao, Chen Zhang, and Yuming Zhou. A dynamic test cluster sampling strategy by leveraging execution spectra information. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 147–154, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-3990-4. doi: 10.1109/ICST.2010.47. URL http://dx.doi.org/10.1109/ICST.2010.47.

[106] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134325. URL http://doi.acm.org/10.1145/1134285.1134325.

[107] M. Ye, B. Feng, L. Zhu, and Y. Lin. Automated test oracle based on neural networks. In *2006 5th IEEE International Conference on Cognitive Informatics*, volume 1, pages 517–522, July 2006. doi: 10.1109/COGINF.2006.365539.

[108] M. Ye, B. Feng, L. Zhu, and Y. Lin. Automated test oracle based on neural networks. In *2006 5th IEEE International Conference on Cognitive Informatics*, volume 1, pages 517–522, July 2006. doi: 10.1109/COGINF.2006.365539.

[109] Cemal Yilmaz and Adam Porter. Combining hardware and software instrumentation to classify program executions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 67–76, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882304. URL http://doi.acm.org/10.1145/1882291.1882304.

[110] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 201–211. ACM Press, July 2009.

[111] Wujie Zheng, Hao Ma, Michael R. Lyu, Tao Xie, and Irwin King. Mining test oracles of web search engines. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 0:408–411, 2011. doi: http://doi.ieeecomputersociety.org/10.1109/ASE.2011.6100085.

[112] Zhi Quan Zhou, ShuJia Zhang, Markus Hagenbuchner, T. H. Tse, Fei-Ching Kuo, and T. Y. Chen. Automated functional testing of online search services. *Softw. Test. Verif. Reliab.*, 22(4):221–243, June 2012. ISSN 0960-0833. doi: 10.1002/stvr.437. URL http://dx.doi.org/10.1002/stvr.437.

Appendix A

Fault Details for All Systems

```
// code with the embedded fault
...
} while ((ch == ' ') || (ch == '\t') || (ch == '\r'));

//original code
,,,
} while ((ch == ' ') || (ch == '\t') || (ch == '\n')
|| (ch == '\r'));
```

Listing A.1 NanoXML V1 - F1 - NonValidator class

```
// code with the embedded fault
...
if (! XMLUtil.checkLiteral(reader, '\%',
        this.parameterEntityResolver, "TLIST")) {

//original code
,,,
if (! XMLUtil.checkLiteral(reader, '\%',
        this.parameterEntityResolver, "TTLIST")) {
```

Listing A.2 NanoXML V1 - F2 - NonValidator class

```
// code with the embedded fault
...
 if (! XMLUtil.checkLiteral(reader, '\%',
                this.parameterEntityResolver, "SYSTEM")) {

//original code
,,,
if (! XMLUtil.checkLiteral(reader, '\%',
```

```
9                this.parameterEntityResolver, "YSTEM")) {
```

Listing A.3 NanoXML V1 - F3 - NonValidator class

```
1 // code with the embedded fault
2 ...
3 if (_enum.hasMoreElements()) {
4 //original code
5 ,,,
6 while (_enum.hasMoreElements()) {
```

Listing A.4 NanoXML V1 - F4 - NonValidator class

```
1 // code with the embedded fault
2 ...
3 Removed Line
4 //original code
5 ,,,
6 XMLUtil.skipTag(this.reader, '&', this.entityResolver);
```

Listing A.5 NanoXML V1 - F5 - StdXMLParser class

```
1 // code with the embedded fault
2 ...
3 name = name.substring(colonIndex);
4 //original code
5 ,,,
6 name = name.substring(colonIndex + 1);
```

Listing A.6 NanoXML V1 - F6 - StdXMLParser class

```
1 // code with the embedded fault
2 ...
3 true,
4 //original code
5 ,,,
6 false,
```

Listing A.7 NanoXML V1 - F7 - StdXMLParser class

```
1 // code with the embedded fault
2 ...
3 String value = XMLUtil.scanString(reader, '\%', true,
4                           this.parameterEntityResolver);
5 //original code
6 ,,,
```

```
String value = XMLUtil.scanString(reader, '\%', false,
                    this.parameterEntityResolver);
```

Listing A.8 NanoXML V2 - F1 - NonValidator class

```
// code with the embedded fault
...
this.currentSystemID = new URL("file:");
//original code
,,,
this.currentSystemID = new URL("file:.");
```

Listing A.9 NanoXML V2 - F2 - StdXMLReader class

```
// code with the embedded fault
...
Removed Line
//original code
,,,
this.currentSystemID = new URL(this.currentSystemID, systemID);
```

Listing A.10 NanoXML V2 - F3 - StdXMLReader class

```
// code with the embedded fault
...
return null;
//original code
,,,
return this.systemID;
```

Listing A.11 NanoXML V2 - F4 - XMLElement class

```
// code with the embedded fault
...
return this.openExternalEntity(xmlReader, id[1], id[0]);
//original code
,,,
return this.openExternalEntity(xmlReader, id[0], id[1]);
```

Listing A.12 NanoXML V2 - F5 - XMLEntityResolver class

```
// code with the embedded fault
...
? (((systemID != null) ? ""
//original code
,,,
```

```
6 ? (((systemID == null) ? ""
```

Listing A.13 NanoXML V2 - F6 - XMLException class

```
1 // code with the embedded fault
2 ...
3 this.encapsulatedException = null;
4 //original code
5 ,,.
6 this.encapsulatedException = e;
```

Listing A.14 NanoXML V2 - F7 - XMLException class

```
1 // code with the embedded fault
2 ...
3 Removed Line
4 //original code
5 ,,.
6 if (nsPrefix != null) {
7         fullName = nsPrefix + ':' + name;
8         }
```

Listing A.15 NanoXML V3 - F1 - StdXMLBuilder class

```
1 // code with the embedded fault
2 ...
3 fullName = nsPrefix + key;
4 //original code
5 ,,.
6 fullName = nsPrefix + ':' + key;
```

