This is to certify that the
dissertation entitled

An Integrated Approach to Autonomous Computation in Data Streaming Applications

presented by

Eric P. Kasten

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Computer Science

Major Professor's Signature

4/27/07

Date

MSU is an affirmative-action, equal-opportunity employer

Abstract
An Integrated Approach to Autonomous Computation in Data Streaming Applications

By

Eric P. Kasten

Increasingly, software needs to adapt to its environment. This need is driven in part by the emergence of pervasive computing (software that interacts with the physical world) and autonomic computing (software required to manage itself). We say that an application is adaptable if it can alter its behavior during execution. Implementation of an adaptable application integrates three elements. First, an adaptable application must implement mechanisms that enable the dynamic modification of parameters and/or program structure. For instance, an adaptable data streaming application may load and insert new transcoding or error correction components to address a drop in available bandwidth or increased packet loss. Second, when components are exchanged during an adaptation, the state of the application must be maintained. For instance, data must not be lost when exchanging error correction components or redeploying a processing element to a different host. Finally, adaptive, autonomic and dynamic data driven systems often must be able to detect and respond to errant behavior or changing conditions with little or no human intervention. Clearly, decision making is a critical issue in such systems, which must learn how and when to invoke corrective actions based on past experience and analysis of sensor data.

Our research investigates adaptation and decision making to support autonomous data-streaming applications. Data-streaming applications comprise an important class of software that includes communications, command-and-control, and environmental monitoring, among others. In these applications, adaptation may require making decisions that consider sensed environmental conditions, data stream type or data stream content. In this dissertation, we first study adaptive mechanisms, state maintenance, and decision making separately; then we integrate these elements to implement an application that can adapt to changing network conditions autonomously. Finally, we propose a technique for automated extraction and analysis of meaningful sequences, called ensembles, from sensor data streams. Ensemble extraction enables a decision maker to respond autonomously to sensed events when they recur. We investigate the utility of ensembles when applied to ecosystem monitoring and analysis of wireless computer network traces.

Alas, then I knew that to end
was but to begin anew.

To Jeannette, for all she did not
do. Whose memory serves as a
candle to focus my thoughts.

To Trofast, friend and family.
Here when my journey began
but not at its end.

Assez de raison d'être

Acknowledgments

This work was supported in part by the U.S. Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, National Science Foundation grants EIA-0130724, ITR-0313142, and CCF 0523449, and a Quality Fund Concept grant from Michigan State University.

PREFACE

Within these pages you will find my report of a journey that, like many, began with only a fleeting thought and ends with the completion of this dissertation. I consider this journey to have begun the moment when Dr. Gongzhu Hu, my then undergraduate professor, mentioned that I should, at least, complete a masters degree. This seemingly harmless assertion, sowed a seed that would eventually grow to mark the beginning of this journey. For that, I thank "Doc," as I had come to call him, for always pushing me a bit farther than I was sure I wanted to go. However, seeds do not grow unless the soil in which they are planted is fertile. There are those that corrupted my otherwise harmless and mundane youth by fueling my curiosity and love for learning and discovery. I will not try to mention them all here; some might even be horrified that I might attribute to them some of the blame for the blown circuit breakers, minor floods and other assorted unnatural disasters that complicated my early days. They have little to fear, however, everyone knew I was most central in such plots. A few, that were particularly influential, I will mention. Thank you Brian Locey for turning me on to the joy of writing. Thank you Gary and Barbara Beardsley for wanting to call me "Doctor."

When I was young I explored the cedar swamps and forests that bordered the Tobacco River. These early excursions most certainly were the first signs of my interest in research. To me, one particular adventure characterizes the essence of

research. I was about ten, and out fishing, when I discovered a school of tadpoles in a murky pool out of reach of the stronger flow of the Tobacco. Of course, I had to take them home to watch them grow into adult frogs. As the days passed, I continued to fish and waited for when the tadpoles would begin to grow legs. Alas, they never did. It turned out that they weren't tadpoles at all, but instead were bullhead fry. Bullheads are a type of catfish that have long whiskers called barbels that can cause painful puncture wounds. When I was fishing, I hated catching bullheads; taking them off the hook was often painful without pliers and gloves. My "research" into tabpole development had somehow left me as "mother" to ten or twenty bullheads. It seemed a bit barbaric to just dump them on the ground, so I returned them to the river. Thus, I learned one of my first lessons about doing research: you do not always know how an experiment is going to end and the results can be surprising. I thank my mother and father for encouraging my early research and tolerating the results, particularly those that had to be thrown back.

My sister once noted that our parent's children shared a trait that others can find a touch frustrating. Namely, that we never appeared to know what we wanted to be when we grew up. Fortunately, I ran into a special someone who gave me an extra push to pursue my masters degree. I thank my wife Barbara Morse Kasten for that push and for continuing to join me on the journeys of a man searching for what he wants to be. I also thank my brittanys Edie and Nokomis for keeping me company as I wrote this dissertation, even though I suspect Nokomis may have just been waiting for a chance to chew on it.

When working on my masters, I would take a course on computer networking, then

called CPS422. The semester project was to implement a program for streaming audio data over a computer network. My implementation would earn me an invitation to my professor's office. Apparently, he thought I showed sufficient promise to take me on as an advisee (sometimes called an experiment), but I still suspect he was simply looking for someone to install Mosaic on his workstation. As I finish this Ph.D. dissertation, Dr. Philip K. McKinley is both my advisor and guidance committee chairperson. His job cannot have been easy; once, when he asked me why I wanted a Ph.D., I responded, "So I can call you Phil!" I express great gratitude for his invaluable advice and patience during the completion of both my masters and doctoral degrees. I'd also like to thank the other members of my guidance committee for their contributions to this work and for their helpful comments, advise and teasing over the years. Thank you Dr. Betty H.C. Cheng, Dr. Sandeep S. Kulkarni and Dr. Brian T. Pentland.

I had the pleasure of meeting Dr. Stuart H. Gage during an on-campus presentation where he described his recent work collecting acoustic data for ecosystem monitoring. Again, I was reminded of my adventures on the shores of the Tobacco river. I remembered avoiding marshland and other untraversable terrain by recognizing the vocalizations of the creatures that lived there. Thus began my research on automated processing of sensor data streams for species detection. I am grateful to Dr. Gage for providing the opportunity to pursue this topic and for his advice, constant encouragement and discussion about this and other subjects. I also thank Wooyeong Joo for his constant enthusiasm and support.

Finally, I would like to thank Ron Fox, my colleague in the National Superconducting Cyclotron Laboratory, for enduring my pursuit of this Ph.D. And last but not

least, I would like to thank my colleagues in the Software Engineering and Network Systems Laboratory and in the Department of Computer Science and Engineering at Michigan State University for their insightful suggestions and discussions regarding my research. An incomplete list of those that deserve a thank you includes: Dr. Juyang Weng, Dr. S. Masoud Sadjadi, Heather Goldsby, Dave Knoester, Farshad Samimi, Jesse Sowell and Zhinan Zhou. There are those that I have likely forgotten to thank. I ask your forgiveness for this omission; your support was most appreciated.

Table of Contents

LIST OF TABLES xvi

LIST OF FIGURES xviii

1 Introduction 1

2 Background and Related Work 10
2.1 Historical Perspective . . . 11
2.2 How Adaptation Occurs: A Taxonomy . . . 14
2.2.1 Parameter adaptation . . . 14
2.2.2 Compositional adaptation . . . 15
2.3 Classification by Composition Time . . . 17
2.3.1 Load-time composition . . . 19
2.3.2 Runtime composition . . . 21
2.4 State Maintenance . . . 21
2.4.1 Reference update . . . 22
2.4.2 Migrating state . . . 22
2.4.3 Synchronizing intercession . . . 23
2.5 Example Systems . . . 24
2.5.1 Examples of component-based runtime adaptation . . . 25
2.5.2 Examples of fragment-based runtime adaptation . . . 25
2.5.3 Examples of operator-based runtime adaptation . . . 26
2.6 Decision Making in Autonomic Systems . . . 28
2.7 Towards an Integrated Design of Autonomous Software . . . 31

3 Mechanisms to Support Autonomic Software 33
3.1 A Closer Look at Reflection . . . 35
3.2 Building Adaptive Software . . . 39
3.2.1 Model of Adaptive Components . . . 42
3.2.2 The Role of Encapsulation . . . 44
3.2.3 Absorption . . . 45
3.2.4 Metafication . . . 46
3.3 A Prototype Language: Adaptive Java . . . 47
3.3.1 Basic Component Structure . . . 47
3.3.2 Absorbing Existing Classes . . . 49
3.3.3 Reifying a Meta-level . . . 50
3.4 Case Study: MetaSockets . . . 51
3.4.1 Block-Erasure Codes . . . 52

3.4.2 MetaSocket Design and Operation. 53
3.5 Mechanisms Enabling Operator Adaptation 54
3.5.1 Dynamic River Operators and Segments 56
3.5.2 Dynamic River Records 58
3.6 Related Work 61
3.7 Discussion 66

4 State Maintenance for Autonomic Software **68**
4.1 Key Concepts and Issues 69
4.1.1 Nontransient state 70
4.1.2 Component equivalence 71
4.1.3 Collateral change 72
4.2 State Transformation 73
4.3 Perimorph Design and Implementation 75
4.3.1 Component construction 76
4.3.2 References and invocations 78
4.3.3 Recomposition 79
4.3.4 Activation and deactivation 80
4.4 Example: Adaptive Queue 81
4.5 Case Study: Mapping Application 83
4.6 State Maintenance in Dynamic River 88
4.6.1 Graceful Shutdown 89
4.6.2 Fault resiliency 90
4.7 Related Work 93
4.8 Discussion 95

5 Perceptual Memory **97**
5.1 Background 99
5.2 MESO Design and Operation 101
5.2.1 Sensitivity Spheres 102
5.2.2 MESO Tree Structure 103
5.2.3 Sensitivity Sphere Size 106
5.2.4 Compression 112
5.2.5 Complexity 115
5.3 MESO Assessment 116
5.3.1 Data Sets and Experimental Method 117
5.3.2 Baseline Experiments 119
5.3.3 Comparison with Other Classifiers 123
5.4 Related Work 127
5.5 Discussion 130

6 Case Study: Adaptive Error Control **131**
6.1 Background and Related Work 132
6.2 Experimental Scenario and Method 133
6.2.1 Pattern Features 134

6.2.2 Imitative Learning . . . 135
6.3 State Maintenance . . . 137
6.4 Results . . . 138
6.5 Feature Analysis . . . 143
6.6 Discussion . . . 144

7 Automated Ensemble Extraction and Analysis of Acoustic Data Streams 147
7.1 Background . . . 151
7.1.1 Visualizing Acoustic Events . . . 151
7.1.2 Piecewise Aggregate Approximation . . . 153
7.1.3 Symbolic aggregate approximation . . . 155
7.2 Ensemble Extraction and Processing . . . 157
7.3 Assessment . . . 163
7.3.1 Data Sets and Methodology . . . 163
7.3.2 Classification Results . . . 165
7.3.3 Species Detection . . . 168
7.4 Related Work . . . 175
7.5 Discussion . . . 181

8 Forecasting Network Packet Loss 183
8.1 Trace Collection and Characterization . . . 184
8.1.1 Trace Scoring and Sampling . . . 185
8.1.2 Trace Characterization . . . 187
8.2 Packet Loss Models . . . 192
8.3 Ensemble Extraction and Processing . . . 195
8.4 Data Sets and Methodology . . . 198
8.4.1 Evaluation Metrics . . . 202
8.5 Assessment . . . 204
8.6 Related Work . . . 212
8.7 Discussion . . . 215

9 Conclusions and Future Work 216
9.1 Contributions . . . 216
9.2 Future Work . . . 221

APPENDICES 224

A Perimorph Data Dictionary of Major Components 225
A.1 BaseComponentDefinition . . . 225
A.2 BaseFactor . . . 226
A.3 BaseFactorContext . . . 227
A.4 BaseFactorset . . . 227
A.5 BaseInterface . . . 229
A.6 ComponentFactorStore . . . 232

A.7 ComponentManager 235
A.8 ComponentStore 239
A.9 DynamicLoadManager 240
A.10 FactorManager 240
A.11 FactorStore 245
A.12 FactorsetVars 246
A.13 ObjectReference 252
A.14 Parameters 253
A.15 ReturnCode 261
A.16 Scope 261
A.17 ScopeManager 262
A.18 ScopeStore 262

B Dynamic River Operators and Support Programs 264
B.1 Basic Pipeline Operators 264
B.1.1 Asciionramp 264
B.1.2 Binary2record 266
B.1.3 Binaryonramp 267
B.1.4 Cabsf 268
B.1.5 Cutout 269
B.1.6 Cutter 270
B.1.7 Daqcat 271
B.1.8 Daqtail 271
B.1.9 Daqtee 272
B.1.10 Dft 273
B.1.11 Feed 274
B.1.12 Float2cplx 274
B.1.13 Paa 275
B.1.14 Ratemeter 276
B.1.15 Raw2record 276
B.1.16 Readout 277
B.1.17 Record 278
B.1.18 Record2binary 278
B.1.19 Record2raw 279
B.1.20 Record2vect 279
B.1.21 Recorddump 280
B.1.22 Reslice 280
B.1.23 Sample 281
B.1.24 Saxanomaly 282
B.1.25 Saxbitmap 283
B.1.26 Segmenter 284
B.1.27 Sieve 285
B.1.28 Stepcutter 286
B.1.29 Steptrigger 287
B.1.30 Trigger 288

B.1.31 Wav2rec 289
B.1.32 Welchwindow 289
B.2 Network Pipeline Operators 290
B.2.1 Streamin 290
B.2.2 Streamout 292
B.3 Support Programs 293
B.3.1 Ctrlcmd 293
B.3.2 Dynriverd 294

C Tabular Data Used for Forecasting Packet Loss **295**

BIBLIOGRAPHY **302**

List of Tables

1.1 Dissertation organization and contributions. 8

3.1 Description of a subset of Dynamic River operators and support programs. 59

3.2 Description of record header fields. 62

5.1 Comparison of 6 different activation functions using $c = 0.6$ for the letter data set (see Section 5.3.1). 111

5.2 Space and time complexities for MESO and several other clustering algorithms [1]. 115

5.3 Data set characteristics. 117

5.4 MESO baseline results comparing a sequential search to MESO tree search. 121

5.5 MESO baseline results comparing different compression methods. 122

5.6 Accuracy, training and test times of IND and HDR (compare with Table 5.4). 126

6.1 Features used for training and testing the Xnaut. 134

6.2 Xnaut results with and without compression. 143

6.3 Feature contribution to MESO accuracy. 145

6.3 (cont'd) 146

7.1 Bird species codes, common names and the number of patterns and ensembles used in the experiments discussed in Section 7.3. 163

7.2 MESO classification and timing results. 166

7.3 Confusion matrix for classification using individual PAA patterns. 167

7.4 Confusion matrix for classification using ensembles comprising PAA patterns. 167

8.1 Total training and testing ensemble counts. 199

8.2 Training and testing data set characterization. 200

8.3 Training and testing data set characterization labeled with FEC codes. . 201

8.3 (cont'd) 202

8.4 MESO forecasting coverage and precision. 210

8.5 MESO forecasting coverage and precision when trained on generated data. 211

C.1 MESO forecasting accuracy with loss rate margin. Plotted in Figure 8.9. 296

C.1 (cont'd). 297

C.2 MESO forecasting accuracy when trained using generated data. Plotted in Figure 8.10. 298

C.2 (cont'd). 299

C.3 MESO forecasting accuracy with FEC code labels. Plotted in Figure 8.11(a).300

C.4 MESO accuracy when trained using generated data with FEC code labels. Plotted in Figure 8.11(b) 300

C.5 MESO forecasting accuracy with Xnaut FEC code labels. Plotted in Figure 8.12(a) 301

C.6 MESO accuracy when trained using generated data with Xnaut FEC code labels. Plotted in Figure 8.12(b) 301

List of Figures

1.1 Conceptual view of a distributed stream processing environment. 4

2.1 Metalevel understanding collected into metaobject protocols (MOPS). . . 14

2.2 Taxonomy for computational adaptation. 14

2.3 Taxonomy for software composition using the time of composition or recomposition as a classification metric. Dynamism increases from left to right. Runtime methods allow immediate or near immediate response to environmental change. 19

2.4 Object reference update problem. Left, component replacement executed using a particular reference. Right, desired result of a component replacement. 23

2.5 An intuitive schema of message filtering (adapted from [2]). In this diagram, (A), (B) and (C) are three filters, while m(), n(), o() and p() are messages. Following message m(), filter (A) rejects m(), passing it to filter (B). Filter (B) matches m() and modifies m(). Filter (C) matches the modified message m() and dispatches it to a target object. 27

3.1 Relationship between MOPS and primitive operations. 41

3.2 Dimensions of component behavior. 41

3.3 Basic Metamodel 43

3.4 Component absorption and metafication 46

3.5 Adaptive Java component structure. 48

3.6 Absorbing a class into a Component 49

3.7 Metafying a component 50

3.8 Physical experimental configuration. 51

3.9 Operation of FEC based on block erasure codes. 52

3.10 Structure of a MetaSocket. 54

3.11 Sample results of dynamically changing MetaSocket configuration. 55

3.12 Basic internal structure of basic stream operators and the `streamin` and `streamout` network operators. 57

3.13 An elided sequence diagram depicting the normal operation of the `streamout` and `streamin` pipeline operators. 58

3.14 C/C++ Definition of the Dynamic River record header. 60

3.15 Depiction of a record stream with data scoping. Numbers indicate the scope nesting depth indicated by the record header scope field. 61

3.16 The construction and alteration of a component-graph [3]. Solid nodes are active, while dashed nodes are inactive slots for storing components. The inactive slots on the left are filled on the right. Messages are redirected through the newly activated nodes. 64

4.1 Relationship of factors, factor sets and a component definition. 77

4.2 Executing a component operation. 79

4.3 Recomposition of a component where a factor is replaced by a new one from a different factor set. Nontransient state is assigned to the replacement factor set from the old. 80

4.4 Composition of the adaptive queue showing several factor sets. 82

4.5 Code segment used to recompose the adaptive queue into a vector based queue. Note the `invoke` calls for pausing and resuming queue access, the assignment of factor sets and the replacement of factors. 84

4.6 Adaptive queue example application. The left pane is the producer, the right the consumer and the center represents status information. Note the Pause and Resume messages where the array-based queue is exchanged with a vector-based queue by the control thread. 85

4.7 2D map prior to recomposition. 86

4.8 Recomposition of the DEM mapping application. Recomposition of both the map plotter and map window components is required. Operations on these components are called by the map control which does not require any change. 87

4.9 3D map following recomposition. 88

4.10 A sequence diagram depicting the graceful termination of `streamin` in response to a `stop` command. 90

4.11 A sequence diagram depicting the unexpected termination of `streamout`. 91

4.12 A sequence diagram depicting `streamin` reconnecting to `streamout`. . . 92

4.13 Data scope synchronization algorithm. 92

5.1 High level view of MESO. 100

5.2 Leader-follower algorithm (adapted from Duda and Hart [4]). 102

5.3 Sensitivity spheres for three 2D-Gaussian clusters. Circles represent the boundaries of the spheres as determined by the current δ. Each sphere contains one or more training patterns, and each training pattern is labeled as belonging to one of three categories (circle, square, or triangle). 103

5.4 MESO tree organization. The rectangles are partitions and the shaded spheres are partition pivots. Partitions are split successively until a leaf is formed where a partition contains only one sphere. 104

5.5 Building a MESO tree from sensitivity spheres. 106

5.6 Training and testing time for the letter data set (see Section 5.3.1). . . . 107

5.7 Sensitivity sphere creation algorithm. 108

5.8 Sensitivity sphere growth function denominator for $\delta = 1, 10, 100, 1,000$ and $10,000$. 109

5.9 The Gaussian 3D example dataset. 110

5.10 Snapshot frames showing MESO sensitivity spheres and mean sphere distances as the spheres are built for the Gaussian 3D example dataset. Top, sphere δ's. Bottom, mean sphere distances. 112

5.11 Effect of means compression on training and testing times for the letter data set, using fixed and variable δ. 114

5.12 Scalability with respect to training set size. For all data sets, typical standard deviations are less than 10% with respect to the corresponding mean accuracies and training times. 124

6.1 Physical network configuration used in the Xnaut case study. 133

6.2 High-level diagram of the primary decision making components. 137

6.3 A sequence diagram depicting sender/receiver interaction when changing FEC codes. 138

6.4 Xnaut results for artificially generated packet losses. 141

6.5 Xnaut results for real packet losses on a wireless network. 142

7.1 Acoustic sensor station and closeup of a Stargate sensor. 149

7.2 Top, an oscillogram (normalized) of an acoustic signal. Bottom, a spectrogram of the same acoustic signal. 152

7.3 A block diagram depicting the operators required to produce a spectrogram. 152

7.4 Example signal and results of Z-normalization and subsequent PAA processing. 154

7.5 Spectrogram of the acoustic signal (see Figure 7.2) after conversion to PAA representation (stretched vertically for clarity). 155

7.6 Conversion of the example PAA processed signal converted to SAX (adapted from [5]). 156

7.7 Using SAX bitmaps to compute an anomaly score for a signal (see [6] for more information). Number of subsequence occurrences shown over frequency. 157

7.8 Block diagram of pipeline operators for converting acoustic clips into ensembles for detection of bird species. 159

7.9 Anomaly score generated for the acoustic signal shown in Figure 7.2. . . 160

7.10 Trigger signal and ensembles extracted from the acoustic signal shown in Figure 7.2. 161

7.11 ROC curves for detection of the black capped chickadee (BCCH) and white breasted nuthatch (WBNU). 171

7.12 Semi-log scale ROC curves and precision for detection of the black capped chickadee (BCCH) and white breasted nuthatch (WBNU). 172

7.13 Power spectral density (PSD) histograms for the black capped chickadee (BCCH) and the white breasted nuthatch (WBNU). 174

7.14 ROC curves for detection of the black capped chickadee (BCCH) and the white breasted nuthatch (WBNU) using only the frequency range $\approx$[1.2kHz,6.0kHz] and $\approx$[1.2kHz,4.8kHz] respectively 175

7.15 Semi-log scale ROC curves and precision for detection of the black capped chickadee (BCCH) and the white breasted nuthatch (WBNU) using only the frequency range $\approx$[1.2kHz,6.0kHz] and $\approx$[1.2kHz,4.8kHz] respectively 176

8.1 Top, a scatter plot of the trace scores for a 30 minute roaming trace. Bottom, a spectrogram of the same roaming trace. 188

8.2 Burst and gap delay histograms and normal-probability plots for a roaming receiver. Normal-probability plots represent the delay range [3ms-50ms] scaled by subtracting 3ms and dividing by 50ms. 190

8.3 Burst and gap run-length histograms for a roaming receiver. Geometric-probability plots are scaled by dividing by the longest run-length. . . 191

8.4 1-second and 5-second Loss rate histograms and exponential probability plots. Exponential probability plots are scaled by dividing by the largest loss rate. 192

8.5 Simplified Gilbert-Eliot Model (adapted from [7]). 193

8.6 Block diagram of pipeline operators for converting network traces into ensembles for forecasting packet loss. 196

8.7 Anomaly score generated for the signal shown in Figure 8.1. 196

8.8 Step trigger signal and 5 second loss rates for the trace score shown in Figure 8.1. 197

8.9 MESO forecasting accuracy with loss rate margin. Experiments conducted using the method described in Section 8.4. 206

8.10 MESO forecasting accuracy when trained using generated data. Experiments conducted using the method described in Section 8.4. 207

8.11 MESO forecasting accuracy with FEC code labels. Left, results are produced by training and testing using patterns from the same data set. Right, results are produced by training on generated data and testing on data from the roam data set. Note, when using the same data set for training and testing, training and testing data does not overlap. Experiments conducted using the method described in Section 8.4. . . 208

8.12 MESO forecasting accuracy with Xnaut FEC code labels. Left, results are produced by training and testing using patterns from the same data set. Right, results are produced by training on generated data and testing on data from the roam data set. Note, when using the same data set for training and testing, training and testing data does not overlap. Experiments conducted using the method described in Section 8.4. 209

9.1 Achievements presented in this dissertation viewed as a whole. 220

Cha

Intr

Chapter 1

Introduction

Increasingly, software needs to adapt to dynamic external conditions involving hardware components, network connections, and changes in the surrounding physical environment [8–12]. For example, to meet the needs of mobile users, software integrated into hand-held, portable and wearable computing devices must balance several conflicting concerns, including quality-of-service, security, energy consumption, and user preferences. Moreover, the promise of autonomic computing systems [13], that enable software to dynamically self-heal and self-manage, appeals to system's administrators and users alike.

Advances in technology have enabled new approaches for sensing the environment and collecting data about the world around us; an important application domain is ecosystem monitoring [14–22]. Small, powerful sensors can collect data and extend our perception beyond that afforded by our natural biological senses. Moreover, wireless networks enable data to be acquired simultaneously from multiple geographically remote and diverse locations. Once collected, sensor readings can be assembled

into data streams and transmitted over computer networks to observatories [23, 24], which provide computing resources for the storage, analysis and dissemination of environmental and ecological data. Such information is important to improving our understanding of environmental and ecological processes. For instance, early detection and tracking of invasive species may enable the establishment of policies for their control [25–27].

Autonomic software systems require the integration of *adaptive mechanisms*, *state maintenance* and the ability to make *autonomous decisions*. That is, in addition to providing facilities that enable software to be dynamically tuned, reconfigured or recomposed, autonomous data-streaming systems must avoid loss or corruption of nontransient operator, system or data-stream state while autonomously invoking decisions that improve performance and address context-specific concerns with only high-level guidance from systems administrators and users. As such, autonomic systems can simplify the task of managing complex or context sensitive software systems.

Autonomous response to changing conditions may elicit modification of either program parameters (parameter adaptation) or program structure (compositional adaptation). In data-streaming environments, data processing can be distributed across multiple hosts connected by a computer network. Data-stream processing requires transfer of data between *operators* that execute a specific function on the data-stream. For instance, operators can transcode or filter a data stream to address the limited memory or processor capacity of a mobile device. A *pipeline computation* comprises one or more operators that may be deployed across multiple hosts. *Pipeline adaptation* can occur by changing operator parameters, exchanging an operator with another,

adding or deleting an operator, or redeploying an operator to a different host.

As shown in Figure 1.1, for a distributed data streaming environment, autonomic responses might address a degradation or failure in a computer network, by redeploying operators, and rerouting affected network traffic, or by triggering operator exchange to instantiate specialized processing in response to the detection of a particular species or environmental condition. Moreover, when data is continuously collected, automated and adaptive processing facilitates the organization and searching of the resulting data repositories. Without timely processing, the sheer volume of the data might preclude the extraction of information of interest. Addressing these problems will likely become increasingly important in the future as technology improves and more sensor platforms and sensor networks are deployed [18,28].

Observation and modification of program behavior can be formalized using computational reflection [29,30]. Maes [30] describes computational, or behavioral, reflection *as the activity of a computational system when doing computation about (and by that possibly affecting) its own computation.* Computational reflection comprises the separate acts of *introspecting* and *interceding* on program (or pipeline) behavior, enabling implementation details and program state to be observed and program behavior modified. In addition, *dynamic compositional adaptation* occurs at runtime and often requires the implementation of techniques that address state maintenance. State maintenance for autonomic systems needs to address both the exchange of individual components or operators and the effect of recomposition on the system as a whole. The runtime exchange of a single algorithmic or structural component requires either (or both) the transfer of nontransient state between an old component

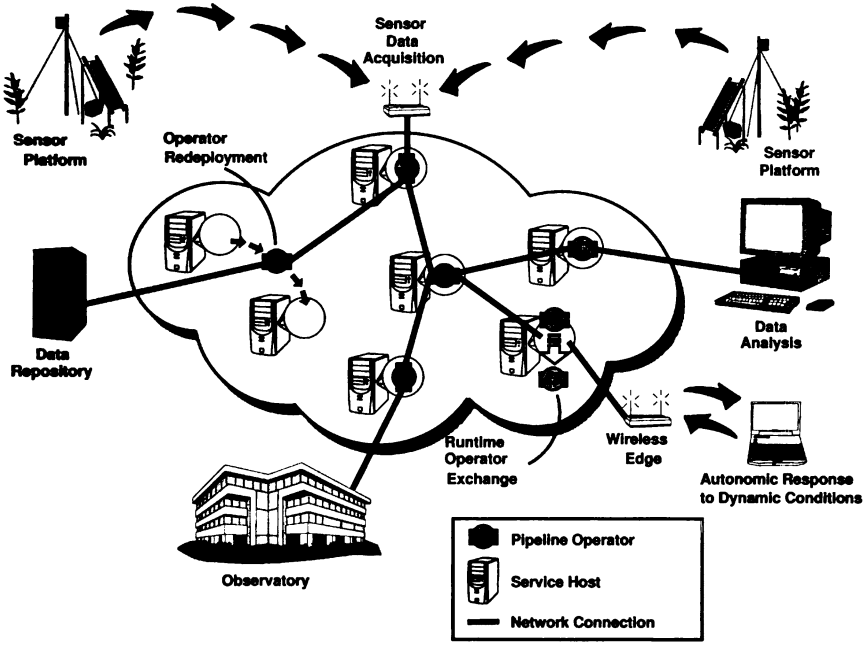


Figure 1.1: Conceptual view of a distributed stream processing environment.

and its replacement or a protocol for removing the old component from use while introducing the new component. For instance, the replacement of a pipeline error correction operator must not cause the loss of inflight data. While the state capture problem has been addressed in other contexts, such as checkpointing [31,32], process or thread migration [32–34] and mobile agents [35,36], the methods employed there generally are not directly applicable because they either incur too much overhead or do not support state transfer between different implementations of a component. Moreover, recomposition in adaptive software involves state transfer as it relates to *collateral change*, which we define as the set of recompositions and parameter changes that must be applied to an application atomically, or in an orchestrated fashion, for

continued correct execution.

Finally, a system cannot be autonomous without an autonomous decision maker to consider system and user requirements and intercede to meet these needs. The system must be able to learn from past experience and apply this knowledge to future situations. This dissertation proposes that the integration of adaptive mechanisms, state maintenance and autonomous decision making enables a software system to adapt to the uncertainty found in many dynamic computing environments.

Thesis Statement. *Integration of adaptive mechanisms, state maintenance and autonomous decision making enables implementation of autonomous, adaptive data-streaming computing applications.*

Table 1.1 depicts the basic organization of this dissertation with respect to its major contributions. This table categorizes the work presented in this dissertation according to whether a study addresses adaptive mechanisms, state maintenance or decision making and by application level (either program or pipeline). The major contributions of this dissertation are summarized as follows.

1. *We investigate adaptive mechanisms that enable observation and runtime recomposition of software.* We propose and evaluate a programming model that separates intercession and introspection. We have developed *Adaptive Java* [37,38], an extension to Java that incorporates programming language constructs to support instrumentation and dynamic recomposition. Our group has used Adaptive Java to design and evaluate the MetaSocket [39–43] component, whose behavior can be adapted in response to changing network conditions by enabling struc-

tural reconfiguration. In addition, we developed *Dynamic River*, a distributed, data-stream pipeline platform that enables dynamic recomposition of pipeline operators across multiple hosts. Our studies show that introspection and intercession are useful abstractions for designing adaptive and autonomic software both for component-based programs and distributed pipeline processing.