Listing A.16 NanoXML V3 - F2 - StdXMLBuilder class

```
1 // code with the embedded fault
2 ...
3 Removed Line
4 //original code
5 ,,.
6 if (key.startsWith("xmlns")) {
7                continue;
8              }
```

Listing A.17 NanoXML V3 - F3 - StdXMLParser class

```
1 // code with the embedded fault
2 ...
```

```
Removed Line
//original code
,,.
this.systemID = systemID;
```

Listing A.18 NanoXML V3 - F4 - XMLElement class

```
// code with the embedded fault
...
if ((childName != null) && childName.equals(fullName)) {
//original code
,,.
if ((childName != null) && childName.equals(name)) {
```

Listing A.19 NanoXML V3 - F5 - XMLElement class

```
// code with the embedded fault
...
Removed Line
//original code
,,.
if (found) {
        return attr;
       }
```

Listing A.20 NanoXML V3 - F6 - XMLElement class

```
// code with the embedded fault
...
str = ", SystemID='" + systemID + "'";
//original code
,,.
str += ", SystemID='" + systemID + "'";
```

Listing A.21 NanoXML V3 - F7 - XMLException class

```
// code with the embedded fault
...
str = XMLUtil.read(this.reader, ' ');
//original code
,,.
str = XMLUtil.read(this.reader, '&');
```

Listing A.22 NanoXML V5 - F1 - ContentReader class

```
// code with the embedded fault
...
Removed Line
//original code
,,.
continue;
```

Listing A.23 NanoXML V5 - F2 - ContentReader class

```
// code with the embedded fault
...
Removed Line
//original code
,,.
if (external) {
        this.peLevel = 1;
    }
```

Listing A.24 NanoXML V5 - F3 - NonValidato class

```
// code with the embedded fault
...
if (! XMLUtil.checkLiteral(this.reader, "CDATA")) {
//original code
,,.
if (! XMLUtil.checkLiteral(this.reader, "CDATA[")) {
```

Listing A.25 NanoXML V5 - F4 - StdXMLParser class

```
// code with the embedded fault
...
XMLUtil.errorExpectedInput(reader.getSystemID(),
                  reader.getLineNr(), "<DOCTYPE");
//original code
,,.
XMLUtil.errorExpectedInput(reader.getSystemID(),
                 reader.getLineNr(), "<!DOCTYPE");
```

Listing A.26 NanoXML V5 - F5 - StdXMLParser class

```
// code with the embedded fault
...
Removed Line
//original code
,,.
```

```
6 buffer.append(ch);
```

Listing A.27 NanoXML V5 - F6 - StdXMLParser class

```
1 // code with the embedded fault
2 ...
3 this.entities.put("amp", "&#38;#38;");
4 //original code
5 ,,.
6 this.entities.put("amp", "&#38;");
```

Listing A.28 NanoXML V5 - F7 - XMLEntityResolver class

```
1 // code with the embedded fault
2 ...
3 "<!-");
4 //original code
5 ,,.
6 "<!--");
```

Listing A.29 NanoXML V5 - F8 - XMLUtil class

```
1 // code with the embedded fault
2 ...
3 case Tokenizer.T_ID: return new AttributeValue(t.ival);
4 //original code
5 ,,.
6 case Tokenizer.T_ID: return new AttributeValue(t.sval)
```

Listing A.30 Siena V2 - F1 - SENP class

```
1 // code with the embedded fault
2 ...
3 if (x == null || y == null) return true;
4 //original code
5 ,,.
6 if (x == null && y == null) return true;
```

Listing A.31 Siena V4 - F1 - SENP class

```
1 // code with the embedded fault
2 ...
3 case BOOL: return bval != 0;
4 //original code
5 ,,.
6 case BOOL: return bval;
```

Listing A.32 Siena V6 - F1 - AttributeValue class

```
// code with the embedded fault
...
if (prog.cur < prog.end)
//original code
,,.
if (prog.cur <= prog.end)
```

Listing A.33 Sed V5 - F1 - Sed

```
// code with the embedded fault
...
ch = *prog.cur++;
//original code
,,.
ch = *(prog.cur+1);
```

Listing A.34 Sed V5 - F2 - Sed

```
// code with the embedded fault
...
addr->addr_number = in_integer(ch);
//original code
,,.
addr->addr_number = inchar();
```

Listing A.35 Sed V5 - F3 - Sed

```
// code with the embedded fault
...
cur_input.string_expr_count = ++string_expr_count;
//original code
,,.
cur_input.string_expr_count = string_expr_count;
```

Listing A.36 Sed V5 - F4 - Sed

Appendix B

An Example to Explain the Encoding Scheme in More Detail

For input/output pairs coding scheme, let us call the NanoXML example in the Table in chapter 3. The input file for NanoXML is an XML file that contains an element (Flower in our case) and some properties for this element (colour, smell, name and season). The output file will be a html file that shows the contents in tabular form (in the example the output will be the name of element only - Flower). To simplify this to a form that could be handled by the learning algorithms we tokenised this string by considering the first letter of each element and its properties for the input - F for Flower, CR for Colour and Red, SS for Smell and Sweet, NR for Name and Rose, SS for Season and Spring and so on for other NanoXML examples (note that even though some letters are duplicated they are unique in their contexts). The output in this example will return the element name which is Flower and coded as F. We applied that automatically to all examples and then we checked random samples to validate that the coding strategy has not missed any valuable information. The validation procedure was done manually by comparing the actual input/output pairs (actual data before transformation) of selected samples and the coded version of the same input/output pairs.

The coding scheme for one trace execution example of NanoXML will be explained as follow:

- Trace generated by Daikon is shown on Figure B.1.

- From such trace we extract sequences of of method invocations (ENTER) and method exits (EXIT) in the exact order as they occur during test execution. Figure B.2 shows extracted sequences for chosen example (1260 sequences).

```
    rep-type hashcode
    flags is_param
    comparability 22
    parent REnumChildren_wy_v1:::OBJECT 1

ppt REnumChildren_wy_v1.main(java.lang.String[]):::ENTER
ppt-type enter
  variable arg0
    var-kind variable
    dec-type java.lang.String[]
    rep-type hashcode
    flags is_param
    comparability 22
  variable arg0.getClass()
    var-kind function getClass()
    enclosing-var arg0
    dec-type java.lang.Class
    rep-type java.lang.String
    function-args arg0
    flags synthetic classname
    comparability 22
  variable arg0[..]
    var-kind array
    enclosing-var arg0
    array 1
    dec-type java.lang.String[]
    rep-type hashcode[]
    comparability 22
  variable arg0[..].toString
    var-kind function toString()
    enclosing-var arg0[..]
    array 1
    dec-type java.lang.String[]
    rep-type java.lang.String[]
    function-args arg0[]
    flags synthetic to_string
    comparability 22

ppt REnumChildren_wy_v1.main(java.lang.String[]):::EXIT59
ppt-type subexit
  variable arg0
    var-kind variable
    dec-type java.lang.String[]
    rep-type hashcode
    flags is_param
    comparability 22
  variable arg0.getClass()
    var-kind function getClass()
```

Fig. B.1 Trace Generated by Daikon

```
net.n3.nanoxml.XMLParserFactory.createDefaultXMLParser():::ENTER
net.n3.nanoxml.StdXMLBuilder.StdXMLBuilder():::ENTER
net.n3.nanoxml.StdXMLBuilder.StdXMLBuilder():::EXIT69
net.n3.nanoxml.XMLParserFactory.createXMLParser(java.lang.String, net.n3.nanoxml.IXMLBuilder, net.n3.nanoxml.IXMLReader, net.n3.nanoxml.IXMLValidator):::ENTER
net.n3.nanoxml.StdXMLParser.StdXMLParser():::ENTER
net.n3.nanoxml.XMLEntityResolver.XMLEntityResolver():::ENTER
net.n3.nanoxml.XMLEntityResolver.XMLEntityResolver():::EXIT63
net.n3.nanoxml.StdXMLParser.StdXMLParser():::EXIT84
net.n3.nanoxml.StdXMLParser.setBuilder(net.n3.nanoxml.IXMLBuilder):::ENTER
net.n3.nanoxml.StdXMLParser.setBuilder(net.n3.nanoxml.IXMLBuilder):::EXIT95
net.n3.nanoxml.NonValidator.NonValidator():::ENTER
net.n3.nanoxml.XMLEntityResolver.XMLEntityResolver():::ENTER
net.n3.nanoxml.XMLEntityResolver.XMLEntityResolver():::EXIT63
net.n3.nanoxml.NonValidator.NonValidator():::EXIT87
net.n3.nanoxml.StdXMLParser.setValidator(net.n3.nanoxml.IXMLValidator):::ENTER
net.n3.nanoxml.StdXMLParser.setValidator(net.n3.nanoxml.IXMLValidator):::EXIT107
net.n3.nanoxml.XMLParserFactory.createXMLParser(java.lang.String, net.n3.nanoxml.IXMLBuilder, net.n3.nanoxml.IXMLReader, net.n3.nanoxml.IXMLValidator):::EXIT168
net.n3.nanoxml.XMLParserFactory.createDefaultXMLParser():::EXIT85
net.n3.nanoxml.StdXMLReader.fileReader(java.lang.String):::ENTER
net.n3.nanoxml.StdXMLReader.StdXMLReader(java.io.InputStream):::ENTER
net.n3.nanoxml.StdXMLReader.stream2reader(java.io.InputStream, java.lang.StringBuffer):::ENTER
net.n3.nanoxml.StdXMLReader.getEncoding(java.lang.String):::ENTER
net.n3.nanoxml.StdXMLReader.getEncoding(java.lang.String):::EXIT194
net.n3.nanoxml.StdXMLReader.stream2reader(java.io.InputStream, java.lang.StringBuffer):::EXIT295
net.n3.nanoxml.StdXMLReader.startNewStream(java.io.Reader):::ENTER
net.n3.nanoxml.StdXMLReader.startNewStream(java.io.Reader):::EXIT495
net.n3.nanoxml.StdXMLReader.StdXMLReader(java.io.InputStream):::EXIT336
net.n3.nanoxml.StdXMLReader.fileReader(java.lang.String):::EXIT131
net.n3.nanoxml.StdXMLParser.setReader(net.n3.nanoxml.IXMLReader):::ENTER
net.n3.nanoxml.StdXMLParser.setReader(net.n3.nanoxml.IXMLReader):::EXIT118
net.n3.nanoxml.StdXMLParser.parse():::ENTER
net.n3.nanoxml.StdXMLReader.getLineNr():::ENTER
net.n3.nanoxml.StdXMLReader.getLineNr():::EXIT503
net.n3.nanoxml.StdXMLBuilder.startBuilding(int):::ENTER
net.n3.nanoxml.StdXMLBuilder.startBuilding(int):::EXIT81
net.n3.nanoxml.StdXMLParser.scanData():::ENTER
net.n3.nanoxml.StdXMLReader.atEOF():::ENTER
net.n3.nanoxml.StdXMLReader.atEOF():::EXIT424
net.n3.nanoxml.StdXMLBuilder.getResult():::ENTER
net.n3.nanoxml.StdXMLBuilder.getResult():::EXIT248
net.n3.nanoxml.XMLUtil.read(net.n3.nanoxml.IXMLReader, boolean[], char, net.n3.nanoxml.XMLEntityResolver):::ENTER
net.n3.nanoxml.StdXMLReader.read():::ENTER
net.n3.nanoxml.StdXMLReader.read():::EXIT374
net.n3.nanoxml.XMLUtil.read(net.n3.nanoxml.IXMLReader, boolean[], char, net.n3.nanoxml.XMLEntityResolver):::EXIT406
net.n3.nanoxml.StdXMLParser.scanSomeTag(boolean):::ENTER
net.n3.nanoxml.XMLUtil.read(net.n3.nanoxml.IXMLReader, boolean[], char, net.n3.nanoxml.XMLEntityResolver):::ENTER
net.n3.nanoxml.StdXMLReader.read():::ENTER
net.n3.nanoxml.StdXMLReader.read():::EXIT374
```