2. *We investigate the issue of state maintenance in adaptive software.* We have designed and implemented *Perimorph*[1] [44], an API that enables the capture and migration of state between components during recomposition. We show that enabling the externalization and management of component state facilitates application recomposition in the face of component exchange. Moreover, we introduce the concept of *collateral change* and enable the declaration of sets of program modifications that must happen atomically when an application is adapted. In a case study, we demonstrate that externalizing state enables application handoff between different devices in a mobile computing environment while enabling recomposition to meet the resource capabilities of different devices. In addition, we consider state maintenance for distributed pipeline processing. We developed Dynamic River, a distributed stream processing engine (SPE), and introduce the concept of *data stream scope* to enable recomposition of data stream operators and graceful recovery in the face of software, host or network failure. These studies demonstrate that state maintenance is a key

[1]The term perimorph is borrowed from geology. A perimorph is a crystal that contains another crystal of a different type. We use it here as an allusion where crystal facets are considered to be components or factors of compositional structure.

concern when designing and implementing adaptive and autonomic software.

3. *We investigate the effect of perceptual memory, a type of long-term memory for remembering external stimulus patterns [45, 46], on the autonomous decision making process.* Storing and retrieving external stimuli and associated meta information in dynamic environments is typically *incremental, data intensive* and *time sensitive.* Moreover, storage and recall must be efficient and avoid affecting the function of the application being adapted while enabling the decision maker to make correct and timely decisions. We have designed and implemented the perceptual memory system, *MESO*[2] [47,48], to address these requirements. We show that MESO accurately retrieves prior experience and can be used to help an autonomous decision maker adapt and optimize an underlying adaptable application.

4. *In a case study, we investigate and evaluate the integration and application of adaptive mechanisms, state maintenance and decision making to adaptive software and data streaming.* We demonstrate that integration of these technologies enables implementation of an autonomous, adaptive application that can balance packet loss with bandwidth consumption as a user roams about a wireless cell. Moreover, we evaluate the ability of a decision maker to learn through interaction with a user or operate completely autonomously while attempting to adapt and optimize the underlying mobile computing application.

[2]The term MESO refers to the tree algorithm used by the system (Multi-Element Self-Organizing tree).

5. *We introduce a technique for automated extraction and analysis of ensembles from sensor data streams.* We investigate the utility of using ensembles for classification and detection of bird species using acoustic data streams and for forecasting near term packet-loss when streaming data to a mobile receiver. Our investigations show that ensembles enable classification, detection and forecasting of time-series events that can be used by autonomic decision makers when adapting an application.

Table 1.1: Dissertation organization and contributions.

<table>
<tr><th></th><th>Adaptive Mechanisms</th><th>State Maintenance</th><th>Decision Making</th></tr>
<tr><td rowspan="2">Program Level</td><td>Adaptive Java
Chapter 3</td><td>Perimorph
Chapter 4</td><td>MESO
Chapters 5 and 6</td></tr>
<tr><td colspan="3">Case study
Chapter 6</td></tr>
<tr><td rowspan="2">Pipeline Level</td><td rowspan="2">Dynamic River
Chapter 3</td><td rowspan="2">Stream Scope
Chapter 4</td><td>MESO
Chapters 5, 7 and 8</td></tr>
<tr><td>Ensembles
Chapters 7 and 8</td></tr>
</table>

The remainder of this dissertation is organized as follows. Chapter 2 provides background on computational adaptation and autonomous software decision making and motivates the need for our research. Chapter 2 also introduces a taxonomy of computational adaptation and classifies some representative research projects. Chapter 3 describes the mechanisms required to support computation adaptation and our experience using Adaptive Java to construct adaptable components. Chapter 3 also introduces the Dynamic River and discusses the mechanisms needed for dynamic recomposition of a distributed pipeline. Chapter 4 discusses state maintenance and

our experience using Perimorph for designing and implementing state-aware software that considers adaptive composition and *collateral change*. In addition, Chapter 4 introduces the concept of *data stream scope* and its implementation in Dynamic River. Chapter 5 describes the design, implementation and assessment of our perceptual memory system, MESO, to address our decision-making objectives. Chapter 6 describes the integrated design, evaluation and analysis of the case study on adaptive error control. Chapter 7 describes our technique for automated *ensemble* extraction and analysis of sensor data streams for classification and detection of bird species using ensembles. Chapter 8 extends our use of ensembles for forecasting network packet loss. Finally, in Chapter 9, we conclude this dissertation and briefly overview future investigations.

Cha

Bac

Chapter 2

Background and Related Work

In this chapter, we review recent work in dynamic compositional adaptation and autonomous software with an eye towards data streaming applications. We consider data streaming applications to be an important class of autonomous software with applications to network traffic management, habitat monitoring, military logistics, immersive environments, and data acquisition and analysis. Moreover, adapting data stream filters, or operators, is nontrivial. For instance, it has been shown that computing an optimal ordering for pipeline filters for data selection is NP-hard when each filter is not independent from the processing of filters preceding it in the pipeline [49]. First, we provide a brief history of compositional adaptation and highlight how changes in technology have helped ignite recent interest in autonomous software. Second, we introduce taxonomies on how and when adaptation occurs and discuss the mechanisms that enable dynamic composition. Third, we discuss background on state maintenance and describe and compare example works. Fourth, we consider recent developments in software-based decision-making. Finally, we discuss open areas in

autonomous system research and motivate the need for the studies described herein.

2.1 Historical Perspective

The history of compositional adaptation dates back to the inception of the von Neumann architecture [50] in the 1940's. This computer design, where both program instructions and data are stored in mutable memory, enabled instructions to act on both data and other instructions. Such "self-modifying code" has been used for dynamic optimization. For instance, calculation of an often used branch condition can be completed once, and all dependent branch instructions are modified such that recalculation of the conditional can be avoided. Another example is the execution of a program that is too large for available memory. Such a program can rewrite or load the next segment and adjust the program counter as necessary. However, virus and worm implementations have also used self-modifying code to avoid detection or enable infection of other machines. Conversely, countermeasures for security sensitive software have used self-modifying code to avoid infection or manipulation by attackers [51]. Self-modifying code can also cause problems with compiler and processor caching, where cached instructions are executed in favor of newly rewritten code.

Malicious uses of compositional adaptation, coupled with the difficulty of understanding and ensuring the correctness [52,53] of self-modifying programs, has yielded the commonly held belief that such implementations should be avoided. However, continued research in this area has been motivated by the promise that adaptive software could enable better management of complex systems and automate response to

changing conditions or user requirements. These features are especially important as computing systems interact increasingly with the physical world. Three enabling technologies are recognized as key to designing and implementing adaptive software [9]: separation of concerns, component-based design and computational reflection. These technologies enable a principled (as opposed to ad hoc) approach to designing and implementing adaptive software.

Separation of concerns [54] enables the separate development of an application's functional behavior, or *business logic*, and the code that implements *crosscutting* [55] nonfunctional behavior, such as quality of service (QoS), fault tolerance and security. Separation of concerns is important to compositional adaption because it helps the developer to identify, design and implement separate nonfunctional subsystems that are often targets for adaptation. For instance, aspect-oriented programming (AOP) [55–59], introduced in the 1990's, is one of the most widely used approaches to codifying different software concerns. In AOP, units of composition are code fragments, called *aspects*, that are inserted, or woven, into application components during execution.

Component-based design [60] is similar to object-oriented programming (OOP) in that both objects and components embody specific programmatic behavior. However, components extend OOP by enabling software units, developed by third parties, to be deployed and composed to produce different implementations of a program. Use of components supports both static and dynamic composition of software. In static composition [61–64], components are assembled at compile or load time to produce an application. In dynamic composition [34, 65–75], a *composer* can reconfigure an

application at runtime. A composer may be either a user or a software decision maker imbued with the ability to autonomously recompose the application to address changing user requirements and the uncertainty found in dynamic environments.

Computational reflection [30,76] allows a program to observe and possibly modify its own behavior. As such, reflection comprises two activities: *introspection* and *intercession.* Introspection, or enabling an application to observe its own behavior, can be used to facilitate environmental awareness and the adaptive process [68,75,77]. On the other hand, intercession enables a system or application to modify its own behavior. Where introspection allows software to consider how best to adjust its behavior in response to changing conditions, intercession enables an application to act on these observations. Reflection was introduced into languages such as 3-LISP [76], 3-KRS [30] and Smalltalk [78], in the 1980's. Language-based implementations demonstrated the feasibility of using reflection to enable software to observe and potentially alter its own structure. In the 1990's and early 2000's, systems such as MetaXa [79] or Composition Filters [2,80] used behavioral reflection [79] to effect runtime modification of base-level program control flow.

As depicted in Figure 2.1, views encompassing environmental concerns and application understanding can be collected and categorized as *metadata.* Metadata presents a view of program context that is considered important to the adaptive process. For instance, program structure or platform and device awareness might be provided as metadata to the application. Metaobject Protocols (MOPS) have been used by several researchers [74, 75, 81–83] to provide a principled way to introduce reflection into adaptive systems.

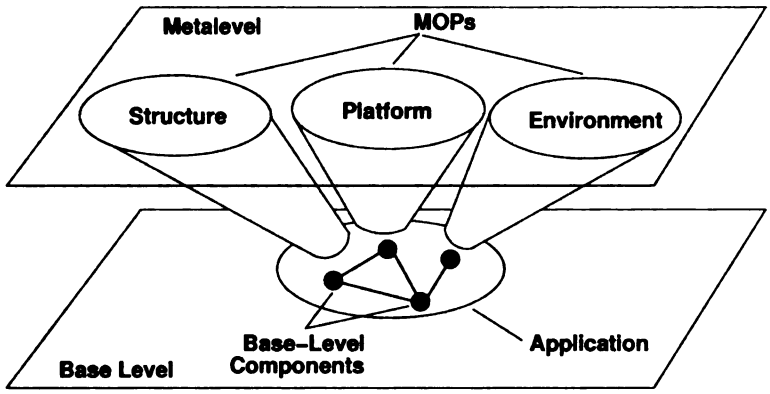


Figure 2.1: Metalevel understanding collected into metaobject protocols (MOPS).

2.2 How Adaptation Occurs: A Taxonomy

As shown in Figure 2.2, we view research on compositional adaptation as having proceeded along two basic branches defined by *how* intercession affects application behavior. A more detailed survey of many of the research works introduced below and how they fit into our taxonomy can be found in [84].

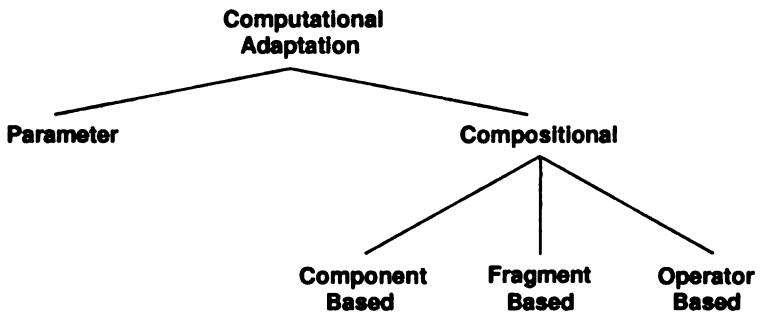


Figure 2.2: Taxonomy for computational adaptation.

2.2.1 Parameter adaptation

Parameter adaptation[1] [77,85,86], influences an application or system by modifying variables that define how a program behaves. As noted by Hiltunen and Schlicht-

[1]Parameter adaptation is sometimes referred to as transformational adaptation.

ing [53], an early example of parameter adaptation is the way that the TCP protocol adjusts its behavior by changing values that affect the algorithms controlling window management and retransmission timeout in response to apparent network congestion [87, 88]. Parameter adaptation can be implemented in middleware or by using toolkits, frameworks or specialized languages. For instance, Puppeteer [89] is middleware that executes adaptive parameter modification on behalf of an application, for example, transcoding multimedia data for display on a mobile device. The application may have little knowledge of the changes made by the middleware. Language support, such as that provided by the Program Control Language (PCL) [90], is a flexible method for supporting adaptation. However, using specialized languages also places a significant burden on designers and programmers to understand how to best codify an adaptive system. A weakness of parameter adaptation is that it cannot adopt algorithms or components left unimplemented during the original design and construction of an application. That is, parameters can be tuned or an application can be directed to use a different *existing* strategy, but strategies implemented following the construction of the application cannot be adopted.

2.2.2 Compositional adaptation

Compositional adaptation [85] results in the exchange of algorithmic or structural parts of the system with those that improve a program's fit to its current environment [52,53,80,91–93]. In comparison to parameter adaptation, compositional adaptation enables an application to adopt new algorithms and strategies for addressing

concerns unforeseen during original design and construction. The flexibility of compositional adaptation enables more than simple tuning of program variables or strategy selection. Instead, for example, new error correction or encryption algorithms can be dynamically added to existing network protocols, helping improve reliability and security.

Compositional adaptation can be further classified as either *component-based*, *fragment-based* or *operator-based*. The term *component* commonly refers to a replaceable class or object that can be reused in the construction of different software systems [55]. As such, components are often plug-compatible within a certain domain, enabling one component to replace another without modifying other parts of the software. As a simple example, let us consider two components having the same interface and implementing a queue data structure, one implementation using a fixed-size array and the other a dynamically-sized vector. These two components are considered compatible in that one component might be replaced with the other. Component-based adaptation replaces entire components, possibly during program compilation, at load-time or during execution.

Alternatively, fragment-based composition refers to the process of constructing components out of smaller building blocks. As an example, a queue object might be assembled out of three separate algorithmic blocks and one structural block. Blocks implementing `put()`, `get()` and `peek()` methods plus a structural block for the underlying data structure are assembled into a queue component. Fragment-based approaches to adaptation replace individual blocks without changing others. Continuing our earlier example, a queue's `put()` method block might be upgraded, to correct

or improve on the existing implementation, without changing the `get()` or `peek()` blocks. However, in this approach, care must be taken that replacement blocks are compatible with those left unchanged.

Operator-based composition is similar to component-based composition in that program construction uses replaceable, plug-compatible units. However, programs are constructed using operators that comprise and execution element, such as a thread or operating system process, and perform a specific function. Operator-based composition is commonly found in distributed and data-streaming environments where operator migration between hosts enables adaptation to address quality of service or fault tolerance. For instance, migrating operators among hosts can be used to achieve distributed load balancing. Moreover, dynamic insertion and removal of operators enables customization of data stream processing in response to changing environmental conditions. For example, detection of a specific bird species by a sensor platform, using acoustics, may elicit the insertion of data-stream operators that route the vocalizations of this species for further analysis. Examples of component [3,65,66,68,94], fragment-based [70,72–74] and operator-based [34,67,93,95–97] systems will be discussed further in Section 2.5 and in Chapters 3 and 4.

2.3 Classification by Composition Time

In this section, we present a second taxonomy, categorizing compositional adaptation according to *when* adaptation occurs. When composition and recomposition takes place at compile or load time, dynamism is limited, simplifying or eliminating many

concerns that are critical for correct runtime recomposition. When software undergoes structural change during execution, however, it often exhibits complex and changing interactions that can be difficult to coordinate.

Figure 2.3 shows a taxonomy for adaptive composition where composition time is the classification metric. As labeled on the right, *static composition* includes *development time* and *compile/link time* methods. A program composed at compile or link time demonstrates little ability to adaptively recompose itself in response to environmental concerns. However, a limited form of adaptation can be implemented by changing how an application is composed during recompilation or relinking. Link-time methods make use of previously compiled components, usually classes or objects, as the units of composition. Typically, components belonging to the same domain share the same interface and similar function. For example, two components that both implement a queue, one with an array and another with a vector, might be exchanged at link time.

Despite demonstrating little dynamism, compile/link-time methods can be used to customize a program to address the requirements of a different environment, such as a new computing platform or network type. For example, aspect-oriented programming languages, such as AspectJ [58], can assemble objects and classes out of algorithmic and structural blocks called *aspects*. As such, different aspects can be woven together, during compilation, to form a version of a program designed to better fit the environment where it will be used.

The static assembly of different versions of an application can be realized using generative methods [55, 98, 99], whereby an application is described using a special-


Increasing Flexibility

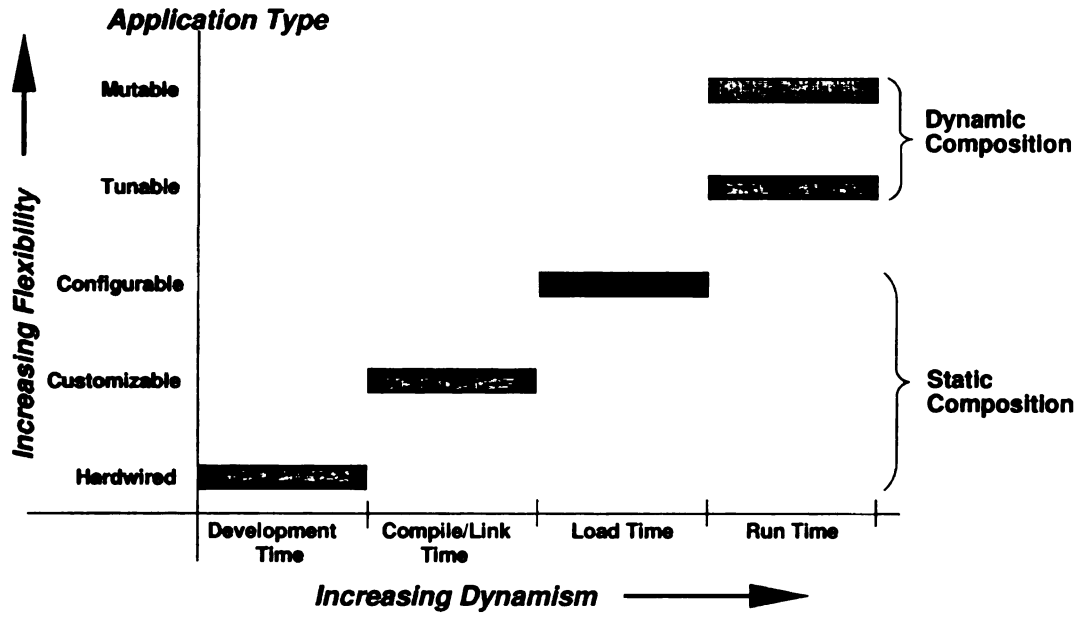


Figure 2.3: Taxonomy for software composition using the time of composition or recomposition as a classification metric. Dynamism increases from left to right. Run-time methods allow immediate or near immediate response to environmental change.

ized programming or high-level declarative language. A compiler or generator then assembles the customized programs. Programs built in this way often form *families* [100], the members of which exhibit a large number of similar characteristics, but differ in ways critical to their use in different situations or environments.

2.3.1 Load-time composition

Load-time composition allows the final decision on what algorithmic units to use in the current environment to be delayed until the component or fragment is loaded by a running application. Load-time composition is similar to static composition, but the increased dynamism also provides this method with some similarities to dynamic composition. Load-time composition requires that classes or objects be dynamically loaded and configured after execution has begun. For instance, the Java Virtual

Machine (JVM) loads classes when they are first used by a Java application. When the application requests the loading of a new component, adaptive decisions may select between different components with different capabilities or implementations based on quality of service, synchronization, security or other requirements, choosing the one that most closely matches current needs. Other load-time methods may dynamically rewrite binary code [62,63] or modify object interfaces [64] when a class is loaded.

One example of load-time composition is Binary Component Adaptation (BCA) [62]. BCA allows existing Java class files to be modified as they are loaded by the JVM. An adaptation is codified in a file that specifies the addition of interfaces, methods or other modifications to a particular class that is compiled into a *delta file*. Using delta files and byte code splicing, BCA can add functionality when a class is loaded. For example, a mobile communication application may require enhanced security to prevent eavesdropping. When the classes for sending and receiving messages are loaded into the JVM, new code for encrypting and decrypting messages is spliced into existing methods.

For adaptation in data-streaming environments, Infopipes [101–103] uses Smart Proxies [96] to enable compile- or load-time adaptation to address quality-of-service requirements in multimedia data flows. Smart proxies enable transparent support for customized data stream processing and communication between a client and server. For instance, data can be compressed or encrypted by a smart proxy in a fashion transparent to the implementation of a client. Smart proxies can be either statically linked to a client or dynamically shipped to the client at load time.

2.3.2 Runtime composition

Runtime composition provides the greatest flexibility. Recomposing an application at runtime allows adaptation without restarting a program and without lengthy interruption of service [34,53,65,67,68]. As such, assaults on the security of an application or a drop in quality of service can be observed and can elicit responses while the application continues to function. Algorithmic and structural units can be replaced or even extended in response to problems or user requirements without halting and restarting the program [67,68,96]. For instance, wireless networks may exhibit dramatic changes in network error levels during the lifetime of an audio connection. The addition of algorithmic components that enhance error correction can help maintain the quality of service desired by the user [53,93,96,97]. Let us next examine runtime composition, the primary focus of this dissertation, in more detail by discussing two major elements required by such systems: state maintenance and decision making.

2.4 State Maintenance

Runtime composition can be used to construct programs that better fit the environment in which they must operate. However, runtime approaches focus on the replacement and exchange of algorithmic and structural units while an application executes. This increased flexibility gives rise to problems not found in static or load-time composition. Examples include reference update, state migration [53,68,91] and synchronization [53,91,92].

2.4.1 Reference update

As shown in Figure 2.4, when one component is exchanged for another it is necessary to update the references pointing to the old component such that they refer to the new one. This action is necessary to ensure that proper program execution continues. For instance, if a producer and consumer thread communicate through a shared array-based queue, replacement with a vector-based queue would sever communication unless both the producer and consumer adopt the new queue as their communication path. A common characteristic of many recomposable systems [34,67,68,96,97] is the decoupling of an application through indirection. This allows an application access to an object in a consistent way, independent of the object's implementation. As such, the implementation of an object can be modified without changing how it is used by the application. A dynamically recomposable system must allow for decoupling an application such that components can be exchanged. Moreover, decoupling can eliminate the need to update references to shared objects in the face of component exchange.

2.4.2 Migrating state

Two components can be considered *plug-compatible* if they have identical interfaces and similar function. However, even if two components are plug-compatible, they may have implementations that differ algorithmically or structurally. Continuing our earlier example, a queue can be implemented using an array of limited length or a vector that can increase its size dynamically, but an array cannot simply be copied

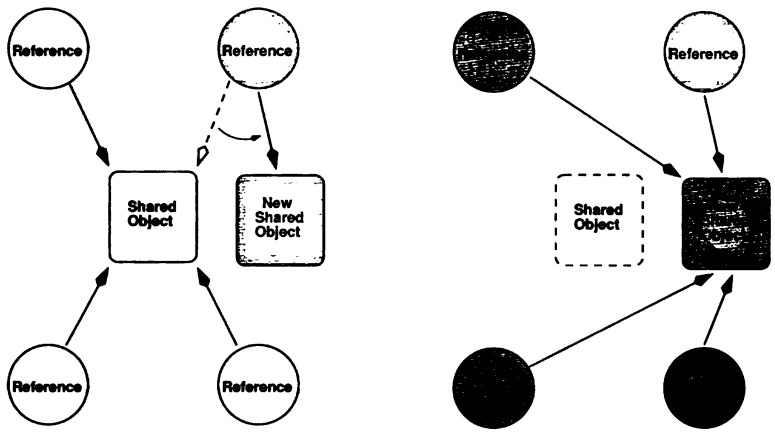


Figure 2.4: Object reference update problem. Left, component replacement executed using a particular reference. Right, desired result of a component replacement.

onto a vector byte-by-byte. Nor can a `put()` operation designed for an array be used to prepend something onto a vector. If two components are to be interchanged at runtime, it may be necessary to *extract the state from one component, transform it into a new but equivalent form, and then inject it into its replacement.*

2.4.3 Synchronizing intercession

When one component is replaced by another, the process must be synchronized such that one component can be unplugged and the other transparently plugged into the application. For instance, during the process of transferring state or when neither component is fully inserted, an application must avoid using either the old or the new component. As an example, a multithreaded application, where two threads both use a shared queue, should disallow either thread from modifying the contents of the queue during replacement of the queue component. If the threads were al-

lowed to continue to execute queue operations during the exchange, one thread could execute a `get()` on the new queue while the other thread `put()`s data on the old queue. Moreover, transferring the state from one queue implementation to the other without synchronization could yield ambiguous results, possibly resulting in loss of data or other undesirable and incorrect behavior. In a data-streaming environment, application adaptation may require the implementation of protocols to ensure synchronization of the data flow to prevent data loss during operator exchange. For instance, an application that streams data to a mobile device may choose to encrypt the data when operating atop an insecure network. However, both the sender and receiver must use the same type of encryption or the receiver will not be able to decipher the data stream. Thus, the sender and receiver must coordinate the insertion of encryption and decryption operators to maintain meaningful data transfer.

2.5 Example Systems

A subset of the reference update, state capture and synchronization problems are addressed in each of the projects introduced below. A more detailed discussion of most of the following works can be found in [84]. Most projects have some ability to update references and synchronize change with application processing. However, only a few, such as dynamicTAO [68] and DACIA [67], address a limited form of state capture and maintenance.

2.5.1 Examples of component-based runtime adaptation

Although reusability and dynamism can be considered as orthogonal to one another, dynamic recomposition shares many characteristics with reusable software. Reusable components often present well-known interfaces and functions such that they can be exchanged with others that have a similar interface and function. Both reuse and recomposition require replacement of one component with one that is better fit for the current environment and context. Research on component-based adaptation is diverse. Network, switch-level reprogramming, like that used in SwitchWare [65,104] and Cactus [53,66,91,105], allows customization of network functions to better suit the requirements of different network applications. Other projects, such as DAS/LEAD++ [3] and GILGUL [94], provide program languages for building applications and systems that support component-based adaptation. All of these projects aim to enhance flexibility, mobility and fault tolerance in changing environments through the exchange of components.

2.5.2 Examples of fragment-based runtime adaptation

Reference update, the migration and maintenance of state, and synchronization issues also affect fragment-based composition. Fragment-based approaches can target functional or nonfunctional composition. Many fragment-based, runtime approaches can be likened to dynamic weaving techniques [57, 70, 73, 74, 106], found in aspect-oriented programming, that enable an application to be rewoven during execution. Fragment-based approaches adapt an application by replacing or augmenting the code

fragments that compose a crosscutting concern. These code fragments are viewed as being woven into the application structure, attaching to classes and objects at specified locations. Composition Filters [2, 72, 80] differs by using containers or wrappers to implement indirection and decoupling, enabling reweaving of an application at runtime. Preprocessing of source code produces objects wrapped with a special *interface*. Depicted in Figure 2.5, *filters* can be applied to these interfaces, filtering incoming and outgoing messages. In this manner, filters are used as functional units similar to aspects enabling fragment-based composition. Notably, this work has some similarity to data stream processing where the program flow is treated as a stream and behavior can be adapted through the insertion and removal of filters.

2.5.3 Examples of operator-based runtime adaptation

Runtime adaptation in distributed environments may migrate, exchange, insert or delete operators to distribute load, provide fault tolerance, or modify data stream processing to respond to sensed changes in the environment. Local or remote network socket connections, coupled with protocols for disconnecting and connecting operators, is one common mechanism for enabling recomposition. In such systems, a composer orchestrates an adaptation across multiple hosts to reconfigure processing to meet current needs. State maintenance may require the migration of nontransient operator state or ensure that data is not lost during operator migration. The Computing Communities project [95] enables device awareness and virtualization [34], decoupling devices, such as monitors and keyboards, from the processes that use them.

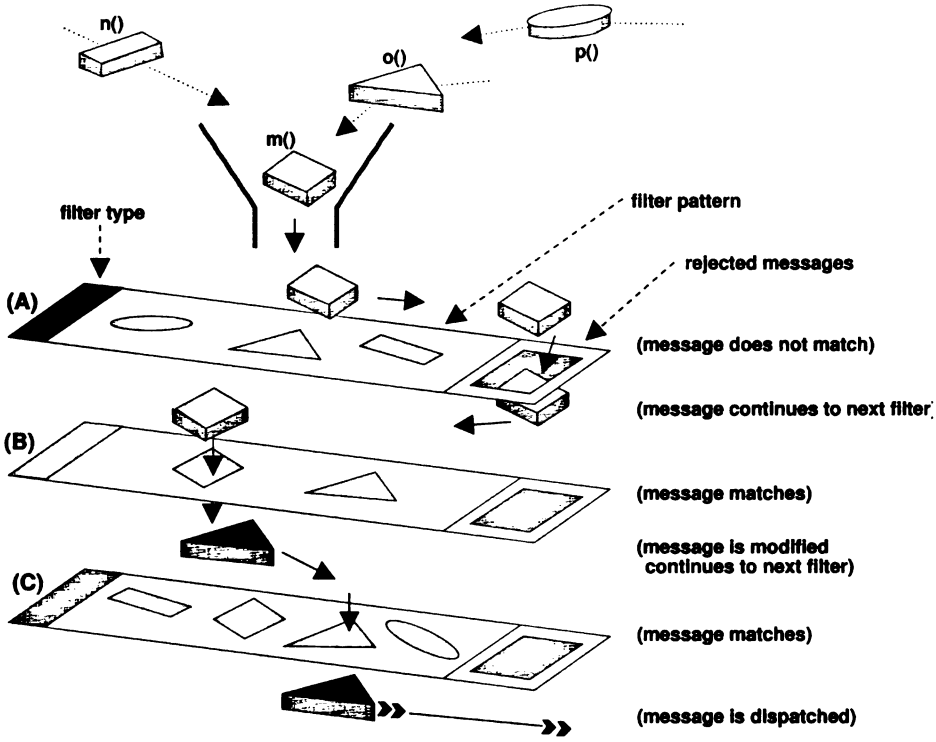


Figure 2.5: An intuitive schema of message filtering (adapted from [2]). In this diagram, (A), (B) and (C) are three filters, while m(), n(), o() and p() are messages. Following message m(), filter (A) rejects m(), passing it to filter (B). Filter (B) matches m() and modifies m(). Filter (C) matches the modified message m() and dispatches it to a target object.

This decoupling allows a distributed computing cluster to adapt to changes in processor load and application requirements through process migration. M-Ware [97, 107] uses the ECho [108] publish-subscribe system to adapt a distributed data stream. Resubscription to services on different hosts enables runtime adaptation to maximize the business value, expressed using utility functions, of data stream processing. However, state maintenance is not addressed and data can be lost during recomposition. Composable, Adaptive Network Services Infrastructure (CANS) [109, 110]

is an application-level infrastructure supporting distributed data-stream processing. Data-stream applications are composed using operators, called drivers, that contain only soft state that does not need to be retained. CANS supports data path reconfiguration involving insertion and deletion of drivers while avoiding data loss by retransmitting data segments that were not received by down-stream drivers.

2.6 Decision Making in Autonomic Systems

In addition to advances in adaptive mechanisms and state maintenance, new approaches that enable software to make decisions in real time are needed. While confronting a dynamic physical world, such *decision making* systems must act autonomously, modifying software composition to better fit the current environment while preventing damage or loss of service. Decision makers must monitor both their physical and virtual environments using software and hardware sensors. These sensors can provide information on current quality of service, such as network packet loss or available battery power. Moreover, pervasive computing environments may require that software *learn* about user behavior, enabling an application to make decisions based on a user's preferences and needs.

A number of different approaches have been used for implementing decision makers. Some projects have used rule-based methods, where adaptive behavior is guided by selecting a response from a set of previously specified condition-action pairs [111]. Other approaches are supported by theoretical models, including those based on control theory [105, 112] and those inspired by biological processes, such as the human

nervous system [113] and emergent behavior in species that form colonies [114]. Planning [115,116] strives to automatically compute system configurations to address current conditions. While these methods have been effective in certain domains, environmental dynamics and software complexity have defied their general application. Moreover, the interaction between different components, fragments and services may conflict, prohibiting the deployment of many configurations. Such feature interaction problems [117,118,118] have a combinatorial complexity and can be difficult to characterize, particularly when the number of possible compositions is large. Nonviable compositions need to be recognized prior to their use to ensure that the adaptation recommended by a decision maker preserves the correctness and safety properties required of the original system.