Fig. B.2 Extracted Sequences

- The trace compression algorithm developed by Nguyen et al. [69] is used (see the algorithm below). The algorithm replace the collections of method sequence entry and exit (Figure B.2) values with their hash key consisting usually of 1 or 2 characters (Figure B.3). For instance, the hash key 12 is the representation for the method sequence net.n3.nanoxml.XMLWriter.write(net.n3.nanoxml.XMLElement, int):::EXIT152 and if the same method sequence occurs during the execution it will also be represented by the hash key 12 and so on.

Fig. B.3 Hash Key for the Collections of Method Sequence

- The compression algorithm takes the repetition of method sequence into account (126 sequences in the final trace representation out of 1260 sequences). For example, the method sequence (net.n3.nanoxml.XMLWriter.write(net.n3.nanoxml.XMLElement, int):::EXIT152) occurred twice, the algorithm will take one of this method sequence in the final trace representation (Figure B.4).

Generally, the coding strategy for the execution traces has not caused any loss in the information, and at any time testers can use the hash key value for specific ENTER or EXIST sequence method as shown in Figure (B.3) and used this value to go back to the actual extracted sequence method as shown in Figure (B.3). Note that, the traces are only compared directly with each other and their content is not used in any other way.

encode_mapping.csv – LibreOffice Calc

A1 net.n3.nanoxml.XMLUtil.skipWhitespace(net.n3.nanoxml.IXMLReader, char, java.lang.StringBuffer, boolean[]):::ENTER

	A	B
82	net.n3.nanoxml.StdXMLParser.scanData():::ENTER	22
83	net.n3.nanoxml.StdXMLParser.processElement():::EXIT467	11
84	net.n3.nanoxml.XMLWriter.write(net.n3.nanoxml.XMLElement, int):::EXIT152	12
85	net.n3.nanoxml.XMLParserFactory.createDefaultXMLParser():::EXIT85	23
86	net.n3.nanoxml.XMLElement.setContent(java.lang.String):::ENTER	N
87	net.n3.nanoxml.StdXMLReader.atEOF():::ENTER	13
88	net.n3.nanoxml.StdXMLBuilder.elementAttributesProcessed(java.lang.String, java.lang.String, java.lang.String):::ENTER	14
89	net.n3.nanoxml.XMLElement.setAttribute(java.lang.String, java.lang.String):::EXIT363	Z
90	net.n3.nanoxml.StdXMLParser.processSpecialTag(boolean):::ENTER	24
91	net.n3.nanoxml.StdXMLParser.scanSomeTag(boolean):::EXIT205	0H
92	net.n3.nanoxml.ContentReader.ContentReader(net.n3.nanoxml.IXMLReader, net.n3.nanoxml.XMLEntityResolver, char, java.lang.String, boolean, java.lang.String):::EXIT123	5
93	net.n3.nanoxml.NonValidator.NonValidator():::EXIT87	25
94	net.n3.nanoxml.StdXMLParser.setReader(net.n3.nanoxml.IXMLReader):::ENTER	26
95	net.n3.nanoxml.StdXMLReader.stream2reader(java.io.InputStream, java.lang.StringBuffer):::ENTER	27
96	net.n3.nanoxml.ContentReader.close():::EXIT237	6
97	net.n3.nanoxml.XMLElement.setAttribute(java.lang.String, java.lang.String):::ENTER	7
98	net.n3.nanoxml.XMLElement.getChildAtIndex(int):::EXIT274	28
99	net.n3.nanoxml.StdXMLReader.StdXMLReader(java.io.InputStream):::EXIT336	1N
100	net.n3.nanoxml.StdXMLBuilder.addAttribute(java.lang.String, java.lang.String, java.lang.String, java.lang.String, java.lang.String):::ENTER	8
101	net.n3.nanoxml.StdXMLParser.setBuilder(net.n3.nanoxml.IXMLBuilder):::EXIT95	29
102	net.n3.nanoxml.XMLElement.getContent():::EXIT433	J
103	net.n3.nanoxml.NonValidator.PCDataAdded(int):::EXIT519	W
104	net.n3.nanoxml.XMLUtil.scanIdentifier(net.n3.nanoxml.IXMLReader, char, net.n3.nanoxml.XMLEntityResolver):::ENTER	L
105	net.n3.nanoxml.NonValidator.PCDataAdded(int):::ENTER	9
106	net.n3.nanoxml.StdXMLBuilder.startElement(java.lang.String, java.lang.String, java.lang.String, int):::ENTER	15
107	net.n3.nanoxml.XMLElement.XMLElement(java.lang.String, int):::EXIT128	P
108	net.n3.nanoxml.StdXMLBuilder.getResult():::EXIT248	0I
109	net.n3.nanoxml.StdXMLBuilder.addAttribute(java.lang.String, java.lang.String, java.lang.String, java.lang.String, java.lang.String):::EXIT196	0A
110	net.n3.nanoxml.StdXMLBuilder.addPCData(java.io.Reader, int):::ENTER	0B
111	net.n3.nanoxml.NonValidator.elementAttributesProcessed(java.lang.String, java.lang.String, java.lang.String, java.util.Properties):::EXIT487	16
112	net.n3.nanoxml.StdXMLParser.processCDATA():::EXIT286	2A
113	net.n3.nanoxml.XMLParserFactory.createDefaultXMLParser():::ENTER	2B
114	net.n3.nanoxml.StdXMLBuilder.endElement(java.lang.String, java.lang.String, java.lang.String):::EXIT169	17
115	net.n3.nanoxml.XMLElement.enumerateAttributeNames():::ENTER	2C
116	net.n3.nanoxml.StdXMLReader.unread(char):::ENTER	5
117	net.n3.nanoxml.NonValidator.attributeAdded(java.lang.String, java.lang.String, java.lang.String, java.lang.String):::EXIT508	0C
118	net.n3.nanoxml.XMLWriter.XMLWriter(java.io.OutputStream):::EXIT80	18
119	net.n3.nanoxml.StdXMLParser.parse():::EXIT135	2D
120	net.n3.nanoxml.StdXMLParser.processAttribute():::ENTER	0D
121	net.n3.nanoxml.StdXMLReader.fileReader(java.lang.String):::EXIT131	2E
122	net.n3.nanoxml.StdXMLParser.processCDATA():::ENTER	2F
123	net.n3.nanoxml.StdXMLParser.processElement():::ENTER	19
124	net.n3.nanoxml.XMLUtil.checkLiteral(net.n3.nanoxml.IXMLReader, char, net.n3.nanoxml.XMLEntityResolver, java.lang.String):::ENTER	2G
125	net.n3.nanoxml.StdXMLReader.read():::ENTER	1
126	net.n3.nanoxml.NonValidator.NonValidator():::ENTER	2H

Fig. B.4 The Final Trace Representation

Appendix C

Trace Compression Algorithm Developed by Nguyen et al. [69]

```
1  #!/usr/bin/env python
2  import fileinput
3  import sys
4  import os
5  import glob
6  from collections import defaultdict
7  import itertools
8  import csv
9
10 path = './'
11 if len(sys.argv) == 2:
12   path = sys.argv[1]
13 else:
14   print """
15   Usage: python encode-events [dir]
16   """
17   exit()
18
19 def generate_event_short_names(size):
20   alphabets = ['0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z']
21   new_names = []
22
23   # estimate max number of combinations
24   max_combinations = 1
25   while len(alphabets)**max_combinations < size:
```

```
26      max_combinations = max_combinations + 1
27
28    current_size = 0
29    for i in range(1, max_combinations + 1):
30      for tupl in itertools.product(alphabets, repeat = i):
31        if (current_size < size):
32          current_size = current_size + 1
33          new_names.append("".join(tupl))
34          if current_size == size:
35            return new_names
36
37 list_all_events = []
38 event_count_map= defaultdict(int)
39
40 # read the events from input files, count their occurence
41 for infile in glob.glob(os.path.join(path, '*.seqs') ):
42   print "Reading file : " + infile
43   for l in fileinput.input(infile):
44     event = l.strip()
45     if event not in list_all_events:
46       list_all_events.append(event)
47     # increase counter
48     event_count_map[event] = event_count_map.setdefault(event, 0) + 1
49
50 # sort so that event that occurs more will be encoded as a shorter
     string
51 event_count_sorted_list = sorted(event_count_map, key=event_count_map.
     get, reverse=True)
52 new_event_names = generate_event_short_names(len(event_count_sorted_list
     ))
53
54 # build the mapping
55 ee_map = {}
56 for pos,event in enumerate(event_count_sorted_list):
57   ee_map[event] = new_event_names[pos]
58
59 # for key in event_count_sorted_list:
60 #    print key, event_count_map[key]
61
62 # store mapping file
63 mapping_file = os.path.join(path, 'encode_mapping.csv')
64 f = csv.writer(open(mapping_file, "w"))
65 for key, val in ee_map.items():
66   f.writerow([key, val])
```

```
67
68 # process encoding file per file
69 outpath = os.path.join(path, "out")
70 os.mkdir(outpath)
71 os.chdir(path)
72 for infile in os.listdir("."):
73   if infile.endswith(".seqs"):
74     print "Encoding file : " + infile
75     out_file = os.path.join(outpath, infile)
76     f = open(out_file, "w")
77     for l in fileinput.input(os.path.join(path, infile)):
78       event = l.strip()
79       f.write(ee_map[event])
80       f.write("\n")
81     f.close()
```

Listing C.1 Trace Compression Algorithm

Appendix D

A Cross Validation Results for All Experiments

As requested, the tables below show ranges of the values for F-measure value over cross validation with their average values and standard deviations. The cross validation will be different based on the test suite size of the subject program. For instance, 6-fold cross validation was employed for NanoXML version 1 when 10% of the labelled data size was examined, because the test suite size was 207 test cases (126 passed test cases and 81 failed test cases). As explained in the selection of labelled and unlabelled data section, in this case the training set had 26 labelled data (21 instances labelled as passed execution and 5 labelled as failed execution), and therefore the data set was divided to 6 training sets to make sure that all passed instances were participate as labelled once and unlabelled data for other iteration. The number of fold cross validation is decreased because the size of the labelled data is increased. For example, 3-fold cross validation was employed for NanoXML version 1 when 20% of the labelled data size was examined, because the test suite size was 207 test cases (126 passed test cases and 81 failed test cases). As explained in the selection of labelled and unlabelled data section, in this case the training set had 45 labelled data (40 instances labelled as passed execution and 5 labelled as failed execution), and therefore the data set was divided to 3 training sets to make sure that all passed instances were participate as labelled once and unlabelled data for other iteration. Those two examples explained the blank entries in all tables. Note that, the labelled failed instances were kept deliberately the same and also small to maintain more realistic scenario for scenario 1.