Another problem is that most of the above approaches are not designed to accommodate high-dimensional sensory data.[2] Decision makers may have access to hundreds or thousands of sensor readings. One example is a computer vision application [119], where a single camera (sensor) image can easily have more than 10,000 pixel values each of which can be treated as a single sensor reading. Moreover, the wide variety of environmental data that can be captured for making a decision needs to be distilled or filtered such that only information pertinent to a particular decision need be considered. Otherwise, extraneous readings can occlude a decision maker's interpretation of current conditions and cause selection of incorrect or counter productive adaptive actions. Some decision makers may learn from past experience and user

[2]High dimension means many sensors and/or each sensor has many scalar measures at each time instant.

interaction, enabling them to adapt an application's function over time [120, 121]. Such incremental learners must continue to learn from new experience and accommodate new adaptations as they become available. The amount of data processed during a system's lifetime may be very large, requiring that some behaviors be "forgotten" to limit both memory consumption and processing requirements. Moreover, these decision makers may need to tolerate the removal or addition of sensors gracefully, continuing to guide application recomposition in the face of new or incomplete information.

We will argue in Chapter 5 that *perceptual memory*, a type of long-term memory for remembering external stimulus patterns [45], may offer a useful model for an important component of decision making in context-aware, adaptive software. The ability to remember complex, high-dimensional patterns that occur as a product of interaction between application users and the environment, and to quickly recall associated actions, enables timely, autonomous system response. Moreover, systems that can learn how to interact with complex pervasive computing and ubiquitous [122, 123] environments while accounting for user preferences are becoming increasingly important.

2.7 Towards an Integrated Design of Autonomous Software

The integration of adaptive mechanisms, state maintenance and autonomous decision making is necessary for enabling software to respond to the uncertainty found in dynamic environments. In this dissertation, we investigate the integration of these elements in the design and implementation of data streaming applications. As a class of software found in many important arenas, such as environment monitoring and data communication networks, data streaming applications and systems have been studied extensively. However, the principled integration of adaptive mechanisms, state maintenance and autonomous decision making to autonomously address user desires and environmental uncertainty has received substantially less attention.

Our integrated approach requires a solid understanding of each element, and requires careful consideration and design to help ensure correct software execution while addressing the needs of different users and the changing environment. Our research is mainly focused on investigating each element and exploring how it can be effectively integrated to provide autonomous software response in dynamic environments. We recognize that many other application classes may benefit from our integrated approach, but prefer to limit our scope and extend what we learn in the data streaming environment at a later time.

In Chapter 3, we describe the design and implementation of Adaptive Java, an extension to the Java programming language that enables the implementation of dynamically recomposable components. Using Adaptive Java we explore the effec-

tiveness and expressiveness of language support in representing adaptive mechanisms, clarifying our understanding on how behavioral reflection enables dynamic software recomposition. Chapter 3 also introduces Dynamic River and investigates the mechanisms needed for dynamic recomposition of distributed pipeline operators. In Chapter 4, we investigate state maintenance issues using a programmer's API that we developed to enable capture of component state and address collateral change during dynamic recomposition. In addition, Chapter 4 also introduces the concept of *data stream scope* and its implementation in Dynamic River to maintain data-stream state during runtime recomposition. Next, in Chapter 5, we describe our design of a perceptual memory system that can be used to capture and organize data acquired through environmental sensors and user interaction. Perceptual memory enables decision making in dynamic environments by providing access to past sensor experience and user interaction. Chapter 6 describes the integrated design, evaluation and analysis of our Xnaut case study, where an application "learns" to balance error correction and bandwidth consumption through interaction with a user. Chapter 7 describes our proposed methods for automated *ensemble* extraction and analysis of sensor data streams for classification and detection of bird species in natural environments. Ensemble extraction addresses the need for techniques that enable timely distillation of information from raw sensor data that can be used for making adaptive decisions. Chapter 8 extends the use of ensembles to forecasting network packet loss. Finally, in Chapter 9, we summarize our work, discuss future directions and conclude.

Chapter 3

Mechanisms to Support Autonomic Software

Understanding the mechanisms that enable dynamic software reconfiguration is essential to the design and implementation of autonomous software. As described in Chapter 2, approaches to compositional adaptation often involve reflection [29, 30]. In this chapter, we explore two approaches for implementing mechanisms that support runtime recomposition. First, we study the idea of using language constructs to separate the two major aspects of reflection. Our approach differs from other approaches that adopt an interpretation of reflection in which the processes of observing behavior (introspection) and changing behavior (intercession) are intermingled. Specifically, we develop constructs for defining metamodels in terms of two types of primitive operations: *refractions*, which provide a (limited) view of the underlying base-level component, and *transmutations*, which modify the functionality of the base-level component. In this manner, the proposed techniques are intended

to complement existing approaches to adaptive software design by facilitating the development of higher-level adaptive services such as MOPS.

Second, we describe the design of Dynamic River, a system we developed to investigate the mechanisms and state maintenance required for dynamic recomposition of pipeline operators. Dynamic River operators can transform, filter or perform other processing on a data stream. Specifically, we study mechanisms that enable operators to be dynamically reconfigured and migrated among hosts. Runtime reconfiguration and migration can address quality of service and fault tolerance. For instance, by redeploying operators to hosts with more resources or by enabling customization of data stream processing. Dynamic River complements other projects that support distributed stream processing by directly considering state maintenance for operator-based runtime adaptation, discussed further in Chapter 4.

The remainder of this chapter is organized as follows. Section 3.1 introduces the properties of computation reflection as they apply to behavioral reflection. Section 3.2 discusses our basic approach to building adaptive components. Section 3.3 presents the design and implementation of Adaptive Java. A case study, implemented by our research group, is presented in Section 3.4, demonstrating the utility of our approach. In Section 3.5, we describe the mechanisms used by Dynamic River to enable dynamic reconfiguration and redeployment of operators. Related work is presented in Section 3.6. Finally, Section 3.7 concludes this chapter.

3.1 A Closer Look at Reflection

In this section, we discuss how the concepts provided by computational reflection can be used to realize a principled approach to runtime reconfiguration; we also describe the theoretical limitations of reflection. We begin by discussing the five properties, defined by Maes [30], that are considered important in the design and implementation of an object-oriented reflective architecture. These properties, listed below, were defined with respect to an object oriented language (OOL) designed to support structural reflection. However, these properties can be recast to address the design of a system that supports behavioral reflection by considering them with respect to enabling dynamic reconfiguration.

1: Disciplined separation between the object-level and the meta-level.
2: Uniform self-representation.
3: Complete self-representation.
4: Consistent self-representation.
5: Modifications to the meta-level result in changes to the runtime computation.

The first property indicates that the meta-level and base-level are cleanly separated. That is, design and implementation of base-level function should not be entangled with the design and implementation of the meta-level. This separation promotes a clean design while enabling base and meta-level design and implementation to be considered separately. From the perspective of enabling runtime behavioral reflection, we would like to defer codifying methods for computing QoS or invoking a reconfiguration until such time that they are needed. Moreover, depending on the context in which the system is deployed, different QoS metrics or reconfigurations may be required.

Deferring meta-level codification requires mechanisms that enable the addition or exchange of base and meta-level constructs at runtime. These mechanisms can take the form of language constructs, such as encapsulation, or software "hooks" that can be used to introduce new components to the running system. Moreover, as implied by the second property, these mechanisms should be uniform, such that the system can be observed and reconfigured in a consistent fashion. With respect to structural reflection, this means that object fields and methods are also modeled as objects and that they also have a reflective interface. Thus, a property of both structural and behavioral reflection is the presentation of a uniform interface.

The third property, from the point of view of the design of a reflective object-oriented language, implies each object in the system has a meta-representation. It is impossible, in a practical sense, to create all possible meta-representations of all objects in the system, since each meta object used to compose the meta-level is itself an object and recursively has a meta-object. This results in an infinite sequence of meta objects. A common method for handling this "problem of infinite towers of reflection" is to delay construction of meta-level objects until they are needed. *Reifying*, or making an abstract object into one that is concrete, constructs meta-objects only when the program needs them. In a system supporting runtime reconfiguration, reification of meta-level constructs is also deferred until they are required.

Another way to state properties 4 and 5 is that a system is causally connected. Property 4 essentially indicates that any modification to the base-level is reflected in the meta-level. Conversely, Property 5 ensures that any modification made to the meta-level is carried back to the base-level. These properties are necessary for both

structural and behavioral reflection to provide a useful interface for observing and modifying a system. If the meta and base-levels were not causally connected, the meta-level would be no more than a snapshot representation of the base-level at the time the meta-level was instantiated. After instantiation, the meta-level would exist independently. All observations and changes made to the meta-object would have no significance with regard to the base-level.

Now, let us consider the limitations of reflection. With respect to reflective software, introspection refers to the act of computing some truth about itself. Moreover, like all computational systems, reflective software is an instance of an axiomatic, symbolic system. A symbolic system is one composed of symbols representing objects, such as numbers, operations or schemas. In a computer, symbols are instructions and axioms are the rules that govern their execution. An axiomatic system, when properly defined, is *consistent.* Specifically, no two syntactically correct propositions in the system can be derived such that the two propositions are contradictory. That is, given a proposition *P*, *P* and ***not*** *P* cannot be true in a consistent system. An inconsistent system enables both *P* and ***not*** *P* to be provably true assertions. Since an introspective computational system is self-referencing, symbolic and axiomatic, limitations on what can be computed (or decided) are addressed directly by *Gödel's Theorem on Undecidability* [124,125] (sometimes called *Gödel's Incompleteness Theorem*) proposed in 1931. This observation stems directly from the causal connectivity of the meta and base-levels. Whatever happens to the base-level must be correctly reflected in the meta-level or causal connectivity ceases. However, if causal connectivity survives then it necessitates that information afforded to the base-level by the

meta-level be directly coupled with the state of the base-level. As such, the base-level is examining itself indirectly, but precisely.

Thus, all introspections and intercessions applied to the meta-level are directly representative of what happens in the base-level. Turing's *halting problem* [126] tells us that there are things that a program cannot decide about itself. Even when using a meta-level to indirectly introspect on the base-level, a program cannot decide all things about itself. As such, reflective systems are considered as *incomplete*. The alternative is that they are *inconsistent*, enabling the computation of contradictory "truths" about system state.

Our observations about the properties and theoretic limitations of runtime behavioral reflection are two-fold. First, the principled design and implementation of runtime recomposable software requires careful consideration of the mechanisms to support behavioral reflection. These mechanisms must separate the base and meta-levels such that the design and implementation of the base-level does not require intimate knowledge of meta-level function. Moreover, despite this separation, a causal connection must be maintained to ensure accurate introspection on base-level behavior. Second, computational introspection has its limitations. Precisely, reflection does not remove the theoretic limitations imposed on axiomatic, symbolic systems.

Hence, Maes' five properties, even though some of them may not be fully attainable, represent the basic goals of a reflective framework. In a nutshell, it is desirable to implement these five properties such that the tendency of the system to preempt how it is later used is *reasonably* minimized. This objective applies both during the implementation of an application that uses this framework and during runtime re-

configuration.

3.2 Building Adaptive Software

An important issue that arises in the application of reflection to software systems, is the degree to which the system should be able to change its own behavior. As discussed by Kiczales for meta-level interfaces [127] (and earlier by Shaw and Wulf for programming languages [128]) *preemption* occurs when the designer of a programming language or framework makes a decision in the implementation that prevents a programmer from using a feature of the language or framework in a way that would otherwise seem natural. That is to say, decisions made when the framework is implemented preemptively restrict how a programmer can effectively use the framework.

On the other hand, a completely open implementation implies that an application can be recomposed entirely at runtime. Specifically, it is possible for all the default components of the system to be destroyed and and new ones instantiated such that the goal of the imperative (base-level) computation is changed. For example, this extreme allows a calculator to be recomposed as a video player. Thus, runtime recomposition can produce a system that is inconsistent with the programmer's intended goal. Moreover, a higher degree of openness in a system entails greater resource consumption to construct and maintain the necessary data structures that enable runtime reflection. In particular, storage and processing power are needed to support meta-models [77, 83, 129], which must be reified as objects and maintain a causal connectivity to the base-level. Thus, complete openness, particularly at run-

time, is not entirely desirable. In fact, it seems that greater openness is more desirable in languages than in runtime reflective systems.

A central focus of our approach is the reflective interfaces exhibited by components. Rather than considering MOPS as orthogonal portals into base-level functionality [75], we consider an alternative architecture in which MOPS are constructed from a set of primitive operations. While different MOPS address different aspects of behavior, they may well overlap in their use of these primitives.

Figure 3.1 illustrates this view of MOPS and their composition. Different MOPS are defined for different dimensions of adaptability (e.g., fault tolerance, security, quality-of-service, power consumption). Each MOP accesses the base layer through a subset of the primitive operations, and these subsets may intersect. This design appears to exhibit several desirable features. First, explicitly defining intersections in MOP functionality may facilitate coordinated adaptation to events. Second, additional MOPS can be constructed to address issues that did not arise in the original design. Third, limiting interaction with the base level may improve the ability of the system to check, at runtime, the consistency of modifications with the specified behavior of the component. Finally, since MOPS are composed of mutable primitives, they can be adapted to meet the subjective concerns of adaptive agents. MOPS can be augmented or new ones built such that these agents can construct views of the system to suit their needs.

In the remainder of this section, we describe an approach to defining and constructing such primitive meta-operations that is based on whether the operation involves introspection or intercession of the base-level. As shown in Figure 3.2, we can

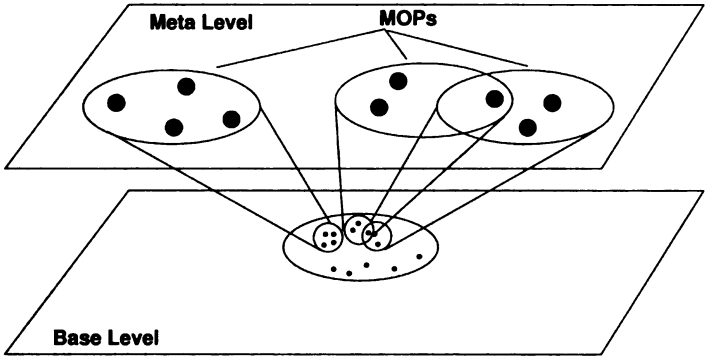


Figure 3.1: Relationship between MOPS and primitive operations.

view these as functionally orthogonal to each other and to the imperative computation of the application. The computation dimension of the application has the goal of fulfilling the principle goal imbued by the designer. The goal of the introspection dimension is to allow the application to observe itself, while that of the intercession dimension is to allow the application to modify its own behavior and structure.

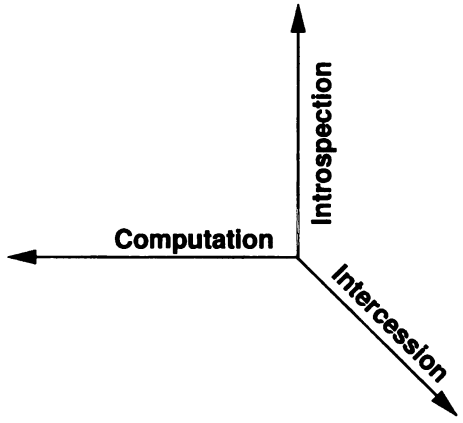


Figure 3.2: Dimensions of component behavior.

3.2.1 Model of Adaptive Components

The basic building blocks used in our adaptive system are *components*. A component can be accessed through three interfaces corresponding to the three dimensions discussed above. Operations in the computation dimension are known as *invocations*; Operations in the introspection dimension are called *refractions*, since they offer only a partial view of internal structure and behavior. Operations in the intercession dimension are called *transmutations*; they are used to transform the imperative behavior of the component. Following are formal definitions of these terms.

Definition 1 Computation *is the interpretation of the imperative function of a computer application.*

Definition 2 *A* **refraction** *is a function for observing an application's composition, resources or other internal properties in a principled fashion.*

Definition 3 *A* **transmutation** *is a function for transforming an application's computational interpretation in a principled fashion.*

Refractive and transmutative interfaces are reified by meta-components to support introspection and intercession. Figure 3.3 illustrates the structure implied by our understanding of these component interfaces. In this figure, the meta-level reflects the base-level computation. A causal connection between the meta-level and the base-level is maintained such that any changes resulting from the use of refractive and transmutative interfaces are carried to the base-level.

With the above definitions in hand, a formal definition of computational metamorphosis can be presented.

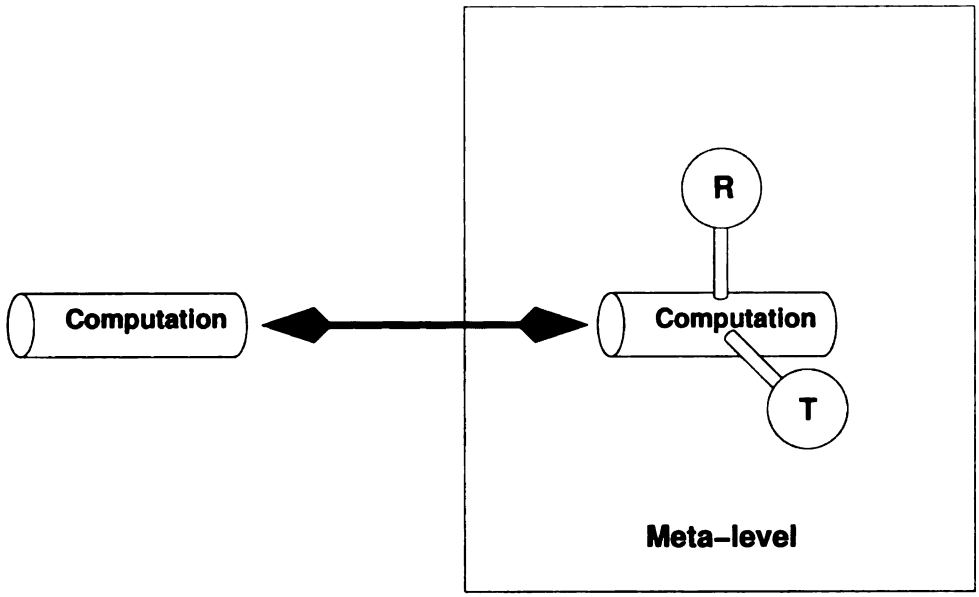


Figure 3.3: Basic Metamodel

Definition 4 Computational metamorphosis *is the principled transmutation of of a running system from one composition to another such that the imperative goal remains unchanged. The morphology of the imperative function comprises the set of compositions that have the same imperative goal. Each member of this set is called a polymorph.*

In short, this definition implies a necessary condition of correctness such that a runtime recomposable system can only be considered correct as long as the imperative goal of the system is maintained. Implicitly, it can be assumed that such a system remains functional such that it does not exhibit anomalous behavior as a result of being recomposed.

3.2.2 The Role of Encapsulation

Most object-oriented languages are based on a static binding of inheritance between subclasses and superclasses. This structure prohibits dynamic restructuring of a program at runtime. For this reason, we adopt encapsulation, where one component contains another, as the principle mechanism for the composition in our system. Encapsulation provides a means by which the functionality of a component can be extended or limited by dynamically encapsulating it within another. Moreover, the addition, deletion and exchange of encapsulated components can be carried out dynamically at runtime.

The composition of a system can be viewed as the parameterization of one component with another. We represent these relationships using notation from GenVoca [98,130]. Specifically, $S = G[F]$ states that system S is composed of component G parameterized by F, or that F is encapsulated by G. Multiple components can be encapsulated within another component, and this relationship is represented as $S = G[E, F]$. The morphology of a system built using encapsulation can be described by a set of system definitions:

$$S = \begin{cases} G[E, F] \\ G[E, F[A, B]] \\ H[G[E, F]] \end{cases}$$

Moreover, a polymorph may result from encapsulating (parameterizing) an existing system within a new component such that $S = H[G[E, F]]$.

3.2.3 Absorption

Components are constructed from objects defined in an object-oriented language (OOL). We used Java in our study. The process of constructing a component from an existing class is referred to as *absorption.* In effect, an object, as provided for by the OOL, can be considered a component without refractive and transmutative capacity. That is, objects are essentially black boxes that do not facilitate reflection.

Absorbing components provides a way to recognize when actions that bore down through the encapsulation layers, such as inheritance, should terminate. Figure 3.4 illustrates the absorption of a class and the metafication of the resulting base-level component to support refractions and transmutations. As part of the absorption procedure, mutable methods called *invocations* are created on the base-level component to expose the functionality of the absorbed class. Invocations are mutable in the sense that they can be added and removed from existing components at runtime using meta-level transmutations. However, the relationship between invocations on the base-level component and methods on the base-level class need not be one-to-one. Indeed, when a component is added to an adaptive system it may be necessary to modify the component's interface such that it fits properly into the system structure. Since component interfaces are mutable and composed of primitive operations, augmentation of an existing interface is possible. Thus, a component can be adapted to achieve a subjective fit. However, some of the base-level methods may be occluded or even combined under a single invocation as the system's form is modified. For example, a read-only socket component can be constructed by occluding the methods

that enable writing.

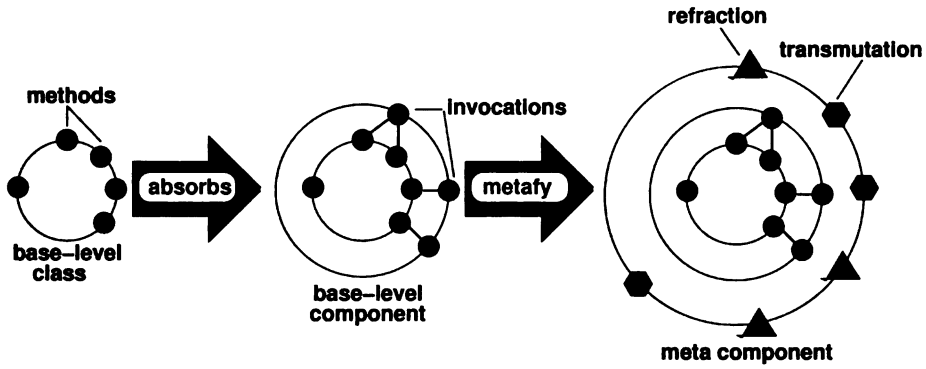


Figure 3.4: Component absorption and metafication

3.2.4 Metafication

Metafication is used to create refractions and transmutations that operate on the base component, as shown in Figure 3.4. Refractions and transmutations embody limited adaptive logic and are intended for defining how the base level can be inspected and changed. The logic for how and why these tool-like operations should be used is provided at other component levels or by other components entirely. That is, a component may refract and transmute itself or a component can be refracted and transmuted by another. For instance, an execution thread providing the imperative functionality of the system might be defined as $S = A[B]$ where A is a meta-component reflecting B. Another execution thread, R, could adapt S using the refractions and transmutations defined by A to recompose $S = A[C]$.

3.3 A Prototype Language: Adaptive Java

In order to gain a better understanding of how the addition of refractive and transmutative elements to a language would affect its use and structure, we defined and implemented a prototype language, Adaptive Java, as an extension to Java.

In this study, we used CUP [131], a parser generator for Java, to implement Adaptive Java. CUP takes our grammar productions for the Adaptive Java extensions and generates an LALR parser, called ajc, which converts Adaptive Java code into Java. Semantic routines were added to this parser such that the generated Java code could then be compiled using a standard Java compiler.

Adaptive Java is a prototype whose purpose is to improve our understanding of which language constructs and mechanisms are desirable in dynamic and adaptive languages. Several of the concepts resulting from this investigation have been incorporated into other tools developed by our group [132,133]. In this section, we provide some examples of the language constructs used to code refractive and transmutative software in Adaptive Java.

3.3.1 Basic Component Structure

Figure 3.5 shows the language structure of a typical Adaptive Java component. Most Java statements are supported within invocation and constructor blocks. Constructors in Adaptive Java are essentially identical to Java constructors and are immutable, only being used to provide flexibility in the initial instantiation of a component. Standard Java methods are replaced by invocations and standard immutable variable dec-

larations are supplemented with mutable variable declarations. Mutable variables can be added and removed from components, using transmutations at the meta-level, in much the same way as can invocations.

```
// A simple component
component BasicComponent {
  // Constructor
  public BasicComponent() { ... }

  // Invocation
  public invocation void method1(String arg) { ... }

      .
      .
      .
}
```

Figure 3.5: Adaptive Java component structure.

Optionally, a component can be declared to extend another component. Extending a component encapsulates the extended component within the newly declared component. The extended component is called the *inner component* whereas the extending component is called the *outer component.* Inheritance is simulated by examining the outer component's invocations for the desired invocation. If the invocation is not found then a recursive search is performed of encapsulated components. For instance, let $S = A[B[C]]$ be a system composed by extending component C with B and then extending B with A. If the execution of invocation `A.exec()` is requested, first component A then B and finally C will be searched for `exec()`. The first instance of `exec()` found will be executed, thus allowing inner invocations to be overridden by

those found at more outer encapsulation levels. It is worth noting, that simulating inheritance in this way allows the inheritance chain to be decomposed and recomposed with different components, possibly modifying the internal processing of component composition.

3.3.2 Absorbing Existing Classes

The **absorbs** keyword is used to construct a component from a regular Java class. Figure 3.6 shows the Adaptive Java code for absorbing a Java socket class into a socket component that is used only for receiving packets. Invocations are created to expose selected functionality of the absorbed class, in this case only the receive and close methods. The absorbed class is accessed by the absorbing component through the **base** keyword. The other methods of the base class are hidden at this level.

```
// receive-only socket component
public component RecvSocket absorbs Socket {
  // constructor
  public RecvSocket(int port, String group, byte ttl)
      throws UnknownHostException, IOException {
    setBase(new Socket(port, group, ttl));
  }

  public invocation void receive(DatagramPacket p)
      throws IOException {
    base.receive(p);
  }

  public invocation void close()
      throws IOException {
    base.close();
  }
}
```

Figure 3.6: Absorbing a class into a Component

3.3.3 Reifying a Meta-level

Meta-components encapsulate other components and support only reflective functionality. The encapsulated component is the meta-component's base level. Meta components are declared using the `metafy` keyword. Figure 3.7 shows an example in which we metafy the RecvComponent component defined in Figure 3.6.

```
// Meta receive-only socket component
public component MetaRecvSocket metafy RecvSocket {
  // Constructor
  public MetaRecvComponent(int port, String group, byte ttl)
      throws UnknownHostException, IOException {
    setBase(new RecvSocket(port, group, ttl));
  }

  // Transmutation that sets the data stream compression level.
  public transmutation void SetCompression(int level) {
      .
      .
      .
  }

  // Refraction that returns the observed bytes transfered
  // by the RecvSocket component.
  public refraction long GetBytesXmit() {
      .
      .
      .
    return bytes_transfered;
  }
}
```

Figure 3.7: Metafying a component

An invocation is called using the `invoke` keyword (e.g., `invoke rSock.receive(pckt)`). This causes the retrieval of a matching invocation object from a hash table that is then cast to the appropriate invocation type. This effects a prototypical indirect binding for invocations. Indirection enables support for adapting component interfaces and supporting subjective MOPS.

3.4 Case Study: MetaSockets

In order to evaluate the design of the Adaptive Java language constructs, our group used Adaptive Java to develop a component called a "metamorphic" socket, or simply, metasocket. In response to external events, an application or middleware platform can use refractions and transmutations on this component to observe and modify socket functionality. In this study, metasockets are used to enhance the quality of wireless audio channels at run time. Figure 3.8 shows the configuration where live audio is streamed from a workstation to multiple iPAQ handheld computers running Windows CE. The audio stream is transmitted on a 100 Mbps Ethernet LAN to a wireless access point, where it is multicast at 11 Mbps on an 802.11b wireless LAN.

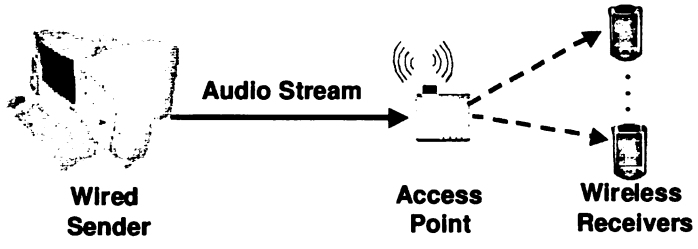


Figure 3.8: Physical experimental configuration.

3.4.1 Block-Erasure Codes

The characteristics of wireless LANs are very different from those of their wired counterparts. Factors such as signal strength, interference, and antennae alignment produce dynamic and location-dependent packet loss [134]. Forward error correction (FEC) can be used to improve reliability by introducing redundancy into the data channel, enabling a receiver to correct some losses without contacting the sender for retransmission. The FEC method used in this study addresses *erasures* of packets resulting from CRC-based detection of errors at the data link layer. As shown in Figure 3.9, an (n, k) *block erasure code* [135, 136] converts k source packets into n encoded packets, such that any k of the n encoded packets can be used to reconstruct the k source packets. These codes have gained popularity recently due to an efficient implementation by Rizzo [136]. Each set of n encoded packets is referred to as a *group*. Here we use only *systematic* (n, k) codes, meaning that the first k packets in a group are identical to the original k data packets. The remaining $n - k$ packets are referred to as *parity* packets.

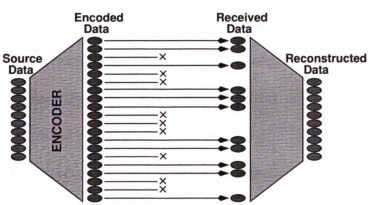


Figure 3.9: Operation of FEC based on block erasure codes.

3.4.2 MetaSocket Design and Operation.

An audio streaming application was implemented in Adaptive Java. The application comprises two main parts. The Recorder uses the Java Sound API to read audio data from a workstation's microphone and multicast it on the network. The Player receives the audio data and plays it using the Java Sound API. Both applications were written in Adaptive Java and converted into pure Java using the ajc parser. They communicate using MetaSockets instead of regular Java sockets.

Figure 3.10 depicts the structure of a `MetaSocket` component. The base component, called **SendSocket**, was created by absorbing the existing Java `Socket` class. Certain public members and methods are made accessible through invocations on **SendSocket**. Since this component is intended to be used only for sending data, the invocations available to other components are `send()` and `close()`. Hence, the application code using the computational interface of a metamorphic socket looks similar to code that uses a regular socket. In addition, three invocations (`SetBuffer()`, `GetFilter()`, `GetLastFilter()`) are intended for use by the meta-level. The `SendSocket` was metafied to create a meta-level component called **MetaSocket**. `GetStatus()` is a refraction that is used to obtain the current configuration of filters. `InsertFilter()` and `RemoveFilter()` are transmutations that are used to modify the filter pipeline.