Test Result Classification Based on Input/Output Pairs

Table D.1 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.79	0.83	0.70	0.75	0.72
Train 2	0.34	0.30	0.42	0.85	0.84
Train 3	0.36	0.29			
Train 4	0.38				
Train 5	0.38				
Train 6	0.38				
Average F-measure	**0.43**	**0.47**	**0.56**	**0.80**	**0.78**
Standard deviation	**0.17**	**0.30**	**0.19**	**0.07**	**0.08**

Table D.2 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 1 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.79	0.79	0.75	0.75	0.75
Train 2	0.34	0.35	0.70	0.70	0.70
Train 3	0.36	0.70	0.72		
Train 4	0.60	0.65			
Train 5	0.70				
Train 6	0.50				
Train 7	0.38				
Average F-measure	**0.51**	**0.62**	**0.72**	**0.72**	**0.72**
Standard deviation	**0.17**	**0.19**	**0.02**	**0.03**	**0.03**

Table D.3 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 1 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.67	0.60	0.79	0.85	0.80
Train 2	0.63	0.60	0.77	0.79	0.78
Train 3	0.65	0.56	0.78		
Train 4	0.60	0.50			
Train 5	0.60				
Train 6	0.63				
Train 7	0.60				
Average F-measure	**0.62**	**0.56**	**0.78**	**0.82**	**0.79**
Standard deviation	**0.02**	**0.04**	**0.01**	**0.04**	**0.01**

Table D.4 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 1 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.42	0.47	0.47	0.68	0.76
Train 2	0.44	0.45	0.45	0.66	0.74
Train 3	0.40	0.47	0.44		
Train 4	0.47	0.41			
Train 5	0.48				
Train 6	0.45				
Train 7	0.42				
Average F-measure	**0.44**	**0.45**	**0.45**	**0.67**	**0.75**
Standard deviation	**0.02**	**0.02**	**0.01**	**0.01**	**0.01**

Table D.5 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.38	0.48	0.38	0.28	0.28
Train 2	0.34	0.47	0.36	0.26	0.26
Train 3	0.36	0.49			
Train 4	0.38				
Train 5	0.38				
Train 6	0.38				
Average F-measure	**0.37**	**0.48**	**0.37**	**0.27**	**0.27**
Standard deviation	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**

Table D.6 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 1 using co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.34	0.16	0.14	0.14	0.15
Train 2	0.33	0.14	0.15	0.16	0.15
Train 3	0.13	0.15	0.16		
Train 4	0.32	0.15			
Train 5	0.25				
Train 6	0.29				
Train 7	0.30				
Average F-measure	**0.28**	**0.15**	**0.15**	**0.15**	**0.15**
Standard deviation	**0.07**	**0.008**	**0.01**	**0.01**	**0**

Table D.7 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 1 using co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.29	0.16	0.14	0.16	0.15
Train 2	0.30	0.14	0.15	0.14	0.15
Train 3	0.24	0.14	0.16		
Train 4	0.29	0.16			
Train 5	0.25				
Train 6	0.30				
Train 7	0.22				
Average F-measure	**0.27**	**0.15**	**0.15**	**0.15**	**0.15**
Standard deviation	**0.03**	**0.01**	**0.01**	**0.01**	**0**

Table D.8 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 1 using co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.40	0.35	0.36	0.10	0.10
Train 2	0.35	0.34	0.35	0.12	0.12
Train 3	0.39	0.36	0.34		
Train 4	0.35	0.35			
Train 5	0.33				
Train 6	0.33				
Train 7	0.30				
Average F-measure	**0.35**	**0.35**	**0.35**	**0.11**	**0.11**
Standard deviation	**0.03**	**0.008**	**0.01**	**0.01**	**0.01**

Table D.9 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using Co-EM (EM-SVM)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.35	0.35	0.37	0.29	0.32
Train 2	0.30	0.34	0.35	0.25	0.22
Train 3	0.33	0.30			
Train 4	0.35				
Train 5	0.34				
Train 6	0.31				
Average F-measure	**0.33**	**0.33**	**0.36**	**0.27**	**0.27**
Standard deviation	**0.02**	**0.02**	**0.01**	**0.02**	**0.07**

Table D.10 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using Co-training (Co-SVM)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.35	0.34	0.36	0.26	0.25
Train 2	0.34	0.30	0.36	0.28	0.29
Train 3	0.35	0.35			
Train 4	0.33				
Train 5	0.33				
Train 6	0.28				
Average F-measure	**0.33**	**0.33**	**0.36**	**0.27**	**0.27**
Standard deviation	**0.02**	**0.02**	**0**	**0.01**	**0.02**

Table D.11 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 1 using Self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.25	0.15	0.12	0.14	0.11
Train 2	0.25	0.13	0.13	0.12	0.13
Train 3	0.27	0.12	0.11	0.10	
Train 4	0.24	0.11			
Train 5	0.28	0.14			
Train 6	0.28				
Train 7	0.25				
Train 8	0.24				
Train 9	0.27				
Train 10	0.47				
Average F-measure	**0.28**	**0.13**	**0.12**	**0.12**	**0.12**
Standard deviation	**0.06**	**0.01**	**0.01**	**0.02**	**0.01**

Table D.12 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 1 using Self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.15	0.19	0.50	0.56	0.56
Train 2	0.14	0.15	0.55	0.52	0.52
Train 3	0.18	0.15	0.57	0.54	
Train 4	0.19	0.20			
Train 5	0.20	0.26			
Train 6	0.15				
Train 7	0.20				
Train 8	0.25				
Train 9	0.25				
Average F-measure	**0.19**	**0.19**	**0.54**	**0.54**	**0.54**
Standard deviation	**0.04**	**0.04**	**0.03**	**0.02**	**0.02**

Table D.13 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 1 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.10	0.12	0.04	0.14	0.14
Train 2	0.13	0.15	0.05	0.16	0.12
Train 3	0.11	0.14	0.06	0.12	
Train 4	0.10	0.17			
Train 5	0.12	0.22			
Train 6	0.14				
Train 7	0.13				
Train 8	0.15				
Train 9	0.17				
Train 10	0.15				
Average F-measure	**0.13**	**0.16**	**0.05**	**0.14**	**0.13**
Standard deviation	**0.02**	**0.03**	**0.01**	**0.02**	**0.01**