A separate utility thread resides within each application and can be used to control the behavior of its respective MetaSocket based on current observable status. For purposes of testing the MetaSocket interfaces, an interactive administration utility

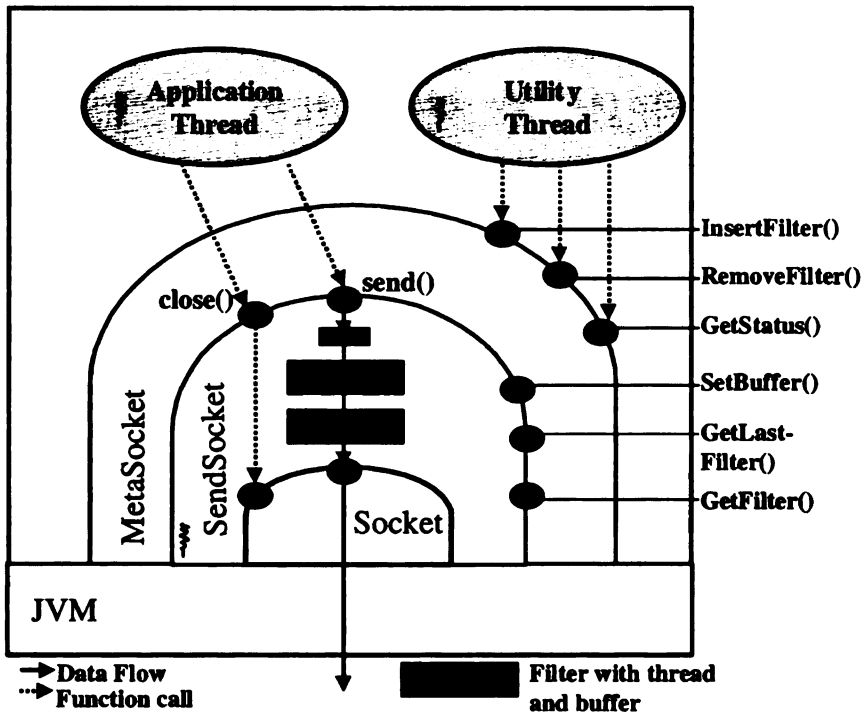


Figure 3.10: Structure of a MetaSocket.

was developed that enables the manipulation of MetaSockets directly. Figure 3.11 shows an example trace where audio was streamed from a desktop to an iPAQ across an 802.11 wireless LAN. The transmutative interface was used to insert an $(8,4)$ forward error correction filter dynamically, significantly reducing packet loss, until the filter was removed.

3.5 Mechanisms Enabling Operator Adaptation

Our implementation of Adaptive Java and our subsequent investigation with the MetaSocket improved our understanding on how to design and build adaptable components. Separation of introspection and intercession enables a software developer to focus on each of these concerns separately and address the design of each in an or-

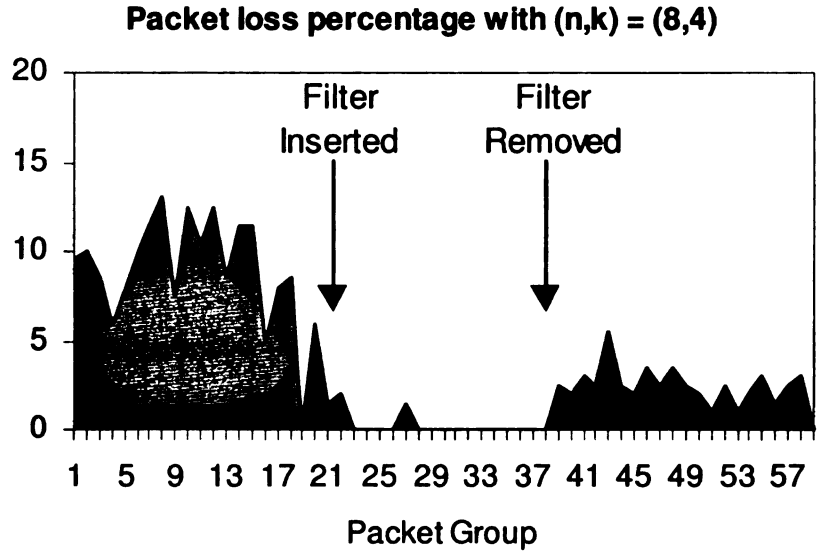


Figure 3.11: Sample results of dynamically changing MetaSocket configuration.

thogonal fashion, easing the burden of designing adaptive software. However, not all approaches to software adaptation focus on language support or component reconfiguration. In distributed, data-streaming systems, adaptation is possible if operators can be introduced into the data stream transparently or gracefully redeployed to different hosts. In this section we introduce the design and implementation of Dynamic River, an extension to the DaSH data acquisition system [137]. Dynamic River comprises operators designed for processing sensor data streams and enables sets of operators to be dynamically relocated to more suitable hosts to better meet quality-of-service requirements.

3.5.1 Dynamic River Operators and Segments

A Dynamic River *pipeline* is defined as a sequential set of operations composed between a data source and its final sink (destination). Operations can transform, filter or perform other processing on pipelined data. Shown in Figure 3.12 is a high-level diagram of the structure of a *basic operator* and two *network operators* (**streamin** and **streamout**). This pipeline can be represented using the notation ⇒**[streamin|operator|streamout]**. A basic Dynamic River *operator* is a system-level process (program) that reads data records from **stdin** and writes records to **stdout**. An operator can process a record, for instance by converting a record comprising a vector of floating pointing values to its complex representation, and emit the results as a new record. The network operator **streamin** reads records from a network socket and writes them to **stdout**. Correspondingly, **streamout** reads records from **stdin** and writes them to a network socket. The network operators enable record processing to be distributed across the processor and memory resources of many hosts. Thus, processor or memory intensive operations can be strategically located on hosts within a compute cluster or across a network to improve overall throughput. Pipeline *segments* are created by composing sequences of operators that produce a partial result important to the overall pipeline application. For instance, a pipeline constructed to compute the Fourier transform might include a segment comprising an operator for converting floating point values to complex numbers, followed by a Fourier transform operator for conversion to frequency data. Segments can receive records using **streamin** and emit records using **streamout**, enabling instantiation of segments and

the construction of a pipeline across networked hosts. Moreover, network operators enable dynamic, runtime pipeline recomposition by enabling a pipeline segment to be moved to a different host.

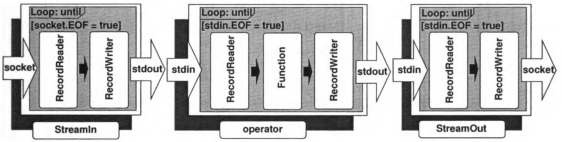


Figure 3.12: Basic internal structure of basic stream operators and the `streamin` and `streamout` network operators.

Figure 3.13 is an elided sequence diagram for the normal operation of a pipeline. The `source` and `sink` are abstract in that they represent an application specific data source and final destination. Moreover, other operators (elided) may be inserted between the `source` and `streamout` or between `streamin` and the `sink`. During normal operation each pipeline operator loops, reading and writing records until the connection between `streamout` and `streamin` closes. Connection closure may occur due to graceful shutdown of a segment, or if a segment unexpectedly terminates due to a program fault. In addition to what is depicted, the entire pipeline can be gracefully shutdown from the `source` to the `sink`, by flushing the pipeline sequentially, if the `source` should reach end-of-file (EOF).

Currently, Dynamic River provides 60 operators for processing or routing data and more operators are being implemented on a regular basis to meet the needs of different applications. Table 3.1 describes a subset of basic operators, 2 network operators and 2 support programs discussed in this dissertation. A more detailed description of these

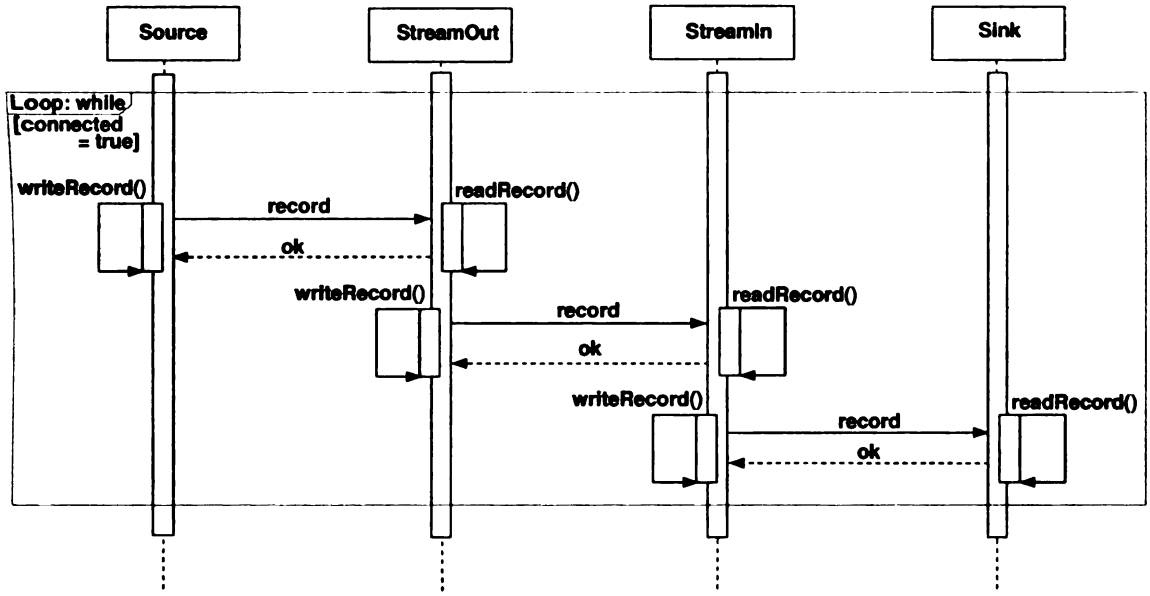


Figure 3.13: An elided sequence diagram depicting the normal operation of the `streamout` and `streamin` pipeline operators.

and other operators and supporting programs is provided in Appendix B. Notably, the `dynriverd` daemon enables remote reconfiguration and redeployment of Dynamic River operators. This daemon can issue control messages, over a named FIFO, to local `streamin` and `streamout` operators, triggering graceful shutdown of pipeline segments. Moreover, `dynriverd` can instantiate new pipeline segments and connect them to segments executing on remote hosts.

3.5.2 Dynamic River Records

As shown in Figure 3.14, a Dynamic River record comprises a fixed length *record header* and a variable length field that comprises the application record data. Record fields are defined in Table 3.2 and provide meta information needed for record storage, transmission and processing in a distributed environment. Moreover, header fields are

Table 3.1: Description of a subset of Dynamic River operators and support programs.

Basic Stream Operators	
Operator	**Description**
`cabs`	Convert an input record of complex floating point values to a record of comprising the complex absolute values of the input record. The output record comprises floating point values.
`cutout`	Convert an input record of floating point values by selecting a specific range of values and discarding the remainder.
`dft`	Convert an input record of complex floating point values by computing the discrete Fourier transform.
`float2cplx`	Convert an input record of floating point values to a complex number representation. Specifically, the real part of the imaginary number contains the original floating point value and the imaginary part is set to 0.
`wav2rec`	Convert WAV format acoustic data into Dynamic River records.
`welchwindow`	Convert an input record by filtering it with a Welch window [138].
`readout`	Acquire sensor readings from either real or synthetic sensors and emit records comprising these readings. This operator is abstract and has different implementations depending on application requirements.
Network Stream Operators	
Operator	**Description**
`streamin`	Accept or initiate network connections to/from `streamout`. Reads records over the network connection and emits these records unaltered to standard output.
`streamout`	Accept or initiate network connections to/from `streamin`. Reads records from standard input and emits these records unaltered on the network connection.
Support Programs	
Program	**Description**
`ctrlcmd`	A command line executable for issuing commands to `dynriverd`. Commands can invoke pipeline segments, terminate segment execution and make status inquiries.
`dynriverd`	A daemon for remote control of pipeline segments. `Dynriverd` can invoke pipeline segments, terminate segment execution and make status inquires. `Dynriverd` is controlled using `ctrlcmd`.

automatically converted to network byte order (using `htonl()` and `htons()`) when written and converted to host byte order when read.

```
typedef struct record_header_struct {
  uint16_t version;
  uint32_t record_size;
  uint32_t record_type;
  uint32_t record_subtype;
  uint16_t status_code;
  uint32_t byte_order;
  uint32_t data_size;
  uint32_t entity_count;
  uint32_t scope_type;
  uint32_t scope;
  // ----------------------------
  // The data field follows the
  // static sized header.  The data
  // field has byte length
  // data_size and comprises the
  // application record data.
  //
  // ubyte *data;
} record_header_t;
```

Figure 3.14: C/C++ Definition of the Dynamic River record header.

As depicted in Figure 3.15, records can be grouped using the **record_subtype**, **scope** and **scope_type** fields. We define a *data stream scope* as a sequence of records that share some contextual meaning, such as having been produced from the same acoustic clip. Within the data stream, each scope begins with an **OpenScope** record and ends with a **CloseScope** record. Scope opens and closes are indicated using the record header **record_subtype** field. The **record_type** for scope records is identical to that of the data records within the scope so that data records and their associated scope information are kept together within the data stream. Optionally, **CloseScope** records can be replaced with **BadCloseScope** records to enable scope closure while indicating that the scope has not reached its intended point of closure. For instance, if an upstream segment terminates unexpectedly and leaves one or more scopes open, the **streamin** operator will generate **BadCloseScope** records to close all open scopes,

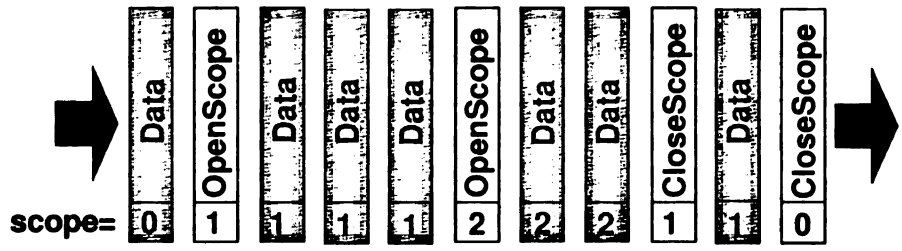


Figure 3.15: Depiction of a record stream with data scoping. Numbers indicate the scope nesting depth indicated by the record header scope field.

Scopes can also be nested. The `scope` field indicates the current scope nesting depth, larger values indicate greater nesting while scope depth 0 indicates the outermost scope. The outer most scope is defined as the scope that represents the data stream as a simple stream of records without any further contextual grouping. Scopes at greater depths indicate an application specific grouping that should be maintained during record transmission and processing. As shown in Figure 3.15, the `scope` field for `OpenScope` records indicates the depth of the newly opened scope. Conversely, the `scope` field on `CloseScope` records indicates the scope depth that will be reentered after scope closure. The `scope_type` field enables the specification of an application specific scope type. For instance, a scope can be identified as comprising an acoustic clip. Optionally, `OpenScope` records may contain context information. As we shall see in Chapter 7, this functionality is useful for recording the sampling rate of an acoustic clip, needed for computing the frequency range depicted by a spectrogram.

3.6 Related Work

Several projects have explored the use of program language constructs to support runtime adaptation. For instance, the Program Control Language (PCL) [90] provides

Table 3.2: Description of record header fields.

Field	Description
`version`	The current record header version. Used for detecting incompatibilities or translating between different record versions.
`record_size`	The entire size of this record in bytes. This size includes the size of this record header and data.
`record_type`	The record type. For instance, may be assigned a well-known named type such as `PktDoc` or `Data`, or a user defined type.
`record_subtype`	The record subtype. For instance, may be assigned a well-known named type such as `OpenScope` or `CloseScope`, or a user defined type.
`status_code`	The current status of this record. For instance, indicating if the record is complete or truncated.
`byte_order`	The four bytes: `0x01020304`. Used for determining the byte order when switching between different architectures.
`data_size`	The size of the `data` field in bytes. This field is present largely to ease application level record processing.
`entity_count`	Set to indicate the number of *entities* included in the `data` field. This field is set by the application programmer. Typically, this field will indicate the number of events, floats, integers or other data types comprising the `data` field.
`scope_type`	The current data scope type.
`scope`	The data scope nesting depth.
`data`	Application specific data. This field immediately follows the record header and is not explicitly defined by `record_header_t`.

programming language support for computational adaptation. An adaptive program is specified using adaptors and targets for the adaptors. Adaptors are typically subclasses of targets and have access to many of the variables defined in the target class. Adaptors can change the behavior of a single target class through the use of variables and methods, called `ControlParameters` and `ControlMethods`, visible in the superclass to the Adaptor. The Adaptor can read and modify `ControlParameters` and call `ControlMethods` as part of the adaptation policy, enabling parameter obser-

vation and modification. Adaptation policies are specified using metrics associated with variables of the target class. Metrics can be sampled, timed or rate based, and are used to trigger events that execute adaptive processing. The PCL project demonstrates how an adaptive system is constructed, providing domain specific constructs layered atop C++. However, PCL does not enable recomposition of a running system, but rather enables the modification of parameters using existing strategies.

Other projects have explored the use of computational reflection. DAS [3] defines a software model supporting dynamic application adaptability, and LEAD++ is an object-oriented reflective language that uses DAS. As shown in Figure 3.16, DAS describes an application using a component graph, where the nodes are components and the edges are message paths. LEAD++ aims to provide program language support for effective design and implementation of adaptive software. Runtime adaptation is realized in several ways. Plug-and-play components can be dynamically instantiated and inserted into the component-graph. Messages can be redirected along different component-graph edges, effectively changing application behavior through invocation of different methods. The component-graph structure can be changed by adding or replacing components and message paths. The DAS system is divided into a base-level and a meta-level. Messages initiated by base-level components are intercepted by dispatchers in the meta-level and directed to the appropriate base-level component as determined by the current environmental context. However, unlike Adaptive Java, DAS and LEAD++ do not address the separation of introspection and intercession or the role of encapsulation for enabling OOL design and development of dynamically recomposable software.

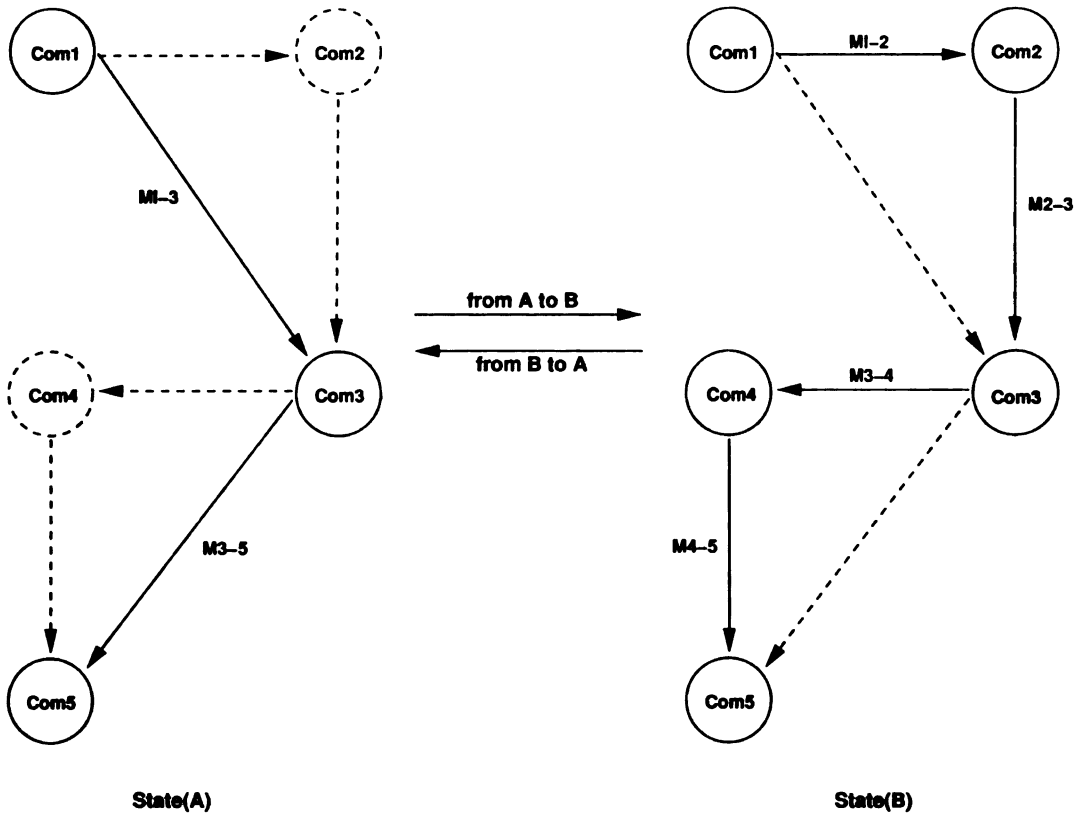


Figure 3.16: The construction and alteration of a component-graph [3]. Solid nodes are active, while dashed nodes are inactive slots for storing components. The inactive slots on the left are filled on the right. Messages are redirected through the newly activated nodes.

The TAILOR project [139] studies unanticipated software change by studying extensions to programming languages and runtime systems. GILGUL [94] is a Java extension language providing a mechanism and type system for handling the update of references during dynamic object replacement. Although sharing a language-based approach with LEAD++ and Adaptive Java, GILGUL does not provide a reflective system supporting adaptive applications. Instead, GILGUL focuses on designing special language features that ease runtime, application restructuring. GILGUL implements objects such that they have a *comparand* and a *referent.* Comparands can be used to compare objects such that equivalence of two objects can be determined

separately from comparing their references. That is, equivalence is not tied directly to object identity. Referents enable the dynamic replacement of objects by allowing multiple objects to hold referents that all indirectly point to a particular object reference. As such, updating a particular referent updates all the other referents that point to the same object reference. Thus, objects can be dynamically replaced through the assignment of references and determination of whether an assignment can be made through comparison of comparands. GILGUL also defines implementation-only classes that cannot be used as types. Variables cannot have the type of an implementation-only class. However, implementation-only objects can be assigned to variables that are typed as a superclass of the implementation-only class. Implementation-only classes help extend the type system such that additive and subtractive replacement of objects is less restrictive.

In this chapter, we studied a possible approach, based on separation of introspection and intercession, for designing reflective primitives. We developed a prototype language, Adaptive Java, and showed how it can be used to construct adaptive components from existing classes. We intend these low-level mechanisms to provide a foundation for the construction and maintenance of meta-object protocols for cross-cutting concerns. Moreover, we introduced Dynamic River, a system that focuses on adaptation of data stream processing. Dynamic River extends our study of adaptive mechanisms to distributed operator reconfiguration and redeployment.

A number of research projects have addressed the construction of distributed stream processing engines that provide quality-of-service optimization or guarantees. Wavescope [140] addresses the need for data streaming systems that combine sig-

nal processing and event-stream processing for sensor data. Wavescope provides a scripting language, called WaveScript, to simplify the implementation of user defined processing and data stream queries and is designed to address resource limitations imposed by sensor platforms or embedded systems. Unlike Dynamic River, Wavescope does not address runtime reconfiguration and redeployment of operators. Aurora [141,142] and Borealis [143], that inherits Aurora's stream processing abilities and Medusa's [144] distributed functionality, focus on database-like query optimization for data streams. The Aurora/Borealis approach optimizes query processing in a piecewise fashion that "drains" a subset of the operators prior to runtime reconfiguration. As such, query processing and dependent data tables can be collocated or queries can be reordered. Similarly, CANS [109, 110] Enables data-stream applications to be composed using operators, called drivers, that contain only soft state that does not need to be retained. CANS supports data path reconfiguration involving insertion and deletion of drivers. The Aurora/Borealis approach to query optimization and CANS reconfiguration of drivers is similar to Dynamic River's method for operator redeployment and reconfiguration.

3.7 Discussion

In this chapter, we studied mechanisms that enable runtime recomposition. However, building autonomous software requires more than the design and implementation of adaptive mechanisms. The exchange of components in a running system requires consideration of component state. For instance, components often encapsulate data

important to proper program execution. In the face of component exchange, such non-transient data must be maintained. Moreover, distributed operator redeployment and reconfiguration requires state maintenance to avoid loss or corruption of data. As we will describe in Chapter 4, our work with Dynamic River complements other projects by directly addressing state maintenance for distributed pipelines while considering graceful recomposition and fault resilience. Moreover, our design and implementation demonstrates our principled, integrated approach to designing and implementing dynamically adaptive software.

Chapter 4

State Maintenance for Autonomic Software

State maintenance comprises the protocols and data transformations required to avoid loss of nontransient state and ensure proper program execution during runtime program recomposition. Where mechanisms supporting dynamic recomposition address the need for principled approaches for decoupling application components and data structures, state maintenance addresses the need for principled approaches for dynamically transitioning an application from one composition to another. As introduced in Section 2.4, state maintenance comprises three major concerns: reference update, state migration and synchronizing intercession. The most common solution for the reference update problem is the use of indirection, where application objects are not referenced directly, but rather through secondary data structures or components. However, while indirection is relatively well-understood, state migration and synchronization of intercession are more complex problems that are less understood.

Dynamic recomposition involves state transfer as it relates to collateral change, state transformation and recomposition protocols. In the next section, we will begin by describing state transformation as it relates to runtime recomposition.

The remainder of this chapter is organized as follows. Section 4.1 discusses non-transient state, component equivalence and collateral change as related to state migration and synchronization. Section 4.2 formalizes state transformation to help clarify how state transformation can enable and limit the flexibility of a recomposable system. Section 4.3 describes Perimorph, an API that enables the capture and migration of state between components during recomposition, and discusses our approach for providing programmer support for principled implementation of state maintenance in adaptive software. An example application and a case study, both implemented using Perimorph, are described in Sections 4.4 and 4.5. State maintenance in Dynamic River is described in Section 4.6. Section 4.7 discusses prior work addressing state maintenance and capture. Finally, Section 4.8 concludes this chapter.

4.1 Key Concepts and Issues

In this section, we introduce key concepts and issues that relate to state maintenance for autonomic software. The issues discussed are general, however, the design and implementation of approaches that address these issues are often specific to the application domain or adaptation goals, such as component upgrade or quality-of-service improvement. Later, in Sections 4.3 and 4.6 we describe implementations that address these issues using an API-based approach in Perimorph, and an approach for

distributed data stream processing in Dynamic River.

4.1.1 Nontransient state

Nontransient state refers to program data that must not be lost during dynamic recomposition if a program is to continue functioning correctly. At the component level, one solution is to enable state extraction to export a *normalized representation* of the component's state, understood by all other components of the same abstract type (i.e. implements a queue). The normalized state can then be assigned to an algorithmically or structurally dissimilar component. A component needs to know only how to code a normalized memento of its own state and how to decode a normalized state memento captured from another component. The memento pattern [145] enables the state of an object to be captured and extracted and later injected back into the same object without violating encapsulation. For example, by using the memento pattern in conjunction with normalization, an array-based queue can be assigned to a vector-based queue.

Transformation of state can be avoided if a component can be quiesced such that it does not contain any nontransient state. For instance, the SwitchWare project [104] uses an active network bridging architecture [65] that allows reprogramming of a network bridge as it executes. Loadable modules, called switchlets, are used to recompose a network protocol stack such that an improved, more secure or repaired version of a protocol can be instantiated dynamically. When a switchlet is upgraded, the old implementation remains until all routers have received the update. In this way, con-

nections using the old protocol can still be serviced during the upgrade. Since the old protocol remains until it is no longer needed, no transformation or transfer of component state is needed. However, transformation of state cannot always be avoided in this manner. For instance, components that collect aggregate statistics about the running system may never be quiesced. Moreover, it may not be acceptable to continue to use an old component when upgrading a component to address security or time-sensitive issues.

4.1.2 Component equivalence

During static composition, an application composer need only consider interface and component function when selecting one plug-compatible component over another. During dynamic composition, on the other hand, the composer must also consider the migration of state from an active component to its replacement. The set of components that can be exchanged with an active component form an equivalence class based on interface compatibility and whether the state of an active component can be transformed into the state of a potential replacement. That is, during dynamic composition, two components can be considered equivalent if they share the same interface and the state of the active component can be transformed and transferred to the replacement. This equivalence class represents a set of candidate component replacements. However, even when component `A` can be replaced with component `B`, this relation does not imply that component `B` can be replaced with component `A`. In other words, the equivalence class for active component `A` may not be the same as that

for active component B, although both components may be plug-compatible during static composition. For example, replacement of a dynamically resizeable queue with one that has a fixed size may be impossible if the entire contents of the resizeable queue cannot be contained by the fixed length implementation.

4.1.3 Collateral change

We have discussed state maintenance as it applies to the exchange of a single component. However, dynamically recomposing software may require implementing more than one exchange to achieve the desired system behavior. We define *collateral change* [44] as the set of recompositions that must be applied to an application atomically, or through an orchestrated set of steps, for continued correct execution. For instance, to insert FEC into a data stream, the sender must first insert an encoder and only after reception of the first encoded packet should the receiver insert a decoder. Of course, the dependencies between components or fragments can be more complex, possibly resulting in a cascade of replacements. Capturing these dependencies enables dynamic recomposition by recognizing the set of components that must be replaced to ensure continued correct program function. Moreover, realizing when a recomposition will cause a cascade of component replacements helps identify potentially costly recompositions that may fail to meet quality-of-service requirements.

Understanding state maintenance is key to designing and implementing dynamically recomposable software. Both the correct operation and system flexibility are impacted significantly by the selected design and implementation chosen for an au-

tonomic system. If a decision maker adapts a system using components that cannot later be easily replaced in the face of changing environmental conditions or user requirements, then the system may become rigid and unable to adapt. As such, state maintenance is a "forward-looking" concern in that changes made in the past may preclude future system polymorphs. Moreover, state maintenance is a holistic concern that must address the effects of adaptive actions on the entire application, not only the replacement of individual components but also the collateral and possibly cascading replacement of components or fragments. Good designs and implementations enable continued system flexibility and ensure correct application functionality while enabling component configurations to converge, meeting the requirements of specific environments or users.

4.2 State Transformation

Making good design decisions for dynamically recomposable software requires understanding the association between the flexibility of the system and state transfer and transformation. This association can be clarified using basic mathematical relations. At the component level, we consider state maintenance as comprising a relation, R, that transfers and transforms the state of one component to meet the form required by its replacement. Below we provide three definitions that describe this relationship.

Definition 5 *Given the set of possible states, S_A, of old component A, we define relation R as* **reflexive** *iff: aRa for all a in S_A.*

That is, state transfer between component A and its replacement requires no state transformation. Such relations are found during upgrade of stateless components and in applications where components can be completely quiesced such that no nontransient state remains.

Definition 6 *Given the set of possible states of an old component A, S_A, and the possible set of states of new component B, S_B, we define relation R as* **symmetric** *iff: aRb implies bRa for a in S_A and all b in S_B.*

Notice that if R is reflexive it is also symmetric under the null transformation. The symmetric relation is often found in component upgrades, where the same operation is computed in the new component, but in an optimized fashion. For example, a component can be replaced by one requiring less memory. The symmetric relation implies that a component can be downgraded, where a component is replaced with an earlier implementation. The symmetric relation could also be read as: "R is invertible."

Definition 7 *Given components A, B and C, with state sets S_A, S_B and S_C we define R as* **transitive** *iff: aRb and bRc implies aRc for all a, b and c in S_A, S_B and S_C respectively.*

From a pragmatic perspective, a reflexive, symmetric and transitive relation is rare. Even in data streaming applications, where components can often be quiesced (by halting the input stream and flushing the component), some components may still retain nontransient state. For example, a component that generates a histogram of the

data in the stream can be replaced by a component that computes an arithmetic mean. In this case, the histogram data can be transformed into the mean by summing the histogram bin values and counts and dividing the sum by the total count. However, this transformation is neither symmetric nor transitive. In fact, adaptive systems that require continued state transformations require consideration not only of a set of potential components, but reduction to a set of components that address the required adaptation with respect to the allowed or possible state transformations.