Table D.14 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 1 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.08	0.08	0.08	0.51	0.50
Train 2	0.08	0.08	0.08	0.50	0.52
Train 3	0.08	0.08	0.08	0.52	
Train 4	0.08	0.08			
Train 5	0.08	0.08			
Train 6	0.08				
Train 7	0.08				
Train 8	0.08				
Train 9	0.08				
Average F-measure	**0.08**	**0.08**	**0.08**	**0.51**	**0.51**
Standard deviation	**0**	**0**	**0**	**0.01**	**0.01**

Test Result Classification Based on Input/Output Pairs Augmented with Execution Traces

Table D.15 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.47	0.47	0.77	0.89	0.92
Train 2	0.44	0.48	0.83	0.91	0.96
Train 3	0.46	0.45			
Train 4	0.45				
Train 5	0.48				
Train 6	0.40				
Average F-measure	**0.45**	**0.47**	**0.80**	**0.90**	**0.94**
Standard deviation	**0.02**	**0.01**	**0.04**	**0.01**	**0.02**

Table D.16 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 1 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	1	1	1
Train 3	1	1	1		
Train 4	1	1			
Train 5	1				
Train 6	1				
Train 7	1				
Average F-measure	**1**	**1**	**1**	**1**	**1**
Standard deviation	**0**	**0**	**0**	**0**	**0**

Table D.17 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 1 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	1	1	1
Train 3	1	1	1		
Train 4	1	1			
Train 5	1				
Train 6	1				
Train 7	1				
Average F-measure	**1**	**1**	**1**	**1**	**1**
Standard deviation	**0**	**0**	**0**	**0**	**0**

Table D.18 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 1 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	1	1	1
Train 3	1	1	1		
Train 4	1	1			
Train 5	1				
Train 6	1				
Train 7	1				
Average F-measure	**1**	**1**	**1**	**1**	**1**
Standard deviation	**0**	**0**	**0**	**0**	**0**

Table D.19 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 1 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.45	0.45	0.44	0.44	0.45
Train 2	0.42	0.44	0.40	0.42	0.41
Train 3	0.43	0.43			
Train 4	0.45				
Train 5	0.45				
Train 6	0.44				
Average F-measure	**0.44**	**0.44**	**0.42**	**0.43**	**0.43**
Standard deviation	**0.01**	**0.01**	**0.02**	**0.01**	**0.02**

Table D.20 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 1 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	1	1	1
Train 3	1	1	1		
Train 4	1	1			
Train 5	1				
Train 6	1				
Train 7	1				
Average F-measure	**1**	**1**	**1**	**1**	**1**
Standard deviation	**0**	**0**	**0**	**0**	**0**

Table D.21 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 1 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	0.96	0.97	1	1
Train 3	1	1	1		
Train 4	1	1			
Train 5	1				
Train 6	1				
Train 7	1				
Average F-measure	**1**	**0.99**	**0.99**	**1**	**1**
Standard deviation	**0**	**0.02**	**0.01**	**0**	**0**

Table D.22 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 1 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	1	1	1
Train 3	1	1	1		
Train 4	1	1			
Train 5	1				
Train 6	1				
Train 7	1				
Average F-measure	**1**	**1**	**1**	**1**	**1**
Standard deviation	**0**	**0**	**0**	**0**	**0**

Table D.23 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 1 using Co-training (Co-SVM)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	1	1	1
Train 3	1	1	1		
Train 4	1	1			
Train 5	1				
Train 6	1				
Train 7	1				
Average F-measure	**1**	**1**	**1**	**1**	**1**
Standard deviation	**0**	**0**	**0**	**0**	**0**

Table D.24 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 1 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	1	1	1
Train 3	1	1	1	1	
Train 4	1	1			
Train 5	1	1			
Train 6	1				
Train 7	1				
Train 8	1				
Train 9	1				
Train 10	1				
Average F-measure	**1**	**1**	**1**	**1**	**1**
Standard deviation	**0**	**0**	**0**	**0**	**0**

Table D.25 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 1 using Self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.18	0.20	0.53	0.53	0.53
Train 2	0.19	0.19	0.55	0.55	0.55
Train 3	0.19	0.17	0.54	0.54	
Train 4	0.20	0.19			
Train 5	0.19	0.20			
Train 6	0.18				
Train 7	0.19				
Train 8	0.19				
Train 9	0.20				
Average F-measure	**0.19**	**0.19**	**0.54**	**0.54**	**0.54**
Standard deviation	**0.007**	**0.01**	**0.01**	**0.01**	**0.01**

Table D.26 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 1 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	1	1	1
Train 3	1	1	1	1	
Train 4	1	1			
Train 5	1	1			
Train 6	1				
Train 7	1				
Train 8	1				
Train 9	1				
Train 10	1				
Average F-measure	**1**	**1**	**1**	**1**	**1**
Standard deviation	**0**	**0**	**0**	**0**	**0**

Table D.27 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 1 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.14	0.14	0.14	0.59	0.54
Train 2	0.15	0.15	0.15	0.50	0.52
Train 3	0.13	0.14	0.13	0.50	
Train 4	0.14	0.15			
Train 5	0.13	0.12			
Train 6	0.14				
Train 7	0.15				
Train 8	0.13				
Train 9	0.15				
Average F-measure	**0.14**	**0.14**	**0.14**	**0.53**	**0.53**
Standard deviation	**0.008**	**0.01**	**0.01**	**0.05**	**0.01**

Table D.28 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 1 using Co-training (Co-SVM)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	1	1	1
Train 3	1	1	1	1	
Train 4	1	1			
Train 5	1	1			
Train 6	1				
Train 7	1				
Train 8	1				
Train 9	1				
Train 10	1				
Average F-measure	**1**	**1**	**1**	**1**	**1**
Standard deviation	**0**	**0**	**0**	**0**	**0**