4.3 Perimorph Design and Implementation

In this section we describe the architecture and operation of Perimorph, an API designed to facilitate state transfer in adaptive programs. Perimorph is implemented using Java and enables an application designer to quantify and codify collateral changes, as related to compositional adaptation, in terms of factor sets. Perimorph uses repositories, called *stores*, to provide a well known structure and interface for manipulating and recomposing an application. Moreover, Perimorph stores provide a meta-level view of the base-level application composition while supporting runtime recomposition. The main contribution of this study is a better understanding of how to codify collateral change and to introduce a principled approach for integrating state maintenance into the design and implementation of adaptive software. A data dictionary for Perimorph major components can be found in Appendix A.

4.3.1 Component construction

Figure 4.1 shows the relationship of a component, *factor sets* and *factors*. Factors represent modifications that can be applied to component operations. Each set of collateral changes can be codified as a factor set that contains factors and nontransient data structures shared between the factors. Components are identified by a name (a Java String) given to them when they are created. Interface sets are added to components and contain operation signatures defining the interfaces implemented by a component. For instance, the adaptive queue has an interface set consisting of the signatures `put(Item)`, `get()` and `isFull()`. Operations comprise an interface signature and zero or more factors. Factors are attached to an interface signature forming the body of an operation.

Factors may be attached as either *pre* or *post* factors. Pre-factors are executed before a *return* and post-factors are executed following a *return*. Pre-factors implement the operation body, while post-factors provide post operation processing. Any pre-factor can trigger a return, preventing the execution of subsequent pre-factors and jumping to post-factor processing. Similarly, any post-factor can trigger completion of an operation. Post-factors allow the completion of functions begun by pre-factors. For instance, a pre-factor may lock a mutex to control concurrent access to a component. A post-factor could unlock the mutex, ensuring that other threads are allowed continued access.

Data structures defined within factor sets represent nontransient state. When factors from one factor set are replaced by another, nontransient state needs to be ex-

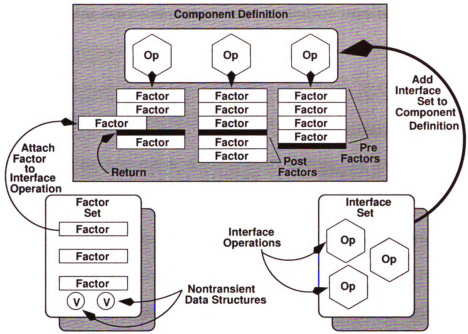


Figure 4.1: Relationship of factors, factor sets and a component definition.

tracted from the old set and injected into the new. The transfer of state is completed using `getState()` and `setState()` factor set methods that extract and inject a normalized state memento. Factor set data structures are shared by all factors belonging to the same factor set.

Aspect Oriented Programming (AOP) aspects [57, 58] and factor sets are related in that an aspect may comprise one or more factor sets. However, rather than focus on designing a system in terms of crosscutting concerns and disentangled code, as in AOP, factors together with factor sets provide constructs that enable a programmer to codify adaptations that must happen collaterally and declare variables that contain nontransient state. In other words, sets of collateral changes represent the *factoring* of an application such that recomposition is defined in terms of viable sets of modifications. All factors that are members of the same factor set must be applied

atomically. Applying nonviable changes to an application usually results in program failure.

4.3.2 References and invocations

Proxies represent components in the base-level, allowing the application to invoke component operations while decoupling components and providing the base-level with a consistent view of the program's structure. Proxies are used in place of base-level component references. For example, if a producer and consumer thread communicate using an array-queue both threads hold a proxy, instead of a reference, for the queue component. When a control thread recomposes the array-queue as a vector-queue, it is unnecessary to update these proxies, since the next invocation of a `put()`, `get()` or `isFull()` operation, will retrieve the vector-queue, instead of the array-queue, from the `ComponentStore`.

Execution of a component operation is depicted in Figure 4.2. An application invokes a component operation by calling a proxy's `invoke()` method and specifying an operation signature, such as `put(Item)`, as a parameter. Using the component's name, the required component is located in the `ComponentStore`, and the specified operation is retrieved from the component's interface set. Factors, previously attached to the operation signature, are invoked one after the other until operation execution is complete. Finally, control is returned to the base-level caller.

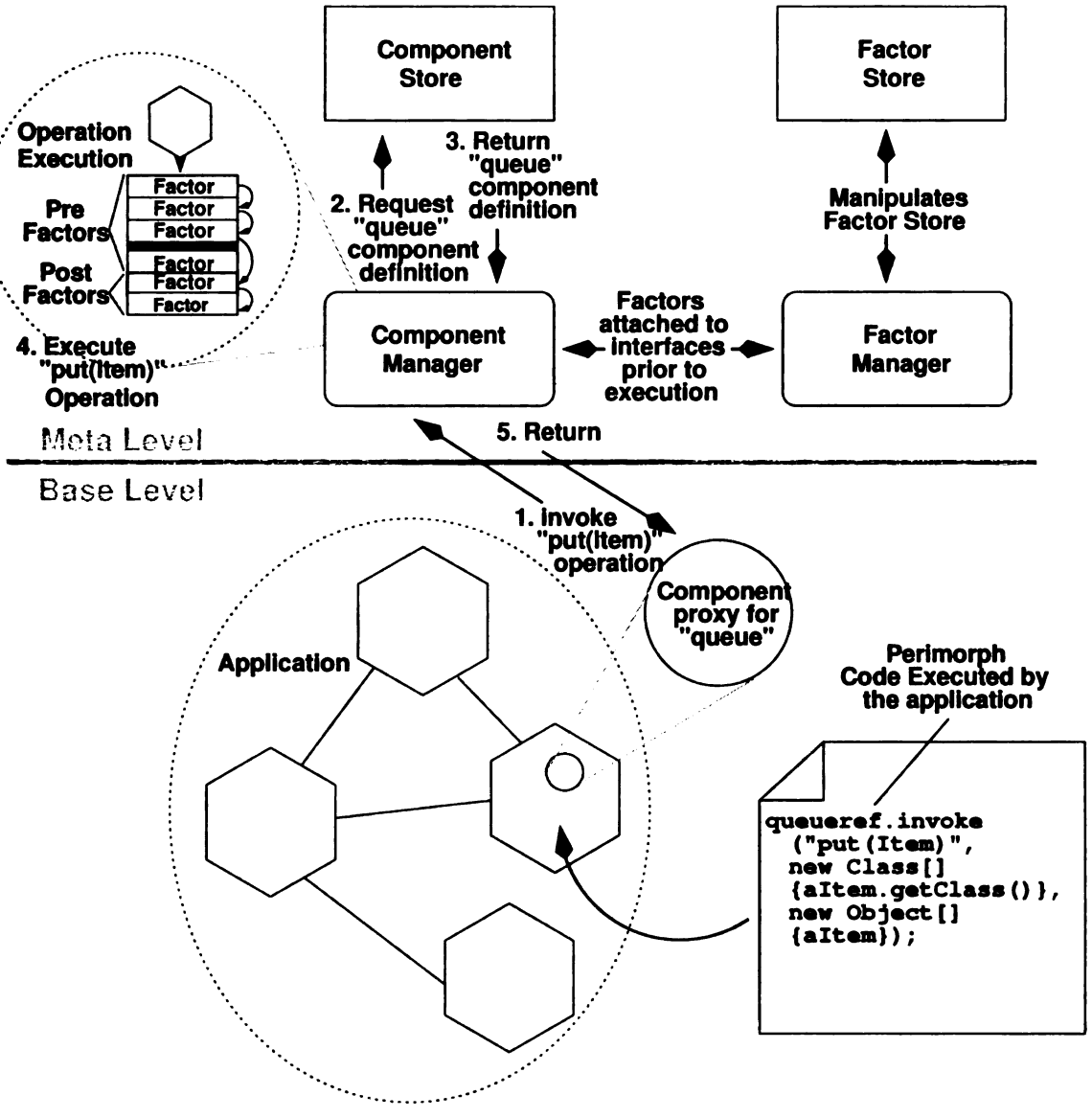


Figure 4.2: Executing a component operation.

4.3.3 Recomposition

As shown in Figure 4.3, recomposing a component involves adding, deleting or replacing factors. Both functional and nonfunctional factors can be added, removed or replaced, allowing the entire function of a component to be changed or augmented. For instance, an array-based queue can be replaced with a vector-based queue. Nonfunctional concerns, such as concurrency controls or security, can be added and re-

moved as needed. Reference update is automatic as recomposition operates on the component definition, leaving all component proxies alone. Separating the definition of a component from the references to it obviates the need to update object references scattered throughout the code, simplifying recomposition significantly.

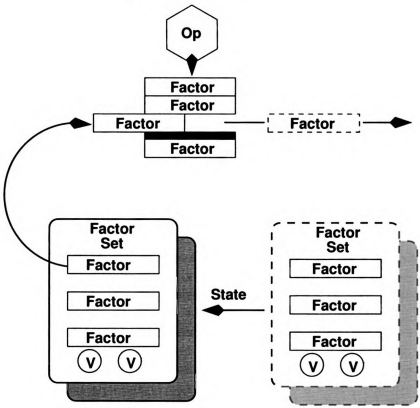


Figure 4.3: Recomposition of a component where a factor is replaced by a new one from a different factor set. Nontransient state is assigned to the replacement factor set from the old.

4.3.4 Activation and deactivation

Factor sets can be activated or deactivated as they are put into or removed from use. Activation and deactivation automates the process of initialization and shutdown of factor sets, such as those defining graphical interfaces or using threads. Reference

counts are kept for all factor sets such that the system can determine when factors are attached to component interfaces. When the reference count drops to zero, the `FactorManager` calls the factor set's `deactivate()` method. When the reference count first rises above zero, the `activate()` method is called. A designer needs only to implement these methods for factor sets that require activation or deactivation; for other factor sets they can simply be left as empty methods. Next, let us describe how to implement an adaptive queue using Perimorph.

4.4 Example: Adaptive Queue

As shown in Figure 4.4, we consider two implementations of a producer-consumer queue as a simple illustrative example. One implementation uses a fixed-length array and the other uses a dynamically resizeable vector. Both implementations provide the same operations, `put()`, `get()`, and `isFull()`. However, the vector `isFull()` operation will always return `FALSE` since the `put()` operation dynamically allocates the necessary structures for appending a new item to the queue. Functional concerns are defined by the array and vector-based queue factor sets, shown at the bottom of the figure. Recomposing a queue using a vector, requires the exchange of factors from the array-based factor set with those of the vector-based factor set. Moreover, the nontransient state of the array-based factor set must be transferred to the vector-based factor set. Two nonfunctional concerns are also implemented. Tracing, as defined by the trace factor set, prints informational messages about calls to the queue interface. Thread concurrency controls, defined by the mutex factor set, prevent the

producer and consumer threads from operating on the queue simultaneously.

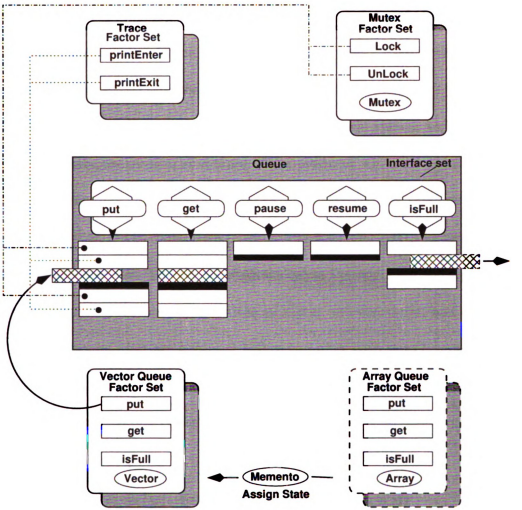


Figure 4.4: Composition of the adaptive queue showing several factor sets.

A segment of Perimorph code, used to convert the adaptive queue from an array to a vector-based implementation, is shown in Figure 4.5. Recomposition of components requires synchronization such that the factor set in use can be frozen, allowing the extraction of state and the replacement of factors. Synchronization can either be explicit or implicit. The adaptive queue employs explicit synchronization using `pause()` and `resume()` operations. The factors that implement `pause()` and `resume()` are

members of the mutex factor set, locking and unlocking the mutex. An interactive application, like the mapping application described in Section 4.5, may be implicitly synchronized since it may simply respond to user requests. Neither the tracing nor the concurrency controls are modified during this recomposition.

Figure 4.6 shows the simple producer-consumer queue system built using Perimorph. The left pane depicts that state of the producer while the right depicts the state of the consumer. The producer `put()s` an integer counter value on the queue, increments the counter by 4 and then repeats. The consumer `get()s` a value from the shared queue as needed. The center window depicts the application status, printing messages indicating the operations of the producer and consumer, and the invocation of trace factors, that are executed before and after each `get()` operation. Also printed are `[Pause]` and `[Resume]` messages, indicating the synchronization required for dynamic recomposition of the array-based queue as a vector-based queue.

4.5 Case Study: Mapping Application

In addition to the example adaptive queue application, we have used Perimorph to implement a digital elevation model (DEM) [146] mapping program. The DEM format is a common data format used by the United States Geological Survey (USGS) and other organizations for recording geographical elevation information. We developed our mapping application using Perimorph such that a 2D viewer can be recomposed into a 3D viewer at run time. Such recompositions are useful during handoff between dissimilar devices. For instance, a palmtop, due to limited memory, processing power

```
// If the queue is full and hasn't been switched to the
// vector-queue, then switch to the vector-queue.
if (isfull.booleanValue()&&(switched)) {
  // Pause operations on the queue.
  queue.invoke(``pause'',new Class[] {},new Object[] {});

  try {
    // Assign state from the array-queue to the vector-queue
    FactogramManager.assignFactogramset(
      ``QueueVectorFactogramset'',
      ``QueueArrayFactogramset'');
    // Replace the ``put(Object)'' factor
    ComponentManager.replacePreFactogram(``queue'',
      ``put(Object)'',
      ``QueueArrayFactogramset.ArrayPutFactogram'',
      ``QueueVectorFactogramset.VectorPutFactogram'');
    // Replace the ``get()'' factor
    ComponentManager.replacePreFactogram(``queue'',``get()'',
      ``QueueArrayFactogramset.ArrayGetFactogram'',
      ``QueueVectorFactogramset.VectorGetFactogram'');
    // Replace the ``isFull()'' factor
    ComponentManager.replacePreFactogram("queue",
      ``isFull()'',
      ``QueueArrayFactogramset.ArrayIsFullFactogram'',
      ``QueueVectorFactogramset.VectorIsFullFactogram'');
  } catch (Error e) {
    AdaptGUI.appendCenter(``Cannot switch queues:  ''
      +e.toString());
  }

  // Resume operations on the queue.
  queue.invoke(``resume'',new Class[] {},new Object[] {});

  switched = true;
  self().variables.setVar(``switched'',true);
}
```

Figure 4.5: Code segment used to recompose the adaptive queue into a vector based queue. Note the invoke calls for pausing and resuming queue access, the assignment of factor sets and the replacement of factors.

and display capability, might use only the 2D viewer. However, upon arriving at the office, a user may handoff the application to a workstation that can easily present a three-dimensional map. With Perimorph, the viewer can dynamically be transformed

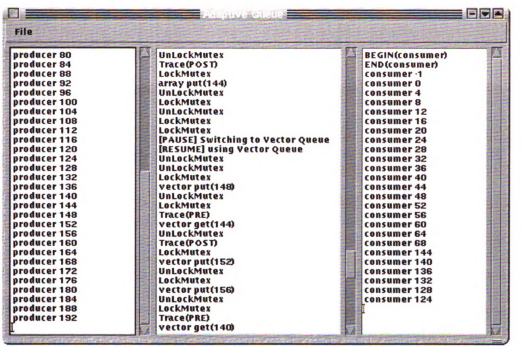


Figure 4.6: Adaptive queue example application. The left pane is the producer, the right the consumer and the center represents status information. Note the Pause and Resume messages where the array-based queue is exchanged with a vector-based queue by the control thread.

into a 3D viewer without loss of application state.

Figure 4.7 shows a two-dimensional representation of Mount St. Helens after eruption in 1980. This representation uses different colors to indicate changes in elevation. Typically, the lighter the color the greater the elevation. Initially, the mapping application comprises factors implementing a 2D viewer. Figure 4.8 depicts the factors recomposed during conversion to a 3D display. Upgrading the map requires modification of the functional concerns of both the map plotter and map window components. The map plotter paints the map on the map window. Depending on whether the map plotter and map window are composed using the two or three-dimensional factor set determines how the map data will be displayed. Nontransient

state, comprising DEM map data, is assigned from the two-dimensional to the three-dimensional factor set during factor exchange.

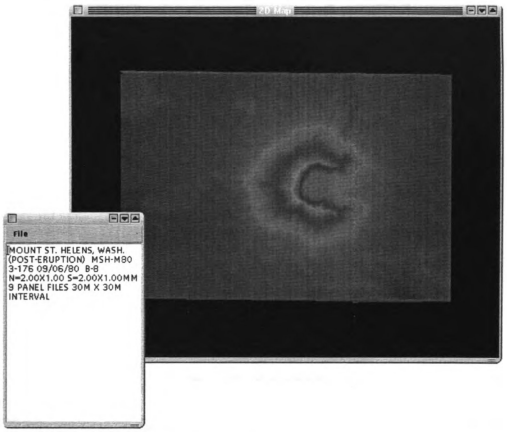


Figure 4.7: 2D map prior to recomposition.

Figure 4.9 shows a three-dimension map following dynamic, runtime recomposition. Proper initialization and construction of the GUI components require the coding of `activate()` and `deactivate()` factor set methods, which were left as empty methods for the adaptive queue.

Besides dynamic reconfiguration, constructing applications with Perimorph enables other state-related functionality. For example, both the adaptive queue and the elevation mapping application can be captured at any point in their execution and

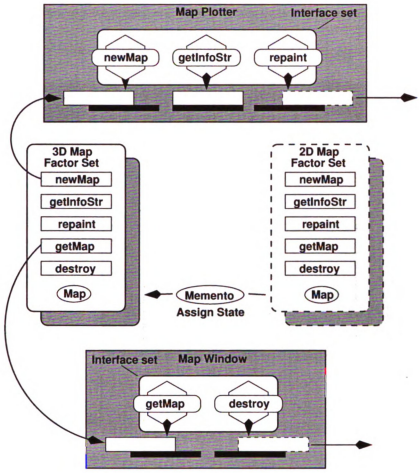


Figure 4.8: Recomposition of the DEM mapping application. Recomposition of both the map plotter and map window components is required. Operations on these components are called by the map control which does not require any change.

stored on disk or sent over the network to another machine. A state memento for an entire application can be constructed by saving the contents of the Perimorph stores and the nontransient state of all factor sets. This memento can be serialized and stored on disk or sent over a network. When the application is restarted, Perimorph requests a reload, deserializing these stores. References to components are reestablished as the application requests references from the `ComponentManager`. Thus, Perimorph applications can easily support checkpointing and distributed handoff in

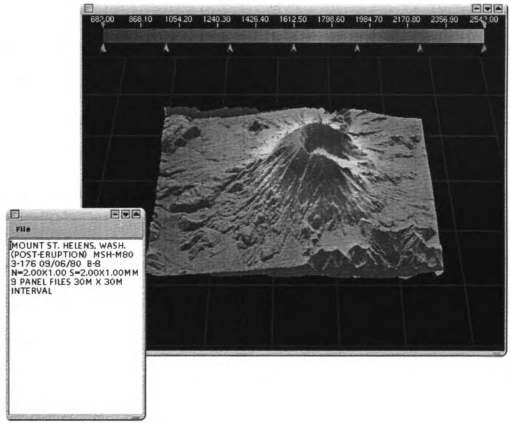


Figure 4.9: 3D map following recomposition.

addition to runtime recomposition. Moreover, composition can be adjusted following handoff, allowing adaptation to new environmental conditions, such as reduced memory or a smaller physical display.

4.6 State Maintenance in Dynamic River

Pipeline recomposition may move segments to different networked hosts to improve performance or (by inserting or removing segments) alter the pipeline functionality. As such, stream processing can be dynamically altered to meet new requirements or to better use the computational resources of networked hosts. However, to avoid

data loss, dynamic recomposition requires that data stream state be maintained when pipeline segments are moved or when segments are added or deleted. Where adaptation at the program level often requires extraction and transformation of nontransient component state, operator reconfiguration and redeployment may require protocols to ensure that reconfiguration is orchestrated and avoids loss of stream data. Moreover, removal and insertion of operators should occur at application specific semantic boundaries so that application processing can continue uninterrupted. In this section we extend our study to the design and implementation of state maintenance for support of adaptive, data streaming applications. There are two basic cases to consider for maintaining data stream state: graceful shutdown and faults.

4.6.1 Graceful Shutdown

Figure 4.10 is a sequence diagram depicting the graceful termination of **streamin** in response to **stop** command issued by **ctrlcmd**. To issue a stop command, **ctrlcmd** connects to **dynriverd** and requests that a **stop** command be sent to **streamin**. Subsequently, **streamin** sends a **stop** command to **streamout**. In response, **streamout** reads and emits records until the outer most scope has been reached. The outer most scope is identified by **streamout** receiving a record with a scope equal to 0 that is either a data record or a **CloseScope** record. If a **CloseScope** record is received, it is transmitted to **streamin** prior to **streamout** closing the connection. As such, segment termination is gracefully completed at the outer scope boundary. All records read by **streamin** are written and the pipeline segment is flushed prior to shutdown.

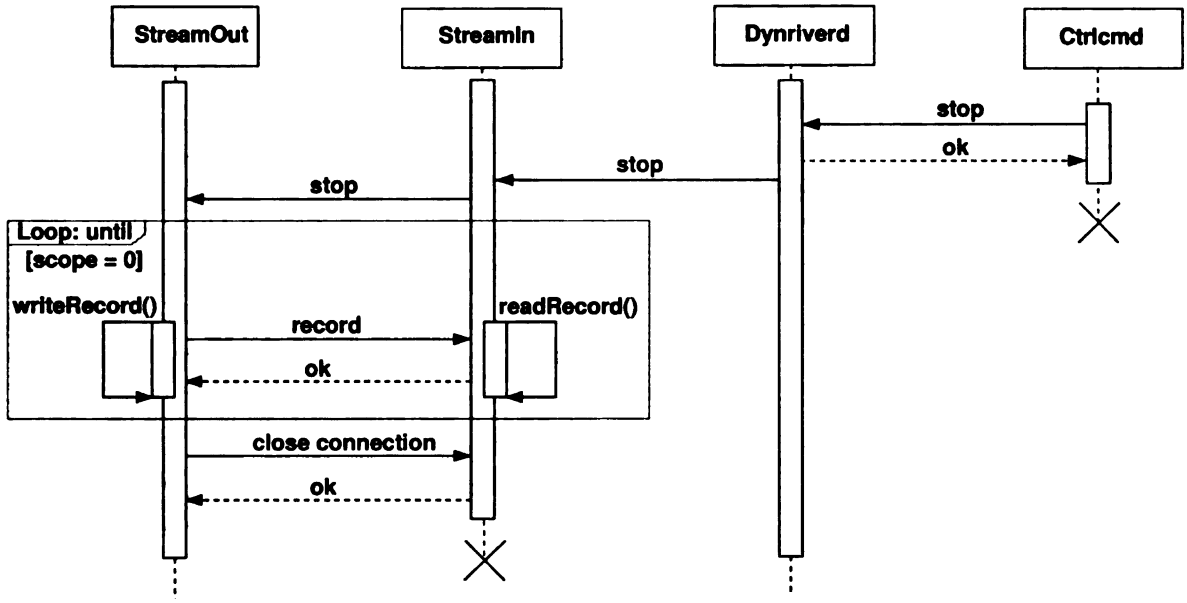


Figure 4.10: A sequence diagram depicting the graceful termination of `streamin` in response to a `stop` command.

4.6.2 Fault resiliency

Although it is difficult or impossible to address catastrophic hardware failures, we can strive to design a system to be resilient to lesser faults. Since faults typically risk some data loss, it is necessary to consider how to restore the state of a distributed pipeline such that execution following a fault will produce correct results. That is, a recovery oriented approach [147] is needed to close scopes left open and resume data stream processing at a scope boundary and depth suitable for resynchronization. By default, Dynamic River resumes processing at the outer most scope in response to unexpected segment termination.

Figure 4.11 depicts a sequence diagram for `streamin`'s response to the unexpected termination of `streamout`. When an unexpected upstream termination generates an end-of-file (EOF) condition on `streamin`'s input socket, `streamin` responds by writing `BadCloseScope` records until the outer most scope has been reached. This

closes all open scopes and enables the detection of early scope closure by downstream operators. In effect, this action enables the detection of an exception condition by downstream operators while closing open scopes and returning the data stream to a point where synchronization is possible.

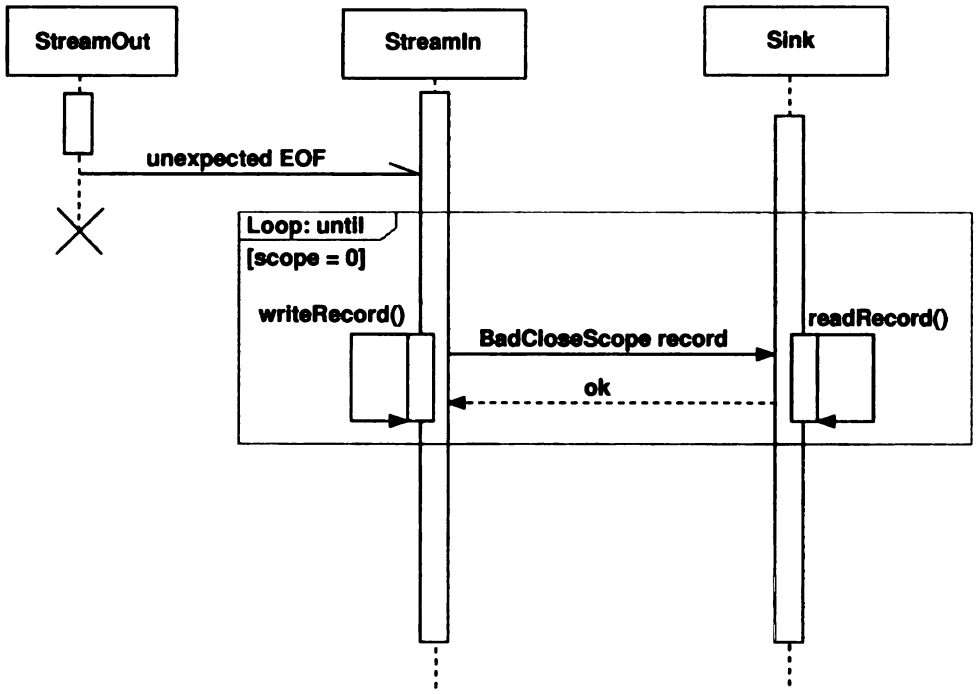


Figure 4.11: A sequence diagram depicting the unexpected termination of `streamout`.

Figure 4.12 depicts a sequence diagram for when a connection is reestablished between `streamout` and `streamin`. The `scopesync` guard is a short function that is shown in Figure 4.13. The `scopesync` algorithm returns `true`, indicating that the data scope has reached a synchronization point, when either an `OpenScope` record was read with a scope depth equal to 1 or when a `CloseScope` or `BadCloseScope` record was read with a scope depth equal to 0.

During reconnection following a fault, as shown in Figure 4.12, `streamout` reads and consumes records until `scopesync` returns `true`. The record that triggered

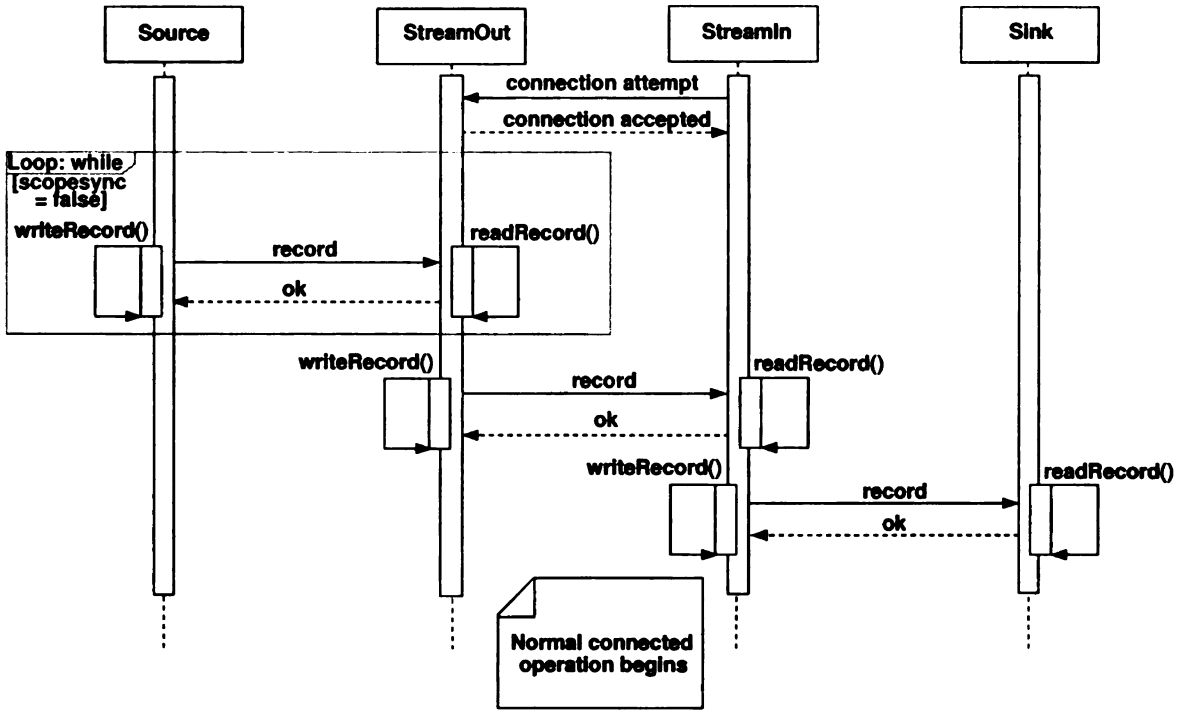


Figure 4.12: A sequence diagram depicting `streamin` reconnecting to `streamout`.

```
function scopesync(record_header_t
hdr)
begin
  if (hdr.scope = 1) and
    (hdr.subtype = open_scope)
      then return true
  else if (hdr.scope = 0) and
    ((hdr.subtype ≠ close_scope) and
    (hdr.subtype ≠ bad_close))
      then return true
  else
    return false
  endif
end
```

Figure 4.13: Data scope synchronization algorithm.

`scopesync` to return `true` is written to `streamin` and then normal pipeline operation resumes. Thus, in the face of a fault, data stream state is resynchronized by orchestrated interaction between `streamout` and `streamin`.

4.7 Related Work

The problem of state capture has been explored in a variety of domains. Checkpointing [31,32], process or thread migration [32–34], and mobile agents [35,36] all employ mechanisms that extract state from a running program, and restore it in some way. Storing a snapshot of a running program provides a level of fault tolerance for a program that executes over long periods. Checkpointing allows an image of the running program to be stored in a file. If the program or machine should crash, this image can be used to restart the program as of its last checkpoint. However, process level checkpointing does not allow transfer of state at the component level, since taking a full image of a program will not help extract and inject state at the component level. Process migration involves saving the process state and restoring it on another machine. Mobile agents [35, 148, 149] use state preservation and restoration to move from machine to machine. Several research groups [32,33] have implemented mechanisms for migrating Java threads. Although threads are not complete processes, the granularity of state preservation exceeds that of a component.