Table D.29 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 1 using Co-training (Co-SVM)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.14	0.14	0.14	0.13	0.13
Train 2	0.15	0.15	0.15	0.14	0.15
Train 3	0.13	0.14	0.13	0.15	
Train 4	0.14	0.15			
Train 5	0.13	0.12			
Train 6	0.14				
Train 7	0.15				
Train 8	0.13				
Train 9	0.15				
Average F-measure	**0.14**	**0.14**	**0.14**	**0.14**	**0.14**
Standard deviation	**0.008**	**0.01**	**0.01**	**0.01**	**0.01**

Table D.30 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 1 for scenario 2 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.18	0.17	0.20	0.32	0.39
Train 2	0.20	0.20	0.44	0.28	0.37
Train 3	0.19	0.20			
Train 4	0.20				
Train 5	0.20				
Train 6	0.23				
Average F-measure	**0.20**	**0.19**	**0.22**	**0.30**	**0.38**
Standard deviation	**0.01**	**0.01**	**0.02**	**0.02**	**0.01**

Table D.31 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 2 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.98	1	1	1	1
Train 2	0.98	1	1	1	1
Train 3	1	1	0.94		
Train 4	1	1			
Train 5	1	0.90			
Train 6	1				
Train 7	0.90				
Average F-measure	**0.98**	**0.98**	**0.98**	**1**	**1**
Standard deviation	**0.03**	**0.04**	**0.03**	**0**	**0**

Table D.32 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 2 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	0.97	0.96	0.96
Train 3	1	0.96	0.97		
Train 4	1	0.96			
Train 5	0.79				
Train 6	1				
Train 7	0.79				
Average F-measure	**0.94**	**0.98**	**0.98**	**0.98**	**0.98**
Standard deviation	**0.01**	**0.02**	**0.01**	**0.02**	**0.02**

Table D.33 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 2 using self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.96	0.96	0.96	0.96	1
Train 2	1	1	1	1	1
Train 3	1	1	0.92		
Train 4	1	0.88			
Train 5	1				
Train 6	0.88				
Train 7	0.88				
Average F-measure	**0.96**	**0.96**	**0.96**	**0.98**	**1**
Standard deviation	**0.05**	**0.05**	**0.04**	**0.02**	**0**

Table D.34 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 2 for scenario 2 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.98	0.88	0.88	0.80	0.80
Train 2	0.98	1	0.80	0.96	0.96
Train 3	1	0.80	0.96		
Train 4	1	0.84			
Train 5	1				
Train 6	1				
Train 7	0.90				
Average F-measure	**0.98**	**0.88**	**0.88**	**0.88**	**0.88**
Standard deviation	**0.03**	**0.08**	**0.08**	**0.11**	**0.11**

Table D.35 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 3 for scenario 2 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	0.81	0.81	0.88	0.88
Train 2	1	0.89	0.88	0.90	0.90
Train 3	1	0.86	0.98		
Train 4	1	1			
Train 5	1				
Train 6	1				
Train 7	1				
Average F-measure	**1**	**0.89**	**0.89**	**0.89**	**0.89**
Standard deviation	**0**	**0.08**	**0.08**	**0.01**	**0.01**

Table D.36 Ranges of values for F-measure value over cross validation with standard deviation for NanoXML version 5 for scenario 2 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0	0.63	0.63	0.70	0.70
Train 2	0	0.75	0.80	0.86	0.86
Train 3	0	0.80	0.79		
Train 4	0	0.78			
Train 5	0				
Train 6	0				
Train 7	0				
Average F-measure	**0**	**0.74**	**0.74**	**0.78**	**0.78**
Standard deviation	**0**	**0.07**	**0.09**	**0.11**	**0.11**

Table D.37 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 2 using Self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	1	1	1
Train 3	1	1	1	1	
Train 4	1	1			
Train 5	1	1			
Train 6	1				
Train 7	1				
Train 8	1				
Train 9	1				
Train 10	1				
Average F-measure	**1**	**1**	**1**	**1**	**1**
Standard deviation	**0**	**0**	**0**	**0**	**0**

Table D.38 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 2 using Self-training (EM-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0	0	0.40	0.40	0.50
Train 2	0	0	0.45	0.45	0.52
Train 3	0	0	0.41	0.41	
Train 4	0	0	-		
Train 5	0	0			
Train 6	0				
Train 7	0				
Train 8	0				
Train 9	0				
Average F-measure	**0**	**0**	**0.42**	**0.42**	**0.51**
Standard deviation	**0**	**0**	**0.02**	**0.02**	**0.01**

Table D.39 Ranges of values for F-measure value over cross validation with standard deviation for Siena version 2 for scenario 2 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	1	1	1	1	1
Train 2	1	1	1	1	1
Train 3	1	1	1	1	
Train 4	1	1			
Train 5	1	1			
Train 6	1				
Train 7	1				
Train 8	1				
Train 9	1				
Train 10	1				
Average F-measure	**1**	**1**	**1**	**1**	**1**
Standard deviation	**0**	**0**	**0**	**0**	**0**

Table D.40 Ranges of values for F-measure value over cross validation with standard deviation for Sed version 5 for scenario 2 using Co-training (Co-Naïve)

Labelled size %	10%	20%	30%	40%	50%
Train 1	0.33	0.30	0.40	0.40	0.42
Train 2	0.30	0.34	0.43	0.43	0.48
Train 3	0.33	0.33	0.40	0.43	
Train 4	0.31	0.26			
Train 5	0.30	0.27			
Train 6	0.33				
Train 7	0.33				
Train 8	0.31				
Train 9	0.43				
Average F-measure	**0.33**	**0.30**	**0.41**	**0.42**	**0.45**
Standard deviation	**0.03**	**0.02**	**0.01**	**0.01**	**0.04**

Appendix E

A Thesis Data

All data for this thesis is available at http://personal.strath.ac.uk/rafig.almaghairbe/