The state capture problem has also drawn attention from the adaptive middleware community. DynamicTAO [68] and some agent systems [150] use a state transfer process similar to the memento pattern [145]. Even if state transfer is allowed between different components, however, the exchange is often between data structures that are identical. Support for transfer of state between dissimilar components requires the conversion of the extracted memento into a form acceptable for injection into a new component.

The memento pattern is not the only way to address state maintenance, however. Candea et al [151] rely on the externalization of nontransient state to support *recovery-oriented computing (ROC)* [147, 152, 153]. This project targets restarting, or *microrebooting*, software components when they fail, enabling faster recovery and reducing down time. Component state is stored outside the component. When a component fails, its state is not lost and can be recovered following reboot. Unlike our queue example above, microreboot does not require the transformation of state to meet the requirements of a different implementation. Externalizing component state removes the need to extract and inject component state during recomposition, but does not alleviate the need for transforming state when exchanging different implementations of a component.

In data streaming applications, Aurora [141, 142] quieses subnets of operators by "draining" operators that contain only soft state prior to reconfiguration. This approach is similar to Dynamic River's use of protocols, but does not use data stream scope to detect a viable stream juncture at which to begin pipeline reconfiguration or redeployment. Borealis [143] addresses state maintenance for aggregate operators, that contain nontransient state, by replaying data stream data following operator redeployment. However, knowing when data buffered for replay will no longer be needed is difficult in many applications. Further study on support of aggregate operators is needed; approaches like the memento pattern may also prove useful for enabling state capture in data streaming applications.

4.8 Discussion

In this chapter, we presented a programmer's API, Perimorph, that enables principled implementation of collateral change and declaration of nontransient state. Perimorph enables dynamic, runtime recomposition of both functional and nonfunctional concerns. This API supports transparent reconfiguration of components. Factor sets provide a construct for describing how collateral change affects system recomposition. Nontransient state is defined at the factor set scope and can be assigned between factor sets of equivalent abstract type using state normalization in conjunction with the memento pattern. In particular, we built both a simple, illustrative "adaptive queue" and a digital elevation mapping application, demonstrating the usefulness of Perimorph constructs. In addition, we studied collateral change in Dynamic River, and introduced the concept of data stream scope. We described several protocols that, in conjunction with stream scoping, enable graceful shutdown, pipeline reconfiguration and fault resiliency for data stream processing.

Further study is needed on how best to factor adaptive systems with respect to collateral change and automate state maintenance. Constructs that provide a high level of abstraction for systems and APIs, like Perimorph, can further improve a software designer's ability to understand and build applications supporting adaptation. Moreover, while the memento pattern provides a useful abstraction of the state transfer and transformation process, it does not address the implementation and data dependent nature of state transformation. Specifically, dynamic recomposition requires that the relation between the state of an old component and its replacement be under-

stood. Systems that support state maintenance in the face of dynamic composition will benefit from methods for reasoning about the transformation of nontransient state, component equivalence and collateral change to verify correctness and guide design and implementation.

Chapter 5

Perceptual Memory

The third major issue we address is decision making. Autonomic software needs to decide how to adapt to dynamic external conditions involving hardware components, network connections, and changes in the surrounding physical environment [9,10,12]. For example, to meet the needs of mobile users, software integrated into handheld, portable and wearable devices must balance several conflicting and possibly cross-cutting concerns, including quality-of-service, security, energy consumption, and user preferences. Applications that monitor the environment using sensors must interpret the knowledge gleaned from those observations such that current and future requirements can be met. Autonomous systems must be able to react to sensor input and make decisions that enable the system to adapt to uncertain and changing conditions. Moreover, many systems must make decisions in real time to prevent damage or loss of service with only high-level human guidance.

We argue that *perceptual memory*, a type of long-term memory for remembering external stimulus patterns [45], may offer a useful model for an important compo-

nent of decision making in context-aware, adaptive software. The ability to remember complex, high-dimensional patterns that occur as a product of interaction between application users and the environment, and to quickly recall associated actions, enables timely, autonomous system response. Moreover, it has been proposed that perceptual memory can play an important role for enabling autonomous software to discover new or improved algorithms [46].

This chapter presents MESO, a perceptual memory system we designed to support online, incremental learning and decision making in autonomic systems. A novel feature of MESO is its use of small agglomerative clusters, called *sensitivity spheres*, that aggregate similar training samples. Sensitivity spheres are partitioned into sets during the construction of a memory-efficient hierarchical data structure. This structure enables the implementation of a content-addressable perceptual memory system: instead of indexing by an integer value, the memory system is presented with a pattern similar to the one to retrieve from storage. Moreover, the use of sensitivity spheres facilitates a high rate of data compression, which enables MESO to execute effectively in resource-constrained environments. Additional benefits of MESO include: incremental training, fast reorganization of the existing hierarchical data structure, high accuracy, and lack of dependence on a priori knowledge of adaptive actions or categorical labels. Each of these benefits is important to online decision making.

After describing the design and operation of MESO, we demonstrate its accuracy and performance by evaluating it strictly as a pattern classifier. In these experiments, cross-validation experiments are used to determine accuracy using standard data sets. The performance of MESO, in terms of accuracy and execution time, compares fa-

vorably to that of other classifiers across a wide variety of data sets.

The remainder of this chapter is organized as follows. Section 5.1 discusses background and related work. Section 5.2 describes MESO's clustering algorithm and the role of sensitivity spheres; three data compression methods that leverage MESO's internal structure are also introduced. Section 5.3 presents experimental results that assess MESO performance (accuracy, compression rate, and execution time) on eight standard data sets. MESO performance is also compared directly with that of other classifiers. Section 5.4 presents related work. Finally, Section 5.5 concludes this chapter.

5.1 Background

Our work with MESO explores data clustering methods for associating adaptive responses with observed or sensed data. The embodiment of this approach is a clustering [4, 154] algorithm that produces an internal model of environmental stimuli. As shown in Figure 5.1, two basic functions comprise the operation of MESO: *training* and *testing*. During training, patterns are stored in perceptual memory, enabling the construction of an internal model of the training data. Each training pattern is a pair (x_i, y_i), where x_i is a vector of continuous, binary or nominal values, and y_i is an application specific data structure containing meta-information associated with each pattern. Meta-information can be any data that is important to a decision-making task, such as the codification of an adaptive action to be taken in response to certain environmental stimuli. MESO can be used as a pattern classifier [4] if an a

priori categorization is known during training. That is, for a classification task, the meta-information need only comprise a categorical label assigning each pattern to a specific real-world category. However, MESO does not rely on categorical labels or meta-information, and instead incrementally clusters the training patterns in a label independent fashion. Lack of dependence on a priori knowledge of categorical labels is an important distinction between MESO and other pattern classifiers. Where many classifiers leverage categorical labels to better classify training samples [4], perceptual memory must accurately retrieve training samples and meta-information without knowledge of fixed categories. Meta-information is malleable, enabling its update by a decision maker to address changing context, user preference or new adaptive actions.

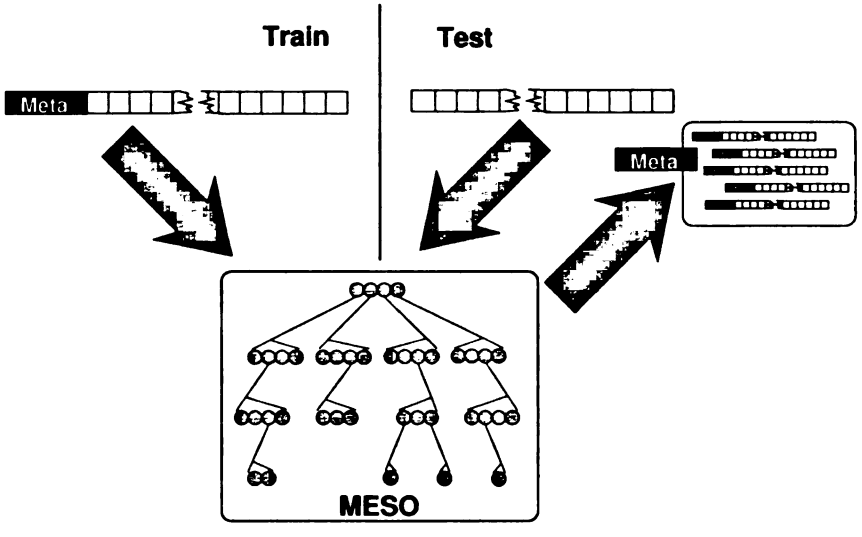


Figure 5.1: High level view of MESO.

Like many clustering and classifier designs, MESO organizes training patterns in a hierarchical data structure, such as a tree, for efficient retrieval. Once MESO has been trained, the system can be queried using a pattern without meta-information. MESO tests the new pattern and returns either the meta-information associated with

the most similar training pattern or a set of similar training patterns and their meta-information. In some domains, it may not be possible to collect a representative set of training samples a priori, so *incremental learning* is required. This process uses an estimation function f_i, which is a function of the first i samples, and which is constructed incrementally using the previous estimator f_{i-1} and the current pattern (x_i, y_i).

5.2 MESO Design and Operation

If categorical labels are known during training, MESO can function as a pattern classifier that incrementally classifies environmental stimuli or other data while accommodating very large data sets. Prior to developing MESO, we conducted experiments using the HDR classifier [155] for this purpose. The insights gained from those experiments led to our design of MESO. MESO incrementally constructs a model of training data using a data clustering approach whereby small clusters of patterns, called *sensitivity spheres*, are grown incrementally. These sensitivity spheres are organized in an hierarchical data structure, enabling rapid training and testing, as well as significant data compression, while maintaining high accuracy. In this section, the details of MESO's core algorithm and data structures are discussed. MESO is based on the well-known leader-follower algorithm [156], an online, incremental technique for clustering a data set. The basic operation of the leader-follower algorithm is shown in Figure 5.2. A training pattern within distance δ of an existing cluster center is assigned to that cluster; else a new cluster is created.

```
initialize cluster centers, δ
input pattern x(t)
find nearest center, e.g., w_i
if d(x_i, w_i) ≤ δ
  update cluster center
else
  create new center w_j = x(t)
next pattern
end
```

Figure 5.2: Leader-follower algorithm (adapted from Duda and Hart [4]).

Traditionally, the value of δ is a constant value initialized based on a user's understanding or experience with the data set at hand. However, this approach makes it difficult to generalize the leader-follower algorithm to arbitrary data sets. We address this issue in MESO by calculating the value of δ incrementally and by organizing the resulting clusters using a novel hierarchical data structure, as described below.

5.2.1 Sensitivity Spheres

In adaptive software, training patterns comprise observations related to quality of service or environmental context, such as network bandwidth or physical location. The quantity of training patterns collected while a system executes may be very large, requiring more memory and processing resources as new patterns are added to the classifier. Unlike the traditional leader-follower algorithm, in MESO the value of δ changes dynamically, defining the sensitivity spheres, which are small agglomerative clusters of similar training patterns. Effectively, the value of δ represents the sensitivity of the algorithm to the distance between training patterns. Figure 5.3 shows an example of sensitivity spheres for a 2D data set comprising three clusters. A

sphere's center is calculated as the mean of all patterns that have been added to that sphere. The δ is a ceiling value for determining if a training pattern should be added to a sphere, or if creation of a new sphere is required. As defined by the δ value, sphere boundaries may overlap, however, each training pattern is assigned to only one sphere, whose center is closest to the pattern.

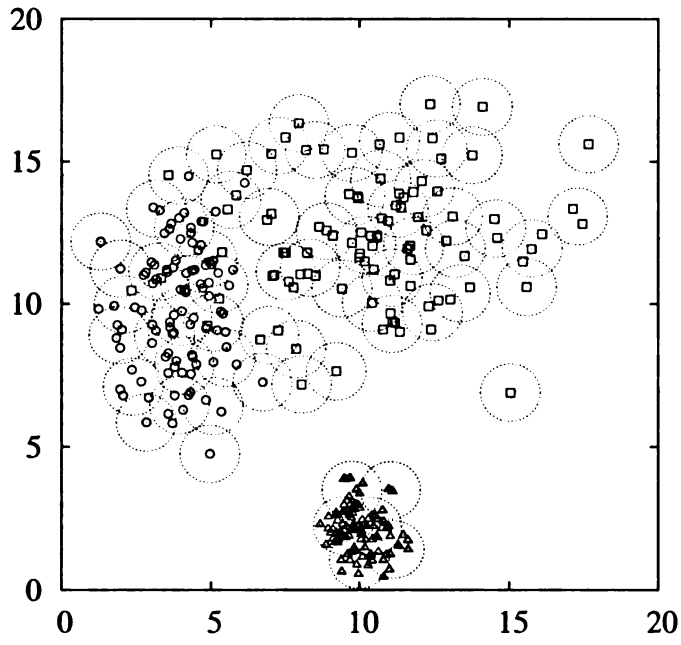


Figure 5.3: Sensitivity spheres for three 2D-Gaussian clusters. Circles represent the boundaries of the spheres as determined by the current δ. Each sphere contains one or more training patterns, and each training pattern is labeled as belonging to one of three categories (circle, square, or triangle).

5.2.2 MESO Tree Structure

As with many approaches to clustering and classification, MESO uses a tree structure to organize training patterns for efficient retrieval. However, the MESO tree, depicted in Figure 5.4, is novel in that its organization is based on sensitivity spheres. A MESO tree is built starting with a root node, which comprises the set of all sensitivity spheres.

The root node is then split into subsets of similar spheres, producing child nodes. Each child node is further split into subsets until each child comprises only one sphere. Many clustering algorithms construct a tree by agglomerating individual patterns into large clusters near the root of the tree, and then splitting these clusters at greater tree depths. Reorganizing such a tree requires processing of the training patterns directly. In contrast, MESO's consolidation of similar patterns into sensitivity spheres enables construction of a tree using only spheres, rather than individual patterns. Moreover, a MESO tree can be reorganized using only existing sensitivity spheres and hence more rapidly than approaches that require direct manipulation of training patterns.

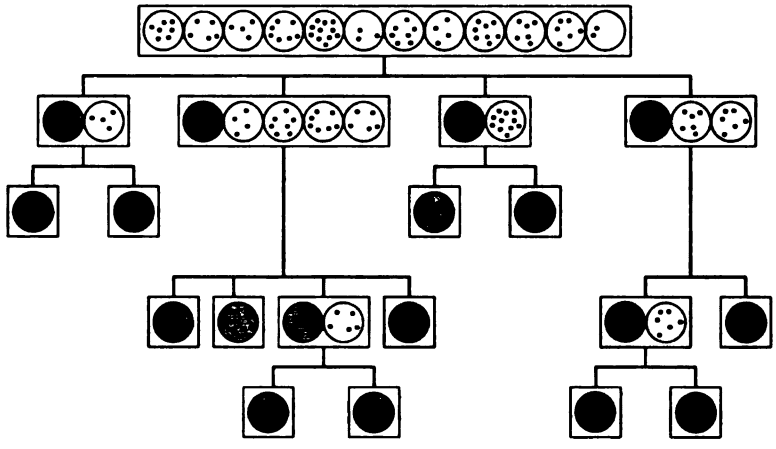


Figure 5.4: MESO tree organization. The rectangles are partitions and the shaded spheres are partition pivots. Partitions are split successively until a leaf is formed where a partition contains only one sphere.

The set of sensitivity spheres for a data set is partitioned into subsets of similar spheres during the construction of a MESO tree. Each node of the tree comprises one such subset, called a *partition*. Figure 5.5 shows the algorithm for building a MESO tree from existing sensitivity spheres. The parameters for this algorithm include: q, the number of children per tree node; p, a partition pivot sphere; *parent*, the parent

node for a set of children; *root*, the root node of the tree; and *part*, the partition associated with a *parent* node. The algorithm is recursive, starting at the *root* of the tree with a partition (*part*) comprising all spheres in the tree. Each call to *splitpartition* divides *part* into q smaller partitions and assigns these partitions as children of the *parent* node. The processes terminates when a partition contains only one sphere. When a partition is divided, the first sphere in each of the q segments is identified as a *pivot*, which is used subsequently in assigning other spheres to that partition. Specifically, for a sphere to be added to a partition requires that the sphere be nearer to that partition's pivot than to the pivot of any other child node. Intuitively, this algorithm can be viewed as a q-way heap sort that organizes sensitivity spheres according to their similarity. The parameter q can be set to any integer value ≥ 2 and, in our experience, has limited impact on the accuracy of retrieving patterns from MESO during testing. In the experiments described in Sections 5.3 we set $q = 8$.

As a result of this process, each non-leaf node in a MESO tree has one or more children, each comprising a subset of the parent's sensitivity spheres. Smaller partitions comprise training patterns of greater similarity and provide finer discrimination during testing. Moreover, the partitioning of sensitivity spheres produces a hierarchical model of the training data. That is, each partition is an internal representation of a subset of the training data that is produced by collecting those spheres that are most similar to a pivot sphere. At deeper tree levels, parent partitions are split, producing smaller partitions of greater similarity.

Using a test pattern, the meta-information associated with the most similar training pattern can be retrieved. Retrieval proceeds by comparing a test pattern with a

```
begin initialize q,p = nil,root,part
  splitpartition(q,p,root,part)

procedure splitpartition(q,p,parent,part)
  if part has a cardinality > 1
    select q pivots from part including
        p, p1..pq
    create q subpartions, part1..partq
    foreach si in part do
      find the nearest pj pivot and add
        si to partj
    done
    foreach pj,partj pair do
      create a child node
      add pj to child
      add child to parent
      splitpartition(q,pj,child,partj)
    done
  endif
```

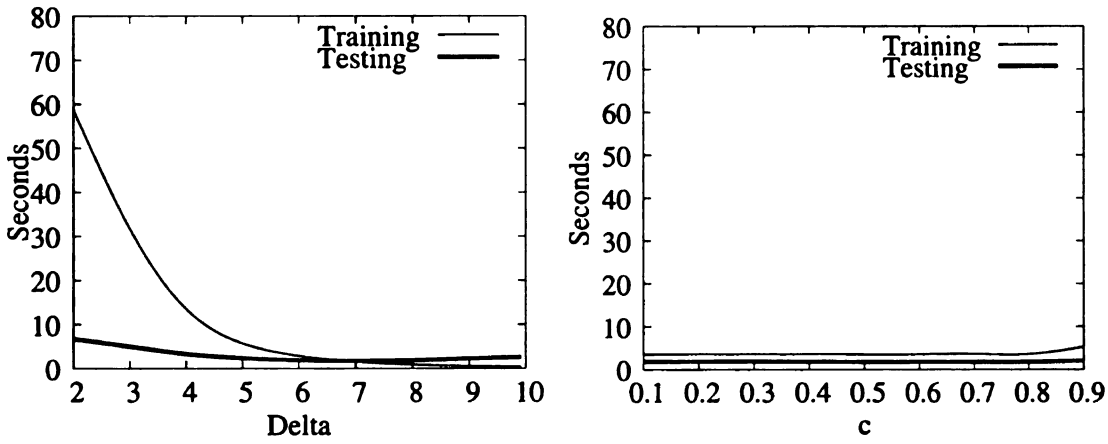
Figure 5.5: Building a MESO tree from sensitivity spheres.

pivot, starting at the root, and following one or more paths of greatest similarity. At a leaf node, the meta-information associated with the most similar training pattern is returned. In lieu of returning only one training pattern's meta-information, all the training patterns contained in the most similar sensitivity sphere can be retrieved. The MESO tree can be constructed incrementally, enabling MESO to be trained and tested during simultaneous interaction with users or other system components.

5.2.3 Sensitivity Sphere Size

An important consideration in building an effective MESO tree is the appropriate value of δ to use in defining sensitivity spheres. Our experiments show that training and testing time are influenced by the choice of δ. For example, Figure 5.6(a) shows results for the letter data set (discussed further in Section 5.3.1), with δ fixed at

various values. If δ is too small, training time increases dramatically. If δ is too large, testing time increases (more evident for larger data sets). Moreover, data set compression requires a proper value of δ to balance the tradeoff between compression rate and accuracy.



(a) using fixed δ (b) using dynamic δ

Figure 5.6: Training and testing time for the letter data set (see Section 5.3.1).

To address this issue, the value of δ is adjusted incrementally as MESO is trained. The δ *growth function* balances sphere creation rate and sphere size. Figure 5.7 shows the algorithm for construction of sensitivity spheres from training patterns. This algorithm begins by initializing the sensitivity δ, the first sensitivity sphere mean vector (u_1), and the first sensitivity sphere (s_1) to 0, x_1, and empty, respectively. Then, for each pattern (x_j), the closest sphere mean vector is located. If the distance between x_j and the nearest sphere mean is less than or equal to δ, then x_j is added to the sphere and the sphere mean recalculated. If the distance between the closest sphere mean and x_j is greater than δ, then the δ is grown, a new sphere is created for x_j and an associated mean vector is initialized.

```
begin initialize δ = 0, u_1 = x_1, s_1
  foreach x_j sample do
    find the nearest u_i for x_j
    if distance from u_i to x_j <= δ
      add x_j to s_i
      recompute u_i using samples in s_i
    else
      let δ = grow_δ
      create new s_{i+1}
      add x_j to s_{i+1}
      let u_{i+1} = x_j
    endif
  done
```

Figure 5.7: Sensitivity sphere creation algorithm.

A good $grow_\delta$ function needs to balance sphere creation with sphere growth. Rapid growth early in the training process can produce few spheres with very large δ's, creating a coarse-grained, inefficient representation. However, slow growth produces a large number of very small spheres, and the resulting tree is expensive to search. In the MESO implementation reported here, the δ growth function is:

$$grow_\delta = \frac{(d-\delta)\frac{\delta}{d}f}{1+\ln(d-\delta+1)^2},$$

where d is the distance between the new pattern and the nearest sensitivity sphere. The $\frac{\delta}{d}$ factor scales the result relative to the difference between the current δ and d. Plotted in Figure 5.8 is the function $\frac{1}{1+\ln(d-\delta+1)^2}$ Intuitively, the denominator of $grow_\delta$ limits the growth rate based on how far the current δ is from d. If d is close to δ then δ will grow to be nearly equal to d. However, if d is much larger than δ, then the increase will be only a small fraction of $d-\delta$. As such, δ growth is discouraged in the face of outliers, new experience and widely dispersed patterns. Hence, when

a new training pattern is distant from existing spheres, a new sphere is likely to be created for it.

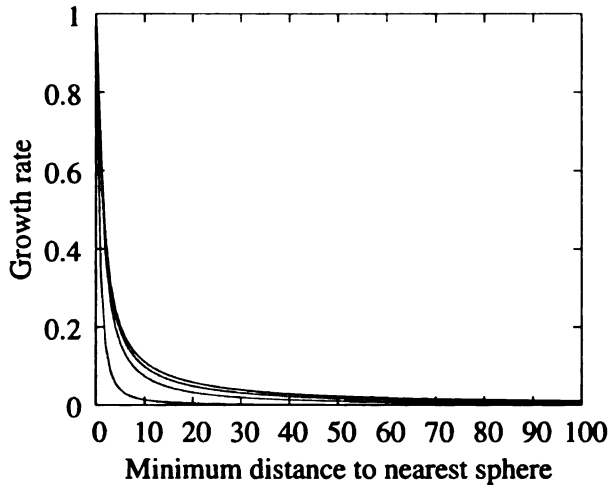


Figure 5.8: Sensitivity sphere growth function denominator for $\delta = 1, 10, 100, 1,000$ and $10,000$.

The activation function, f, needs to balance the creation of new spheres with sphere growth. Table 5.1 depicts 6 candidate activation functions, where $r = \frac{spheres}{patterns}$ and c is a configuration parameter in the range $[0, 1.0]$. Increasing c moves the center of the activation function to the right. The statistics shown were generated using cross-validation (discussed further in Section 5.3.1) in conjunction with the letter and MNIST data sets. As shown, functions (b), (d) and (f) produce a significantly larger number of sensitivity spheres than the other functions. However, a large sphere count inhibits compression (discussed further in Section 5.2.4) and exhibits higher training and testing times. Functions (c) and (e) produce fewer spheres, but exhibit somewhat lower accuracies or longer training and testing times than function (a). Overall, function (a) shows the best balance between accuracy and training and testing times while producing a sufficiently small number of spheres to enable high compression.

Intuitively, function (a) inhibits sensitivity sphere growth when the number of spheres is small compared to the number of patterns, but encourages rapid sphere growth when the number of spheres is large. The remaining experiments presented in this paper use the activation function (a), with parameter c set to 0.6.

Figure 5.6(b) plots the measured training and testing time for the letter data set, against the configuration parameter, c. The $grow_\delta$ function balances sphere production with sphere growth, producing good spheres for a wide range of values for c. Only for very large values of c is growth inhibited sufficiently to significantly impact training time. The $grow_\delta$ function promotes the production of trees that are comparable with good choices for fixed δ values.

Using this $grow_\delta$ function, let's consider the example data set shown in Figure 5.9. This synthetic data set comprises three 3D-Gaussian clusters. Two of these clusters overlap while the third is largely separated from the other two.

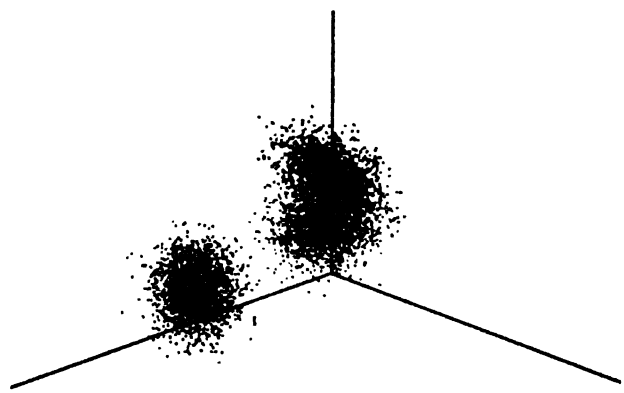
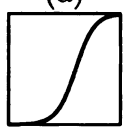
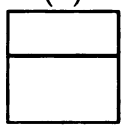
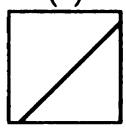
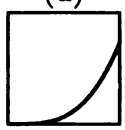
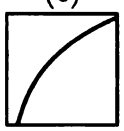
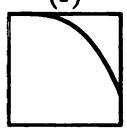


Figure 5.9: The Gaussian 3D example dataset.

Figure 5.10 depicts several frames showing the production and growth of sensitivity spheres using the example data set shown in Figure 5.9. The training sequence

Table 5.1: Comparison of 6 different activation functions using $c = 0.6$ for the letter data set (see Section 5.3.1).

Data set	(a)  $\frac{1}{2} + \frac{\tanh\left(\frac{3r}{c} - 3\right)}{2}$	(b)  $c = 0.6$	(c)  $\frac{r}{c}$
Letter			
Accuracy%	90.6±0.3%	88.1±0.2%	87.9±0.2%
Training (s)	1.8±0.0	2.0±0.0	1.8±0.0
Testing (s)	0.2±0.0	0.2±0.0	0.2±0.0
Spheres	570±22	659±2	563±3
MNIST			
Accuracy%	94.3±0.1%	94.8±0.1%	94.6±0.1%
Training (s)	70.9±1.1	109.9±1.6	92.1±3.2
Testing (s)	6.7±0.2	9.3±0.3	8.6±0.7
Spheres	6279±11	9696±6	7050±8
	(d)  $(\frac{r}{c})^3$	(e)  $log_{10}\left(\frac{9r}{c}\right)$	(f)  $1 - (\frac{r}{c})^3$
Letter			
Accuracy%	88.7±0.3%	87.9±0.2%	88.7±0.2%
Training (s)	2.2±0.0	1.7±0.0	2.5±0.0
Testing (s)	0.3±0.0	0.2±0.0	0.3±0.0
Spheres	803±8	510±3	953±4
MNIST			
Accuracy%	95.0±0.1%	94.5±0.1%	95.2±0.1%
Training (s)	125.9±1.4	86.7±1.2	175.0±1.6
Testing (s)	10.2±0.1	7.9±0.2	11.3±0.3
Spheres	10300±19	6300±2	13758±2

proceeds from left to right. The top sequence shows sphere δ's while the bottom sequence shows the average distance between sphere patterns and the sphere mean vector. Notice that early training produces many small spheres while sphere growth

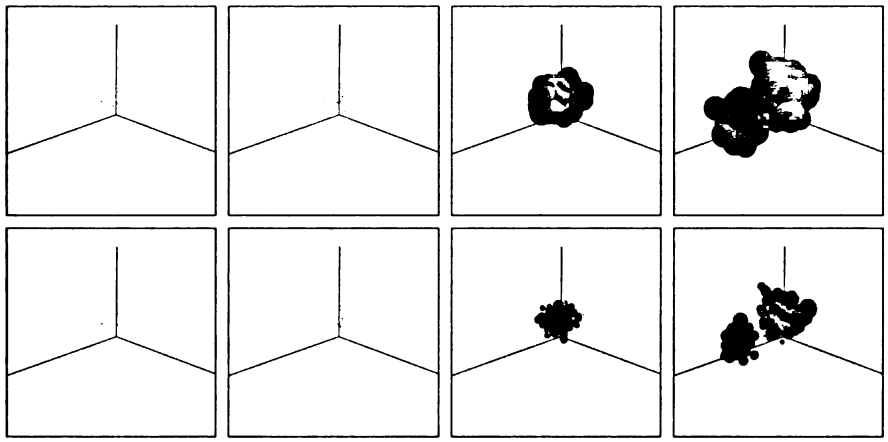


Figure 5.10: Snapshot frames showing MESO sensitivity spheres and mean sphere distances as the spheres are built for the Gaussian 3D example dataset. Top, sphere δ's. Bottom, mean sphere distances.

is inhibited until later.

5.2.4 Compression

Online learning is a data intensive process, and adaptive systems often must continue to function for long periods of time while responding to the sensed environment. The enormous amount of input data consumes substantial processing and storage resources, potentially inhibiting timely responses or impacting application performance. MESO uses lossy compression to limit the consumption of memory and processor cycles. Compression is applied on a per sensitivity sphere basis. That is, rather than trying to compress the entire data set using a global criterion, the patterns in each sensitivity sphere are compressed independent of other spheres. Since information about each sphere is retained, the effect of information loss on accurate pattern retrieval is minimized. We implemented three types of compression, the evaluation of

which is discussed in Section 5.3.2.

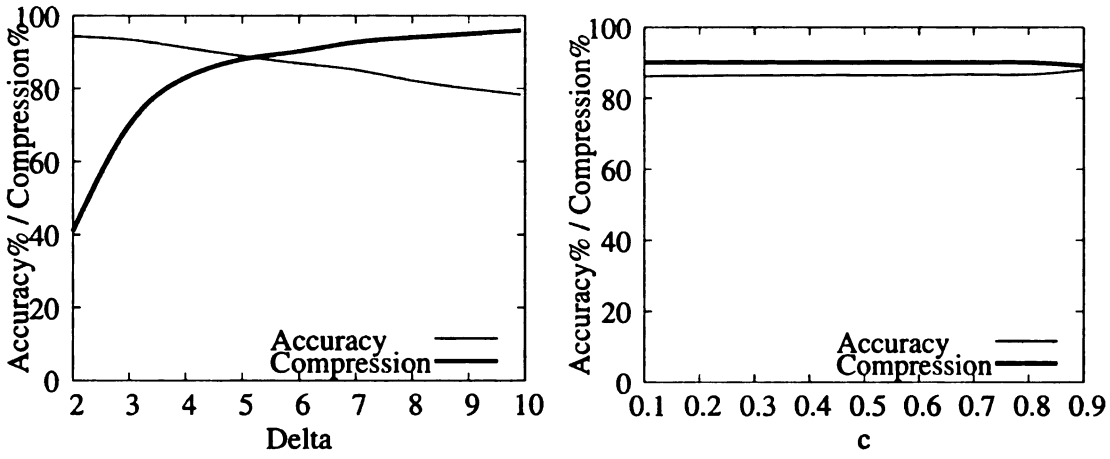
Means compression reduces the set of patterns in each sensitivity sphere to the mean pattern vector. If training patterns are categorically labeled, then the mean vector for each category is retained. This is the most aggressive and simple of the compression methods. Moreover, the computational requirements are quite low.

Spherical compression is a type of boundary compression [157] that treats patterns on the boundaries between spheres as most important to the classification of test patterns. For each sphere, the feature values are converted to spherical coordinates. Along a given vector from the sphere center, only those patterns farthest from the sphere center are kept.

Orthogonal compression removes all the patterns that are not used for constructing an orthogonal representation of a sphere's patterns. The idea is to keep only those patterns that are most important as determined by their orthogonality. Patterns that represent parallel vectors in m-dimensional space are removed.

Using compression requires some consideration of δ growth. As shown in Figure 5.11(a), accuracy decreases with higher compression rates. Moreover, compression rate is directly influenced by the value of δ . That is, if the sensitivity sphere δ is very large and few spheres are produced, compression is high and too much information will be lost during compression. However, if the δ is very small, very little compression is possible.

To avoid growing overly large spheres in the face of compression, we modified the



(a) accuracy and compression, fixed δ (b) accuracy and compression, variable δ

Figure 5.11: Effect of means compression on training and testing times for the letter data set, using fixed and variable δ.

activation function f to be:

$$f = \frac{1}{2} + \frac{\tanh\left(\frac{3r}{\max(v,c)} - 3\right)}{2},$$

where v is the compression rate, defined as the fraction of patterns removed during compression. Under high compression rates, using v instead of c as the center point of the activation function causes the sigmoid curve to move to the right, further inhibiting sphere growth. Figure 5.11(b) plots the accuracy and compression rate for experiments on the letter data, using means compression and the modified activation function. Accuracy and compression rate remain high for a wide range of c values. Only very large values of c cause a drop in compression rate, along with a slight increase in accuracy.

5.2.5 Complexity

Table 5.2 shows the space and time complexities for training MESO and several well-known clustering algorithms [1]. In this table, n is the number of patterns, k is the number of clusters and l is the number of iterations to convergence. Without compression, MESO has a worst case space complexity of $O(n)$, comparable to the shortest spanning path algorithm. MESO's memory consumption can be significantly reduced with compression, as shown in the next section.

Intuitively, time complexity for training can be considered in terms of locating the sensitivity sphere nearest to a new pattern and adding the pattern to that sphere. If a sufficiently close sphere cannot be found, a new sphere is created. Locating the nearest sphere is an $O(\log_q k)$ operation. This search must be completed once for each of n patterns. Each pattern must also be added to a sensitivity sphere, and k sensitivity spheres must be created and added to the MESO tree. Assuming an appropriate value of δ and a data set of significant size, this process yields a complexity of $O(n \log_q k) + O(n) + O(k) + O(k \log_q k)$ which reduces to $O(n \log_q k)$.

Table 5.2: Space and time complexities for MESO and several other clustering algorithms [1].

Algorithm	Time	Space
MESO	$O(n \log_q k)$	$O(n)$
leader	$O(kn)$	$O(k)$
k-means	$O(nkl)$	$O(k)$
ISODATA	$O(nkl)$	$O(k)$
shortest spanning path	$O(n^2)$	$O(n)$
single-link	$O(n^2 \log n)$	$O(n^2)$
complete-link	$O(n^2 \log n)$	$O(n^2)$

The complexity for classifying a test pattern using MESO is $O(\log_q k) + O(\bar{s})$

for a balanced tree, where q is the maximum number of children per node, $\bar{s}$ is the average number patterns agglomerated by a sensitivity sphere, and k represents the number of sensitivity spheres produced. The $\bar{s}$ component represents the number of operations required to assign a category label once the most similar sensitivity sphere has been located. Thus, the worst case search complexity occurs when only one cluster is formed and the search algorithm degenerates into a linear search of $O(n)$. Conversely, a best case search complexity of $O(\log_q n)$ occurs when one sensitivity sphere is formed for each training pattern.

5.3 MESO Assessment

In this section, we evaluate MESO as a pattern classifier, where pattern meta-information comprises a categorical label, on several standard data sets in cross-validation experiments. First, we describe the data sets used in the experiments and the experimental procedures. Next, we present baseline results that evaluate the accuracy of MESO, the training and testing time needed, and the effects of the three compression methods described earlier. Finally, to benchmark performance, we compare MESO performance to that of other classifiers, specifically three versions of IND [158], which uses batch training, and HDR [155], which can be configured to use either batch or incremental training.

5.3.1 Data Sets and Experimental Method

Table 5.3 lists the eight data sets used to assess MESO. The number of patterns and features per pattern are shown for each data set, along with the number of distinct labels (or classes) of patterns. Six of the data sets were retrieved from the UCI [159] and KDD [160] machine learning repositories. The exceptions are AT&T faces [161], acquired from AT&T Laboratories Cambridge, and MNIST [162], downloaded from `http://yann.lecun.com/exdb/mnist/`.

Table 5.3: Data set characteristics.

Data Set	Patterns	Features	Labels
Iris	150	4	3
ATT Faces	360	10,304	40
Multiple Feature	2,000	649	10
Mushroom	8,124	22	2
Japanese Vowel	9,859	12	9
Letter	20,000	16	26
MNIST	70,000	784	10
Cover Type	581,012	54	7

These sets represent a wide variety of data types and characteristics. The iris data set [163] comprises just 150 patterns from 3 classes, each class representing a type of iris plant. The classification task is to correctly identify the type of iris by the length and width of the flower's sepals and petals. The AT&T faces data set [161] is also relatively small, comprising 360 images of 40 different human subjects. However, the number of features (each of 10,304 image pixels) is very large. The classification task is to identify the subject of the image from the pixel values.

Three data sets involve numbers and letters. Patterns in the multiple feature data set [164,165] consist of features that describe 10 handwritten numerals extracted from

Dutch utility maps. examples include morphological features, Fourier coefficients, and pixel averages. The classification task is to identify a digit from these features. The MNIST data set [162] also comprises features of handwritten digits, and the task is to identify the digit. However, the features are the 784 integer pixel values, and the number of patterns is much larger than in the multiple feature data set. The letter data set [166] contains 20,000 patterns, each comprising 16 integer measurements of features such as width, height or mean pixel values. The classification task is to classify each pattern as one of the 26 letters in the Latin alphabet.

The mushroom [167] and Japanese vowel [168] data sets are similar in size and feature count, but very different in content. Each pattern in the mushroom data set comprises 22 nominal values (alphabetic characters) that represent mushroom features such as cap shape or gill attachment. Since MESO does not address non-numeric attributes explicitly, each alphabetic character is converted to its numeric ASCII value. The binary label associated with a pattern indicates whether the mushroom is poisonous or edible. The Japanese vowel data set comprises 270 time series blocks, where each block consists of a set of records. Each record comprises 12 continuous measurements of utterances from nine male speakers. The 9,859 patterns are produced by treating each record as an independent pattern and randomizing the data set. As such, no understanding of utterance order is retained. The classification task is to identify the speaker of each utterance independent of its position in a time series.

Finally, the cover type data set [169] comprises 581,012 patterns for determining forest cover type. Each pattern has 54 values, including: 10 continuous values, indi-

cating features such as elevation and slope; 4 binary wilderness areas; and 40 binary soil types. The classification task is to identify which of 7 forest cover types (such as spruce, fir or aspen) corresponds to a test pattern.

We tested MESO using cross-validation experiments as described by Murthy et al. [170]. Each experiment is conducted as follows:

1. Randomly divide the training data into k equal-sized partitions.
2. For each partition, train MESO using all the data outside of the selected partition. Test MESO using the data in the selected partition.
3. Calculate the classification accuracy by dividing the sum of all correct classifications by the total number of patterns tested.
4. Repeat the preceding steps n times, and calculate the mean and standard deviation for the n iterations.

In our tests, we set both k and n equal to 10. Thus, for each mean and standard deviation calculated, MESO is trained and tested 100 times.

5.3.2 Baseline Experiments

Tables 5.4 and 5.5 presents results of cross-validation experiments using MESO to classify patterns in the eight data sets. All tests were started with $\delta = 0.0$ and $c = 0.66$ and executed on a 2GHz Intel Xenon processor with 1.5GB RAM running Linux. Means and standard deviations are provided. Before discussing the results, let us briefly comment on the distance metric used. Since the use of sensitivity spheres effectively divides the larger classification problem into a set of smaller tasks, it turns out that a relatively simple distance metric, such as Euclidean distance, can be used to achieve high accuracy. Euclidean distance is defined as:

$$distance_{Euclidean}(Q, P) \equiv \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2},$$

where Q and P are two patterns of length n. Although we experimented with more complicated distance metrics (e.g., Mahalanobis), none achieved higher accuracy than Euclidean distance, which also exhibited shorter times for training and testing. Therefore, all experiments described here and in later sections use Euclidean distance.

Let us focus first on the results for experiments that do not use compression, shown in Table 5.4. MESO exhibits an accuracy of over 90% on all the data sets, using either sequential or tree based search. MESO's accuracy on the AT&T Faces and MNIST data sets, which contain high-dimensional, image data, indicates that MESO may be effective in computer vision applications. Compared to a sequential search of sensitivity spheres, use of the MESO tree structure reduces training and testing times in most cases. The improvement is particularly notable for large data sets. For MNIST, training time is improved by a factor of 18 and testing time by a factor of 20. For Cover Type, training time is improved by a factor of 18 and testing time by a factor of 12. Although using the hierarchical tree structure reduces the accuracy in most cases, typically between 0% to 4%, this tradeoff may be considered acceptable for applications where decision making is time sensitive.

Next, let us consider the results for experiments using data compression, shown in Table 5.5. The three methods (means, spherical, and orthogonal) had only minimal effect on the three smallest data sets, where sphere growth is inhibited early in the training process, producing spheres with few samples. However, the memory usage

Table 5.4: MESO baseline results comparing a sequential search to MESO tree search.

Data set	Uncompressed	
	(Sequential)	(Tree)
Iris		
Accuracy%	95.5±0.0%	96.1±1.4%
Compression%	0.0%	0.0%
Training (secs)	0.0±0.0	0.03±0.0
Testing (secs)	0.0±0.0	0.0±0.0
ATT Faces		
Accuracy%	97.3±0.0%	94.0±1.4%
Compression%	0.0%	0.0%
Training (secs)	1.85±0.0	1.87±0.0
Testing (secs)	0.39±0.0	0.08±0.0
Mult. Feature		
Accuracy%	95.0±0.0%	94.1±0.5%
Compression%	0.0%	0.0%
Training (secs)	4.58±0.0	1.69±0.0
Testing (secs)	0.99±0.0	0.07±0.0
Mushroom		
Accuracy%	100.0±0.0%	100.0±0.0%
Compression%	0.0%	0.0%
Training (secs)	1.24±0.1	0.62±0.0
Testing (secs)	0.14±0.0	0.05±0.0
Japanese Vowel		
Accuracy%	93.1±0.2%	91.5±0.3%
Compression%	0.0%	0.0%
Training (secs)	0.30±0.0	0.39±0.1
Testing (secs)	0.04±0.0	0.05±0.0
Letter		
Accuracy%	93.1±0.2%	90.6±0.3%
Compression%	0.0%	0.0%
Training (secs)	1.83±0.2	1.27±0.0
Testing (secs)	0.21±0.0	0.16±0.0
MNIST		
Accuracy%	96.5±0.0%	94.3±0.1%
Compression%	0.0%	0.0%
Training (secs)	1307.22±9.7	70.94±1.1
Testing (secs)	157.32±1.3	6.73±0.2
Cover Type		
Accuracy%	96.3±0.0%	96.1±0.0%
Compression%	0.0%	0.0%
Training (secs)	1974.76±11.8	109.97±0.4
Testing (secs)	227.08±1.4	18.74±0.1

Table 5.5: MESO baseline results comparing different compression methods.

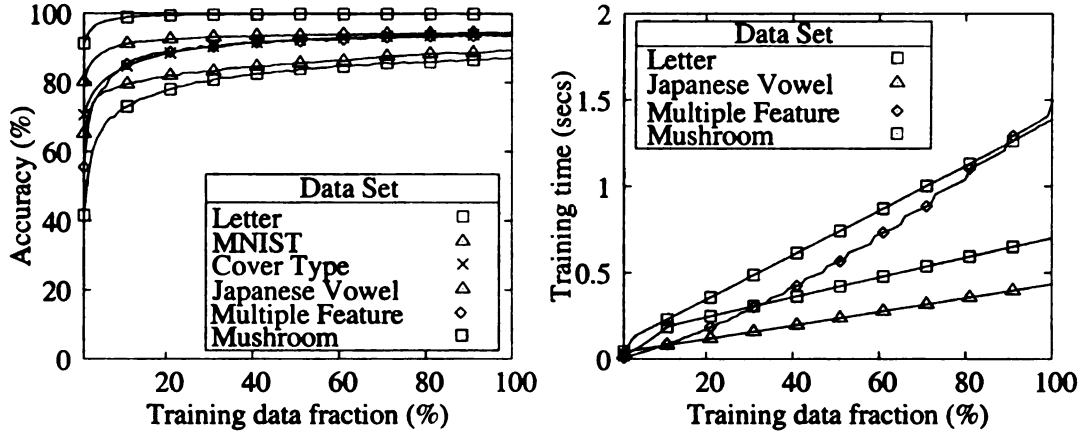
Data set	Compressed (Tree)		
	Means	Spherical	Orthogonal
Iris			
Accuracy%	95.8±0.8%	95.1±1.3%	95.9±2.1%
Compression%	1.86±0.2%	0.0±0.0%	1.9±0.0%
Training (secs)	0.03±0.0	0.03±0.0	0.03±0.0
Testing (secs)	0.0±0.0	0.0±0.0	0.0±0.0
ATT Faces			
Accuracy%	93.5±1.6%	94.5±1.2%	93.7±1.4%
Compression%	0.0±0.0%	0.0±0.0%	0.0±0.0%
Training (secs)	1.90±0.0	1.86±0.0	2.04±0.0
Testing (secs)	0.08±0.0	0.08±0.0	0.08±0.0
Mult. Feature			
Accuracy%	94.2±0.4%	94.1±0.5%	94.4±0.5%
Compression%	0.3±0.0%	0.0±0.0%	0.3±0.0%
Training (secs)	1.73±0.0	1.81±0.0	1.78±0.0
Testing (secs)	0.07±0.0	0.07±0.0	0.07±0.0
Mushroom			
Accuracy%	100.0±0.0%	99.8±0.0%	99.9±0.0%
Compression%	90.2±0.0%	73.9±0.3%	90.2±0.0%
Training (secs)	0.67±0.0	0.81±0.0	0.77±0.0
Testing (secs)	0.05±0.0	0.05±0.0	0.05±0.0
Japanese Vowel			
Accuracy%	81.3±0.4%	90.2±0.3%	81.3±0.2%
Compression%	93.7±0.0%	28.3±0.2%	93.8±0.0%
Training (secs)	0.41±0.0	0.89±0.0	0.49±0.0
Testing (secs)	0.03±0.0	0.05±0.0	0.03±0.0
Letter			
Accuracy%	87.8±0.3%	90.1±0.2%	87.8±0.3%
Compression%	88.6±0.2%	23.6±0.2%	88.3±0.2%
Training (secs)	1.42±0.0	2.28±0.0	1.77±0.0
Testing (secs)	0.12±0.0	0.17±0.0	0.12±0.0
MNIST			
Accuracy%	93.3±0.1%	94.3±0.1%	93.3±0.1%
Compression%	86.5±0.0%	0.0±0.0%	86.5±0.0%
Training (secs)	73.35±1.3	179.53±5.4	78.65±1.4
Testing (secs)	6.29±0.2	6.81±0.2	6.30±0.2
Cover Type			
Accuracy%	81.6±0.1%	95.2±0.0%	81.6±0.0%
Compression%	98.5±0.0%	50.2±0.0%	98.5±0.0%
Training (secs)	114.54±0.7	232.64±2.5	127.97±0.9
Testing (secs)	11.14±0.1	17.29±.03	11.56±0.1

for these data sets is low. On the other hand, both the means and orthogonal methods were very effective in reducing the memory requirements for the five larger data sets (at least an 85% reduction in all cases), while retaining high accuracy. We attribute this behavior to the application of compression to individual sensitivity spheres, enabling the capture of the n-dimensional structure of the training data while limiting information loss. Spherical compression was the least effective in reducing memory usage; the translation of training patterns from Euclidean to spherical coordinates also adds to the cost of training.

Figure 5.12 shows how MESO's accuracy and training times scale with the size of the training data set. To create these plots, each data set was first randomized and then divided into 75% training and 25% testing data. The training data was further divided into 100 segments. MESO was trained and then tested 100 times. During the first iteration only the first segment was used for training; at each subsequent iteration, an additional segment was added to the training set. This process was repeated 10 times for each data set and the mean values calculated. The mean values are plotted in Table 5.12. As shown, MESO's accuracy increases rapidly during early training, and then slows but continues to improve as training continues. Training time increases linearly with respect to the size of the training data set.

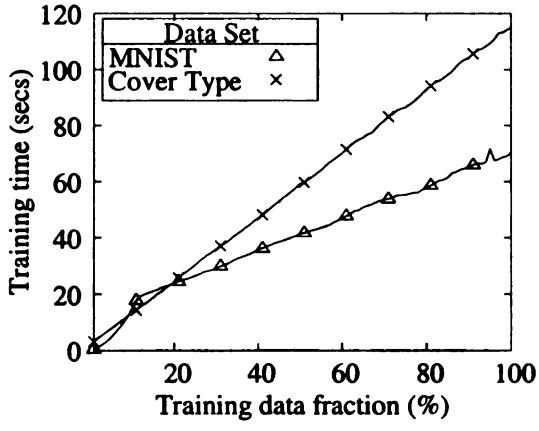
5.3.3 Comparison with Other Classifiers

In this section, we compare MESO performance with that of the IND [158] and HDR [155, 171] classifiers. We note that MESO is trained incrementally, whereas



(a) Accuracy

(b) Training time (small sets)



(c) Training time (large sets)

Figure 5.12: Scalability with respect to training set size. For all data sets, typical standard deviations are less than 10% with respect to the corresponding mean accuracies and training times.

IND can be trained only in batch mode. Classifiers that are batch trained typically have an advantage over those that are trained incrementally: processing the entire training data set when building a classifier may produce a better data model than that produced by incremental training. Therefore, batch training often yields higher accuracy and faster testing.

Table 5.6 compares the MESO results, shown in Table 5.4, with those measured for the IND and HDR classifiers. The implementation of IND, written in

C, was provided by W. Buntine, formerly with NASA's Bayesian Learning Group (http://ic.arc.nasa.gov/ic/projects/bayes-group/ind). The HDR implementation, which uses both C and C++, was provided by J. Weng of the Embodied Intelligence Laboratory at Michigan State University (http://www.cse.msu.edu/ei). MESO is implemented in C++. IND can be used to build a decision tree style classifier using several different algorithms. We tested three different algorithms: CART [172], ID3 [173] and Bayesian [174]. We conducted two sets of experiments with HDR, one using batch training and the other using incremental training.

Let us first compare the MESO results with those of IND. As shown, despite its use of incremental training, MESO accuracy compares favorably with that of all three IND variations, exhibiting higher accuracy in almost all cases. The NC designation indicates that IND could not complete a particular test. Specifically, for the AT&T Faces data set, insufficient memory prevented IND from completing the data set encoding process, which must be done before IND is trained. Somewhat surprisingly, MESO exhibits high accuracy for the Mushroom data set. This data set consists entirely of nominal values, which have no comparative numeric value since they simply indicate characteristics, such as cap shape, by name. IND, like many decision tree algorithms [175], addresses the issue by designating some features as nominal. MESO does not explicitly address nominal values, but still accurately classifies these patterns.

Next, let us consider the training and testing times of MESO relative to those of IND. Although MESO exhibits slower testing times than IND for most data sets, in many cases MESO spends less time training, which would help to reduce the overhead

Table 5.6: Accuracy, training and test times of IND and HDR (compare with Table 5.4).

	IND (Batch)			HDR	
Data set	CART	ID3	Bayesian	(Batch)	(Incremental)
Iris					
Accuracy %	92.8 ±0.3%	93.5 ±0.7%	94.2 ±1.1%	96.4 ±1.4%	89.5 ±3.6%
Training (secs)	0.0±0.6	0.0±0.0	0.01±0.0	0.0±0.0	0.0±0.0
Testing (secs)	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0
ATT Faces					
Accuracy %				94.8 ±1.7%	93.1 ±1.9%
Training (secs)	NC	NC	NC	9.0±0.2	1.8±0.1
Testing (secs)				0.3±0.0	0.3±0.0
Mult. Feature					
Accuracy %	93.1 ±0.6%	94.2 ±0.2%	94.4 ±1.1%	95.2 ±0.4%	88.8 ±0.5%
Training (secs)	22.2±0.3	8.6±0.0	19.2±0.2	2.1±0.0	2.5±0.0
Testing (secs)	0.0±0.0	0.0±0.0	0.0±0.0	2.1±0.0	0.5±0.0
Mushroom					
Accuracy %	99.9 ±0.0%	100.0 ±0.0%	100.0 ±0.0%	100.0 ±0.0%	64.6 ±0.6%
Training (secs)	0.7±0.0	0.0±0.0	0.0±0.0	4.5±0.1	4.2±0.1
Testing (secs)	0.0±0.0	0.0±0.0	0.0±0.0	1.4±0.0	1.1±0.0
Japanese Vowel					
Accuracy %	82.3 ±0.3%	84.2 ±0.3%	84.7 ±0.3%	96.3 ±0.2%	84.9% ±0.8%
Training (secs)	82.3±0.3	2.6±0.0	7.0±0.6	0.9±0.0	5.0±0.2
Testing (secs)	0.0±0.0	0.0±0.0	0.0±0.0	0.9±0.0	1.3±0.0
Letter					
Accuracy %	84.4 ±0.3%	87.9 ±0.1%	88.6 ±0.2%	93.4 ±0.1%	86.2 ±1.0%
Training (secs)	5.4±0.1	0.9±0.0	3.0±0.0	5.0±0.1	30.8±0.5
Testing (secs)	0.0±0.0	0.0±0.0	0.1±0.0	0.6±0.0	7.0±0.1
MNIST					
Accuracy %	88.3 ±0.1%	88.1 ±0.1%	89.0 ±0.1%	97.4 ±0.0%	91.2 ±0.8%
Training (secs)	1225.2±48.2	211.9±3.8	565.1±42.4	5887.0±439.0	3997.9±252.4
Testing (secs)	12.7±1.2	10.9±0.1	10.9±0.1	6007.7±158.2	898.1±55.8
Cover Type					
Accuracy %	93.9 ±0.9%	95.2 ±0.2%	94.4 ±0.3%	96.6% †	71.2% †
Training (secs)	414.4±3.0	51.5±0.2	118.9±5.1	41164.0	52755.3
Testing (secs)	0.5±0.0	0.6±0.2	1.0±0.3	15148.0	11600.0

All tests began with $\delta = 0.0$ and $c = 0.6$. Executed on a 2GHz Intel Xenon processor with 1.5GB RAM running Linux. All experiments conducted using cross-validation.
† The Cover Type data set was not completed for either batch or incremental executions of HDR. Neither was completed due to long execution time requirements.

in acquiring and assimilating new experiences in an online decision maker. Moreover, incremental training as provided by MESO is important to autonomic systems that

need to address dynamic environments and changing needs of users.

Finally, let us compare MESO with HDR, which was designed primarily for computer vision tasks. Batch-trained HDR demonstrates slightly higher accuracy than MESO, attributable to HDR's use of discriminant analysis to help select salient features from the training patterns. However, when HDR is trained incrementally, MESO achieves higher accuracy on all eight data sets, including the two image data sets, AT&T Faces and MNIST. Moreover, the training and testing times of MESO are significantly lower than those of HDR in almost all cases. In several cases, the advantage is more than an order of magnitude. Collectively, these results indicate that MESO may be effective in a variety of autonomic applications requiring online decision making.

5.4 Related Work

Research in clustering and pattern classification is a very active field of study [170, 176–178]. Recently, a number of projects have addressed clustering and classification of large data sets, a characteristic of decision making for autonomic software. Tantrum et al. [179] consider model-based refractionation for clustering large data sets. Yu et al. [180] use an hierarchical approach to clustering using support vector machines (SVMs). Kalton et al. [181] address the growing need for clustering by constructing a framework that supports many clustering algorithms. Methods for online clustering and classification have also been explored [182–184]. Like MESO, methods that address large data sets and online learning may provide a basis for a percep-

tual memory system. However, to our knowledge, MESO is the first to consider the combined tradeoffs of data intensity, time sensitivity and accuracy with respect to memory systems within a decision making environment.

Some of the concepts used in MESO are reminiscent of other clustering systems, and in some cases a complementary relationship exists. For example, like MESO, M-tree [185] partitions data objects (patterns) based on relative distance. However, MESO uses an incremental heuristic to grow sensitivity spheres rather than splitting fixed sized nodes during tree construction. Moreover, rather than select database routing objects for directing the organization of the tree, MESO introduces the concept of pivot spheres for this purpose. BIRCH [186] also uses hierarchical clustering while iteratively constructing an optimal representation under current memory constraints. Where BIRCH mainly addresses data clustering when memory is limited, MESO attempts to balance accuracy, compression and training and testing times to support online decision making. MESO may benefit from BIRCH's concept of *clustering features* as an efficient representation of training patterns, while BIRCH may benefit from MESO's approach to growing sensitivity spheres. Data Bubbles [187] focuses on producing a compressed data set representation while avoiding different types of cluster distortion. Its data analysis and representation techniques might enable alternative approaches to representing and compressing sensitivity sphere data in MESO, whereas MESO's growth and organization of sensitivity spheres could provide an efficient data structure for application of these techniques.

Other works have explored the use of statistical methods and pattern classification and clustering techniques in learning systems, including those that enable a system

to learn online through interaction with the physical world. For example, Hwang and Weng [155] developed hierarchical discriminant regression (HDR) and applied it successfully as part of the developmental learning process in humanoid robots. Notably, HDR provides an hierarchical discrimination of features that helps limit the impact of high-dimensional feature vectors, enhancing the ability of the system to correctly classify patterns. However, as shown in Section 5.3, HDR requires significantly more time for training and testing than does MESO. In addition, Ivanov and Blumberg [157] developed the layered brain architecture [120], which was used for the construction of synthetic creatures, such as a "digital dog." That project used clustering and classification methods to construct perceptual models as part of the dog's developmental learning system. A notable aspect of the layered brain project is the use of compression to limit the effect of large training sets on memory consumption and processing power requirements. MESO also uses compression, but applies it to individual sensitivity spheres in order to maintain high accuracy in the face of data loss.

Finally, researchers have applied data clustering and classification methods to other aspects of autonomic computing, such as fault detection and optimization of algorithms. Fox et al. [188] used data clustering to correlate system faults with failing software components. Once the failing components were identified they could be selectively restarted, avoiding a complete system reboot while shortening mean time to recovery. Geurtz et al. [189] considered several machine learning algorithms for identifying if a system is running atop a wired or wireless network. This method enables the autonomous adaptation of the TCP protocol to address dynamic network

conditions. We anticipate that similar systems can use MESO for automated fault detection or optimization when the software is faced with the uncertainty found in dynamic environments.

5.5 Discussion

We have presented a perceptual memory system, called MESO, that uses data clustering techniques to support online decision making in autonomic systems. We showed that, when used as a pattern classifier, MESO can accurately and quickly classify patterns in several standard data sets, comparing favorably to existing classifiers. We postulate that the integration of a perceptual memory into adaptive software may enable better decision making by autonomous systems. Moreover, by enabling the storage and retrieval of sensed environmental stimuli and associated meta-information, decision makers may be better able to address the uncertainty found in autonomic, pervasive and ubiquitous [122,123] computing environments. In the next three chapters, we test this hypothesis.

Chapter 6

Case Study: Adaptive Error Control

To explore the use of MESO to support learning in adaptive software, we conducted a case study involving adaptive error control. Specifically, we used MESO to implement the decision maker in an audio streaming network application, called Xnaut, that adapts to changes in packet loss rate in a wireless network. The Xnaut uses forward error correction (FEC), whereby redundant information is inserted into the data stream, enabling a receiver to correct some losses without contacting the sender for retransmission. In our experiments, the Xnaut decision maker is trained, through interaction with a human user, in how to balance packet loss with bandwidth consumption. Once trained, the Xnaut is allowed to autonomously adapt to changing network conditions as a user roams about a wireless cell.

The remainder of this chapter is organized as follows. Section 6.1 presents background and related work. Section 6.2 describes our experimental scenario and method.

State maintenance and collateral change are discussed in Section 6.3. Next, Section 6.4 presents the results of our experiments, while Section 6.5 provides an analysis of pattern features to better understand what types of features automated decision makers find most useful. Finally, Section 6.6 concludes this chapter.

6.1 Background and Related Work

This case study complements other studies of *imitative learning*, where a learner acquires skills by observing and remembering the behavior of a teacher. For example, Amit and Matarić [190] used hidden Markov models (HMMs) to enable humanoid robots to learn aerobic-style movements. The ability of the system to reconstruct motion sequences is encouraging, demonstrating the potential importance of imitative learning. Jebar and Pentland [121] conducted imitative learning experiments using a wearable computer system that included a camera and a microphone. A human subject was observed by the system during interactions with other people. The observed training data was used to train an HMM. Later the system was allowed to respond autonomously when presented with visual and audio stimuli, demonstrating a limited ability to reproduce correct responses. However, since learning by observing real human behavior is very complex, even limited recognizable response is significant and promising. This case study complements these approaches by studying perceptual memory to support autonomic decision making for mobile, wireless communication.

In earlier studies, our group has investigated several ways that mobile systems can adapt to changing conditions on wireless networks. Examples include adapt-

able proxies for video streaming [191], adaptive FEC for reliable multicasting [192], several adaptive audio streaming protocols [193, 194], and the design of middleware components whose structure and behavior can be modified at run time in response to dynamic conditions [42, 43]. However, in those approaches, the rules used to govern adaptation were developed in an ad hoc manner as a result of experiments. Here, we investigate whether the system itself can *learn* how to adapt to dynamic conditions.

6.2 Experimental Scenario and Method

In our experimental scenario, depicted in Figure 6.1, a stationary workstation transmits an audio data stream to a wireless access point, which forwards the stream to a mobile receiver over the wireless network. As a user roams about the wireless cell and encounters different wireless channel conditions, the Xnaut should dynamically adjust the level of FEC in order to maintain a high-quality audio stream. However, the Xnaut should also attempt to do so efficiently, that is, it should not consume channel bandwidth unnecessarily.

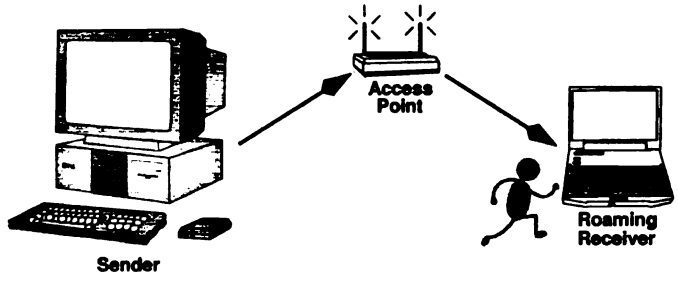


Figure 6.1: Physical network configuration used in the Xnaut case study.

6.2.1 Pattern Features

In the experiments, 56 environmental features are sensed directly, or calculated from other features, and used as input to the decision making process. The features are listed in Table 6.1. The first 4 features are instantaneous sensor readings. Perceived features represent the application's viewpoint. That is, perceived packet loss represents the packet loss as observed by the application after error correction, while real packet loss is the number of packets actually dropped by the network prior to error correction. The second group of 28 features are produced by applying 7 different metrics (mean, standard deviation, etc.) to each of the four directly measured features as sampled over time. The last group of 24 features are produced by calculating 6 Fourier spectrums for each of the four directly measured features.

Table 6.1: Features used for training and testing the Xnaut.

#	Feature Description
1–4	Instantaneous measurements: bandwidth, perceived packet delay, perceived loss and real loss.
5–32	Time-sampled measurements: median, average, average deviation, standard deviation, skewness, kurtosis and derivative.
33-56	Fourier spectrum of the time-sampled measurements: median, average, average deviation, standard deviation, skewness and kurtosis.

The decision maker's goal is to consider these 56 features and autonomously adapt the system to recover from network packet loss while conserving bandwidth. The adaptation is realized by having the receiving node request the sender to modify the

(n, k) settings and change the packet size. The decision maker needs to increase the level of error correction when packet loss rates are high and reduce the level of error correction when packet loss rates are low.

Audio is sampled at 8 KHz using 16-bit samples. Each application-level packet includes a 12-byte application-level header containing a sequence number, stream offset and data length. So, for example, a 32-byte packet contains the header and 10 samples, equivalent to 1.25 milliseconds of audio. A system-level header is prepended to the application-level packet containing 16-bytes of information required for orchestrating FEC code parameter changes. These parameters include an FEC sequence number, FEC packet size and an FEC (n, k) combination. We experimented with larger packet sizes and other (n, k) combinations, but the above values provided sufficient diversity in MESO-based learning and autonomous decision making.

6.2.2 Imitative Learning

In our experiments, the Xnaut decision maker uses MESO to "remember" user preferences for balancing packet loss with bandwidth consumption. The decision maker gains this knowledge through *imitative learning*. A user shows the Xnaut how to adapt to a rising loss rate by selecting an (n, k) setting with greater redundancy. If the new setting reduces the perceived loss rate to an acceptable level, the user reinforces the new configuration (e.g., by pressing a particular key), and the Xnaut uses MESO to associate the sensed environment and selected (n, k) configuration. Later, when operating autonomously, the decision maker senses current environmental conditions and

calculates time-sampled and Fourier features, constructing a pattern. Using this pattern, the Xnaut queries MESO for a system configuration that most likely addresses current conditions. Then, the decision maker emulates the user's actions and adapts the Xnaut, changing the configuration to match that returned from MESO.

Specifically, a user (teacher) shows the Xnaut (learner) how to adapt to a rising loss rate by selecting an FEC code with greater redundancy. If the perceived loss rate then drops to an acceptable level, the user reinforces the Xnaut's new configuration. During this process, the decision maker has two states. In the *training state*, the decision maker observes the teacher, remembering good responses to observed conditions. In the *testing state*, the decision maker autonomously implements responses to current conditions based on what it observed during training.

Figure 6.2 depicts this procedure. We can separate the activities that require human interaction from those that can be completed independently by the decision maker. When the Xnaut is training, the user issues commands to the system that adapt the application to current conditions. Implementing these adaptations also updates the `MessageRepository` to reflect these changes. If the user's actions produce a good software configuration, then the user reinforces this configuration. The decision maker then saves the current system configuration to MESO.

When the Xnaut is testing, the decision maker retrieves the current conditions from the `MessageRepository`, and queries MESO for a system configuration to address current sensed conditions. If the query returns a configuration that differs from the one currently implemented, the decision maker adopts the retrieved configuration and changes the Xnaut's configuration to match that returned by MESO.

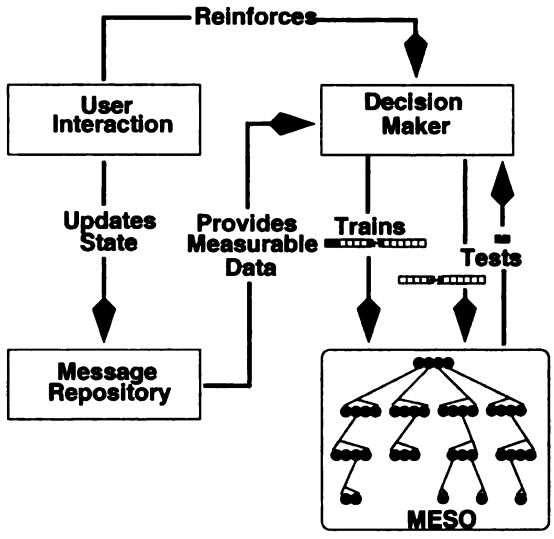


Figure 6.2: High-level diagram of the primary decision making components.

6.3 State Maintenance

During execution, the Xnaut sender and receiver can instantiate new FEC encoders and decoders to exchange with those in current use. However, this exchange must be orchestrated such that data encoded with one (n, k) combination is not decoded with a different (n, k) combination, causing data to be corrupted or lost.

The sequence diagram in Figure 6.3 illustrates how the Xnaut orchestrates encoder/decoder exchange. When the receiver decides that a different FEC code is required to address the current packet loss rate, it transmits a `changeFEC(n,k,pktsize)` request to the sender. The sender receives the request and continues transmitting packets until all n packets encoded with the current FEC code have been sent. The sender then calls `newEncoder(n,k,pktsize)` to instantiate and insert a new encoder using the (n, k) combination and packet size requested by the receiver. When the receiver receives a packet encoded with the new FEC code

(as indicated by the system-level header), it calls `newDecoder(n,k,pktsize)` to instantiate and insert a new decoder. The packet is then passed to the new decoder for processing, and normal packet transmission resumes.

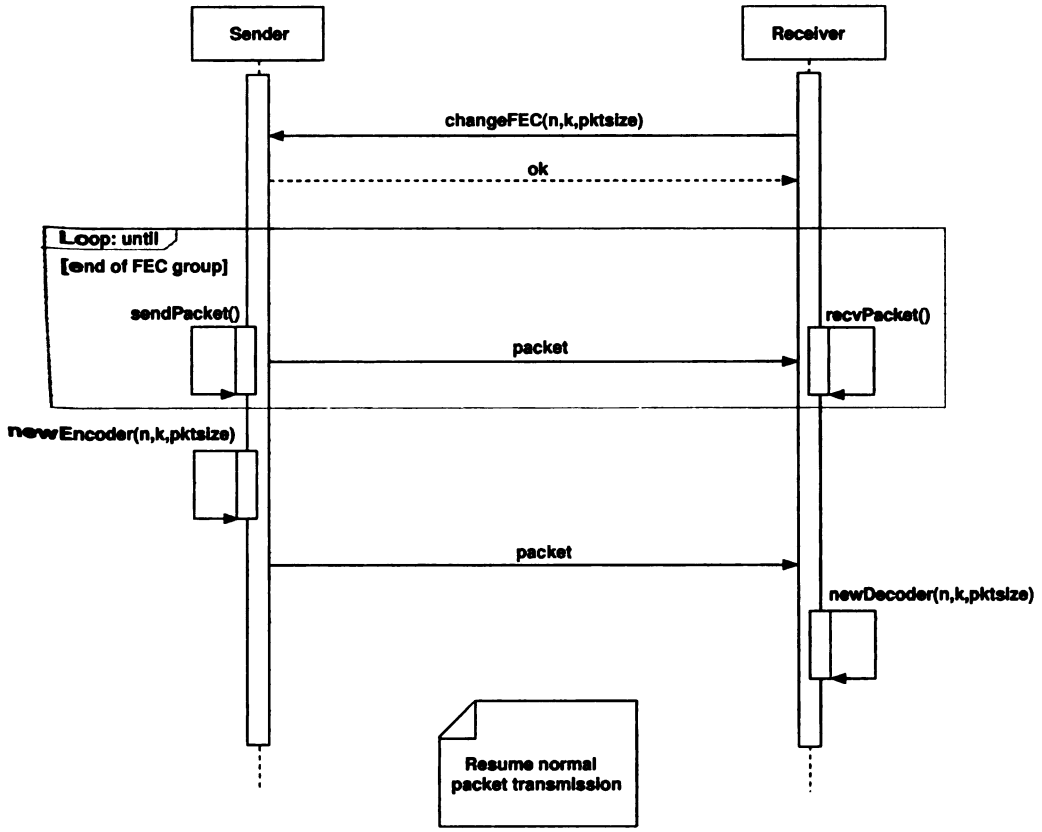


Figure 6.3: A sequence diagram depicting sender/receiver interaction when changing FEC codes.

6.4 Results

We report results for two sets of experiments designed to evaluate the ability of the Xnaut to autonomously balance error control effectiveness and bandwidth consumption. The transmitting station was a 1.5GHz AMD Athlon workstation, and the

mobile receiver was a a 500MHz X20 IBM Thinkpad notebook computer. Both systems run the Linux operating system.

The first set of experiments was conducted in a controlled setting, specifically using a wired network and artificially generated packet losses. These experiments were designed to verify that the Xnaut could learn to respond accurately to a simple loss model. We trained and tested the Xnaut using TCP over a 100Mb wired network, thereby avoiding the effects of spurious errors and overruns of UDP buffers. Packets were dropped at the sender according to a probabilistic loss model, which varied the loss rate from 0.0 to 0.3 in steps of size 0.05, at 15 second intervals. After starting the receiver and sender, the system was trained by having a user select (n, k) values and packet sizes in an attempt to minimize the perceived loss and bandwidth consumption. When a combination satisfying user preferences is found, the Xnaut (receiver) is notified that the current combination is "good" (by pressing the "g" key). Good FEC/packet size combinations and system measurements were then used to train MESO. Training concluded in one hour with MESO storing 34,982 training patterns associated with 6 FEC code combinations:

$$32(10,2) \quad 32(8,2) \quad 64(1,1) \quad 64(4,2) \quad 64(6,2) \quad 64(8,2).$$

In testing, the Xnaut collected system measurements and used them to query MESO for the FEC code/packet size combination associated with the most similar set of measurements observed during training.

Figures 6.4(a) and (b), respectively, show the (artificially generated) network packet loss and the perceived packet loss, during the testing phase of the experi-

ment. All changes to error correction are made autonomously by the Xnaut decision maker. Figure 6.4(c) plots the redundancy-ratio defined as:

$$ratio_{redundancy} \equiv \frac{(n-k)}{n},$$

reflecting the changes in FEC (n, k) values corresponding to the loss rates shown in Figure 6.4(a). For comparison, Figure 6.4(c) also depicts a plot of the optimum redundancy ratio given the FEC codes specified during training. The optimum ratio is computed using the FEC code that provides redundancy greater than or equal to the real loss rate. From these figures, it can be seen that the Xnaut significantly reduces packet loss as perceived by the application by automatically adapting FEC parameters and packet size. Notably, in order to conserve bandwidth the Xnaut did not simply choose a high (n, k) ratio, but changed parameters to correspond with the changing loss rate.

The second set of experiments were conducted using real packet losses on an 11Mbps 802.11b wireless network. The experimental configuration is shown in Figure 6.1. These tests required the Xnaut to autonomously balance real packet loss and bandwidth consumption as a user roamed about a wireless cell. The Xnaut was trained by a user for one hour using an artificial loss rate that varied from 0.0 to 0.6 in steps of size 0.05 at 15 second intervals. Such a model allowed the Xnaut to be trained for the higher loss rates often found at the periphery of a real wireless cell. Training generated 32,709 training patterns in 10 classes that were used to train MESO for autonomous testing atop a wireless network. Each class "label" is a FEC

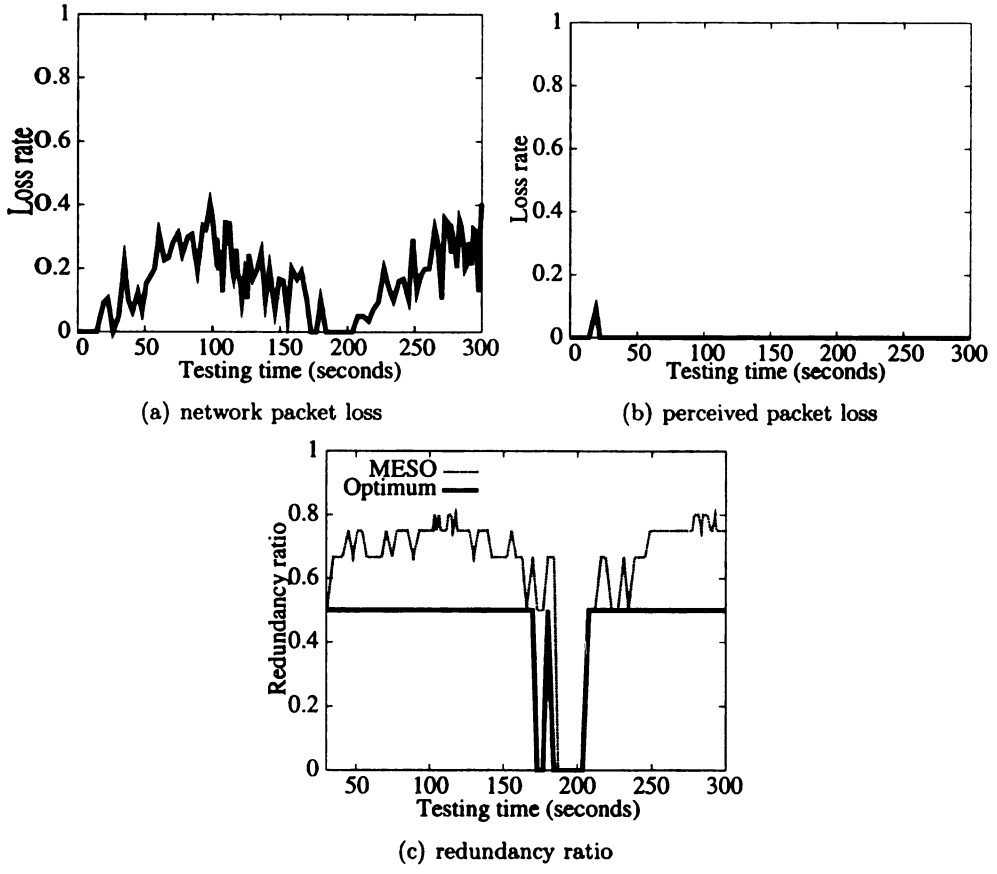


Figure 6.4: Xnaut results for artificially generated packet losses.

configuration specifying a (n, k) pair and a packet size. The 10 classes (packet size / FEC code combinations) were:

$$\begin{array}{lllll} 32(10,2) & 32(12,2) & 32(14,2) & 32(16,2) & 32(18,2) \\ 32(8,2) & 64(1,1) & 64(4,2) & 64(6,2) & 64(8,2). \end{array}$$

In the testing phase, we turned off simulation and enabled the Xnaut to autonomously balance real packet loss and bandwidth consumption. The sender was located on a stationary workstation connected to a wireless access point through a 100Mb hub. A wireless PCMCIA card provided network access to the notebook computer. The UDP/IP multicast protocol was used for transmission of the data stream.

Data was collected as a user roamed about a wireless cell carrying a notebook running an Xnaut receiver. Again, all changes to error correction were made autonomously by the Xnaut decision maker. Figure 6.5 shows the results, using the same format as in the earlier tests. Under real conditions, the Xnaut is able to significantly reduce loss rate as perceived by the application, while conserving bandwidth under good channel conditions.

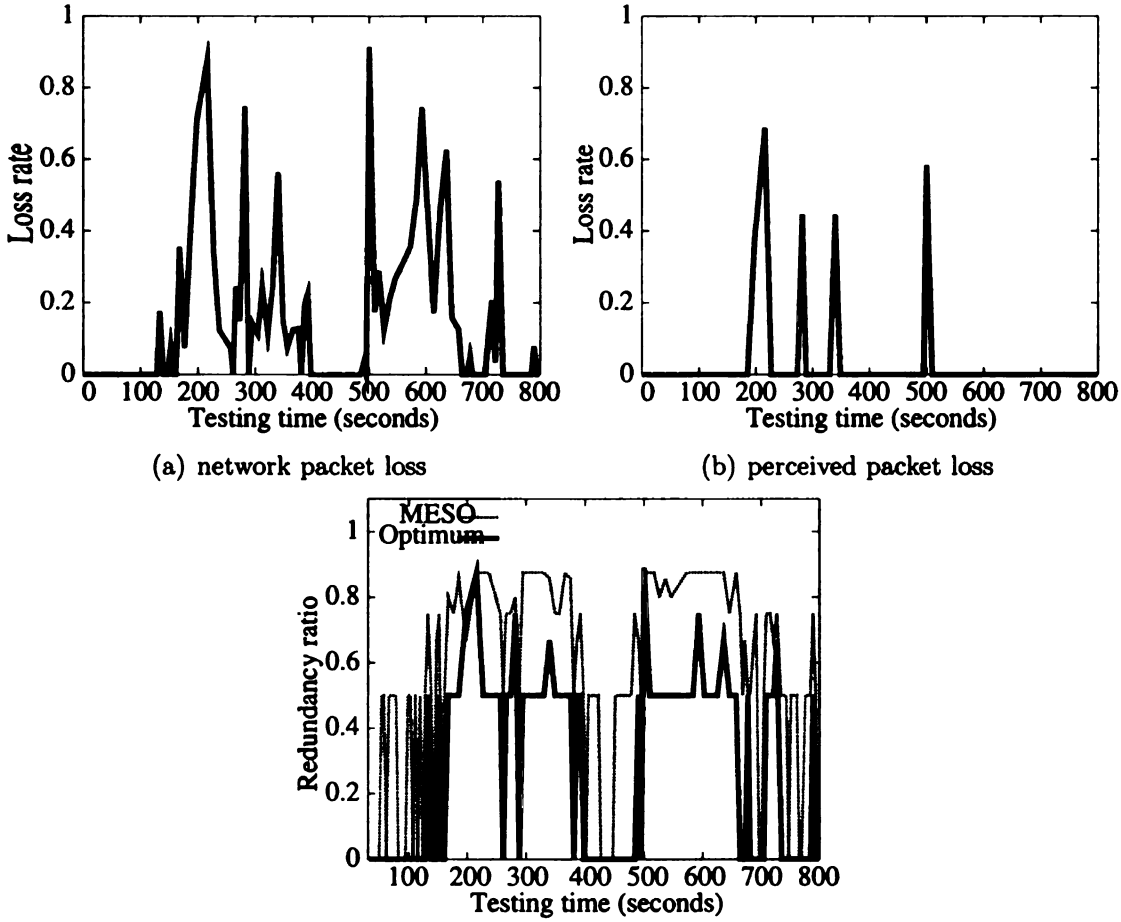


Figure 6.5: Xnaut results for real packet losses on a wireless network.

We further tested MESO using cross-validation experiments (described in Chapter 5.3.1). In our tests, we set both k and n equal to 10. Thus, for each mean and standard deviation calculated, MESO is trained and tested 100 times. Table 6.2 shows

Table 6.2: Xnaut results with and without compression.

Data set	Uncompressed	Means	Spherical	Orthogonal
Xnaut				
Accuracy%	94.1%±0.2%	87.7%±0.2%	92.4%±0.7%	87.3%±0.4%
Compression%	0.0%	91.8%±0.1%	5.8%±0.2%	91.8%±0.14%
Training (s)	33.096±2.123	18.059±0.3	74.705±6.5	19.359±0.582
Testing (s)	1.127±0.016	1.024±0.013	1.119±0.014	1.024±0.010

Data set size is 32,709. Executed on a 2GHz Intel Xenon processor with 1.5GB RAM running Linux. All experiments conducted using cross-validation.

results from running cross-validation tests using the data acquired during Xnaut training. This data was produced during training for autonomous Xnaut operation on the real wireless network. This table shows accuracy, with and without compression, helping quantify how well the Xnaut can be expected to imitate a user. The system achieved 94% accuracy without compression, and maintained an accuracy level above 87% even when data was compressed by over 90%. We regard these results as promising and justifying further study of MESO for on-line decision making in autonomic systems.

6.5 Feature Analysis

Using cross-validation, we designed an experiment to determine which of the 56 features were most significant for classifying the patterns used by the Xnaut decision maker. Each experiment is conducted as follows:

1. Starting with the first pattern feature, iteratively select a feature and remove it from the set of patterns.
2. Run a cross-validation experiment and calculate the classification accuracy.
3. If classification accuracy has been reduced by greater than 2%, retain the feature and add it back into the set of patterns. Otherwise, do not retain the feature for use in subsequent cross-validation runs.

4. Repeat the preceding steps until all features have been tested.

In our tests, we set the cross-validation parameters, k and n, both equal to 10. Thus, for each mean and standard deviation calculated, MESO is trained and tested 100 times.

As shown in Table 6.3 (continued on the next page), our analysis of pattern features revealed that only 4 features accounted for approximately 91.7% of the accuracy attained when using all 56 features. Moreover, these 4 features were statistical metrics, such as skewness and kurtosis, calculated over multiple sensor readings. We attribute the significance of these 4 features to the utility of statistical metrics for characterizing distributions of sensor readings. However, sensor readings taken from dynamic environments are unlikely to produce a single stationary distribution, but rather are likely to transition between many distributions. Detecting sensor reading transitions may enable a decision maker to better recognize when current environmental conditions have changed that may require the software to adapt.

6.6 Discussion

In this chapter we presented a case study that demonstrates that perceptual memory can support decision making in dynamic environments. As a user roamed about a wireless cell, the Xnaut decision maker autonomously exchanged FEC encoders and decoders to balance packet loss with bandwidth consumption. Moreover, MESO demonstrated high accuracy, with and without compression, when classifying patterns comprising the 56 features sensed or computed by the Xnaut. These results indicate

Table 6.3: Feature contribution to MESO accuracy.

Kept	Accuracy	Description
	91.4%±0.5%	bandwidth
	91.6%±0.5%	perceived delay
	91.5%±0.5%	real loss
	91.6%±0.5%	perceived loss
	91.5%±0.5%	bandwidth median
	91.5%±0.5%	bandwidth mean
	91.5%±0.6%	bandwidth avg. dev.
	91.4%±0.5%	bandwidth std. dev.
	91.2%±0.5%	bandwidth skewness
	91.4%±0.4%	bandwidth kurtosis
	91.4%±0.5%	bandwidth derivative
	91.5%±0.5%	bandwidth FFT median
	91.3%±0.6%	bandwidth FFT mean
	91.4%±0.5%	bandwidth FFT avg. dev.
	90.7%±0.5%	bandwidth FFT std. dev.
	90.8%±0.5%	bandwidth FFT skewness
	89.5%±0.6%	bandwidth FFT kurtosis
	89.6%±0.6%	perceived delay median
	89.6%±0.5%	perceived delay mean
	89.6%±0.5%	perceived delay avg. dev.
	89.5%±0.6%	perceived delay std. dev.
	89.4%±0.5%	perceived delay skewness
	88.0%±0.5%	perceived delay kurtosis
	88.0%±0.6%	perceived delay derivative
	88.0%±0.5%	perceived delay FFT median
	88.0%±0.6%	perceived delay FFT mean
	88.0%±0.6%	perceived delay FFT avg. dev.
	87.2%±0.6%	perceived delay FFT std. dev.

Classification accuracy when all 56 features are used is 91.3%±0.5%. Using only the 4 most significant features accuracy is 83.7%±0.6%. Accuracy column indicates accuracy after that feature was removed.

that perceptual memory can play an important role in the design and implementation of autonomic decision makers. However, an analysis of the 56 pattern features revealed that only 4 features accounted for most of MESO's accuracy. Moreover, these 4 features were statistical metrics calculated using multiple sensor readings, indicating a preference for metrics that describe the underlying distribution of sensor readings.

Table 6.3: (cont'd)

Kept	Accuracy	Description
	87.1%±0.6%	perceived delay FFT skewness
	86.1%±0.6%	perceived delay FFT kurtosis
	86.0%±0.6%	real loss median
	86.1%±0.5%	real loss mean
	86.1%±0.6%	real loss avg. dev.
	85.9%±0.6%	real loss std. dev.
✔	82.4%±0.7%	real loss skewness
	88.1%±0.5%	real loss kurtosis
	88.2%±0.6%	real loss derivative
	88.0%±0.5%	real loss FFT median
	87.5%±0.5%	real loss FFT mean
	87.4%±0.6%	real loss FFT avg. dev.
✔	61.6%±0.9%	real loss FFT std. dev.
	86.8%±0.6%	real loss FFT skewness
✔	75.7%±0.6%	real loss FFT kurtosis
	86.6%±0.5%	perceived loss median
	86.6%±0.6%	perceived loss mean
	86.5%±0.6%	perceived loss avg. dev.
	86.5%±0.6%	perceived loss std. dev.
	86.3%±0.6%	perceived loss skewness
	86.2%±0.6%	perceived loss kurtosis
	86.0%±0.6%	perceived loss derivative
	85.8%±0.6%	perceived loss FFT median
	85.6%±0.7%	perceived loss FFT mean
	85.2%±0.6%	perceived loss FFT avg. dev.
	84.8%±0.6%	perceived loss FFT std. dev.
	83.7%±0.6%	perceived loss FFT skewness
✔	78.2%±0.7%	perceived loss FFT kurtosis

Chapter 7

Automated Ensemble Extraction and Analysis of Acoustic Data Streams

The main contribution of this chapter is to introduce a process that enables detection of transitions and extraction of meaningful sequences, called *ensembles*, from sensor data streams. We define ensembles as time series sequences that recur, though perhaps rarely. We couple MESO and Dynamic River to address decision making related to pipeline processing. Using Dynamic River, we implement our technique for automated extraction of ensembles that can be processed by MESO for detection and classification of sensed events. Here we apply this method to support automated detection and classification of bird species using acoustic data streams collected in natural environments. *Classification* attempts to accurately recognize which species

produced a particular vocalization, while *detection* indicates the likelihood that an acoustic clip contains a song voiced by a particular species. The effort is a collaboration between computer scientists and researchers at the Computational Ecology and Visualization Laboratory (CEVL) [195] at Michigan State University. Acoustic data is collected from in field sensor stations located at the Kellogg Biological Research Station (KBS) [196] and other locations in Michigan. Species classification and detection enables the automation of ecological surveys [197–199], traditionally done by human observers in the field. Moreover, processing of data as it is collected enables annotation of sensor data with meta information that can facilitate analysis and enable autonomic decision making. For instance, detection of a specific species may trigger a decision maker to increase sensor sampling rate or invoke specialized processing. Although this target application is relatively specific, the process employed is general and can be extended to other problem domains such as mobile communication and military reconnaissance.

As shown in Figure 7.1(a), the acoustic sensor stations comprise a pole-mounted sensor unit and a solar panel coupled with a deep cycle battery for providing power over extended periods. Figure 7.1(b) shows the internal components of the sensor unit. Each sensor unit contains a Crossbow Stargate processing platform [200] equipped with a microphone and an 802.11b wireless interface card. The Stargate platform has a 400MHz Intel PXA225 processor and 64MB of RAM. The operating system used is TinyOS [201]. Acoustic clips are collected by the sensor units and transmitted over a wireless network to a laptop in a protected enclosure, for temporary storage and later relay to the CEVL. Currently, clips are approximately 30 seconds long and are

collected every half hour. We anticipate increasing the collection rate as computing, storage and power resources permit.

(a) Sensor station (b) Sensor unit

Figure 7.1: Acoustic sensor station and closeup of a Stargate sensor.

Sensor collection of acoustic data facilitates monitoring of natural environments despite visual occlusions, such as trees or buildings, or even darkness. Moreover, microphones can collect data from all directions simultaneously. However, acoustic data is rich and complex. For instance, bird vocalizations vary considerably even within a particular bird species. Young birds learn their songs with reference to adult vocalizations during sensitive periods [202–204]. At maturity, the song of a specific bird will crystallize into a species-specific stereotypical form. However, even stereotypical songs vary between individual birds of the same species [205]. Moreover, many vocalizations are not stereotypical but are instead plastic, and may change when

sung or due to seasonal change, while some species can learn new songs throughout their lives [206]. Variation of song within a species and the occurrence of other sounds in natural settings, such as the sound of wind or that produced by human activity, is a significant obstacle to automated detection and classification of birds. Extraction of candidate bird vocalizations from acoustic streams enables accurate recognition of a species where misidentifying one species as another should be avoided.

Our technique for ensemble extraction uses Dynamic River operators to construct a pipeline for detecting anomalies in sensor data streams. When an anomaly is detected, downstream operators respond by extracting an ensemble. Once extracted, each ensemble is further processed and then converted into a set of patterns suitable for processing by MESO. Results of the experiments presented later in this chapter are promising and suggest that the proposed methods for automated monitoring of natural environments are effective. Moreover, the processes employed are general and can be extended to other problem domains.

The remainder of this chapter is organized as follows. Section 7.1 describes background on the components of the ensemble extraction method. Section 7.2 describes in detail the approach for ensemble extraction, and Section 7.3 presents the results of our experiments using ensemble extraction for classification and detection of bird species. Section 7.4 presents related work. Finally, in Section 7.5, we summarize and discuss the contribution of this chapter.

7.1 Background

This section provides background on processing of acoustic and other time series data streams. First, in Section 7.1.1, we describe the methods we use for visualizing acoustic clips. In Section 7.1.2 and 7.1.3, respectively, we briefly review piecewise aggregate approximation (PAA) [207,208] and symbolic aggregate approximation (SAX) [5] and describe the benefits of using PAA representation and how SAX bitmaps can be used for anomaly detection.

7.1.1 Visualizing Acoustic Events

Figure 7.2 depicts two common methods for visualizing an acoustic clip, such as those collected by sensor stations at the KBS. The top graph shows a plot of the signal's amplitude normalized by subtracting the mean and scaling by the maximum amplitude. The bottom graph shows the same clip plotted as an acoustic spectrogram. A spectrogram depicts frequency on the vertical axis and time on the horizontal axis. Shading indicates the intensity of the signal at a particular frequency and time. Spectrograms are useful for visualizing acoustic signals in the frequency domain. Moreover, spectral representations can be used for automated classification and detection of acoustic events. In this study, for example, spectrogram segments are distilled into signatures that can be used to identify the bird species that produced a particular vocalization.

Figure 7.3 depicts a block diagram of a pipeline for constructing a spectrogram. This process uses the discrete Fourier transform [209] to compute the frequency domain power spectra. To plot a spectrogram, the acoustic data is first divided into

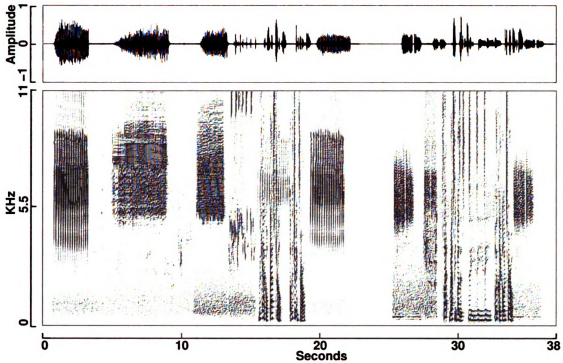


Figure 7.2: Top, an oscillogram (normalized) of an acoustic signal. Bottom, a spectrogram of the same acoustic signal.

equal sized segments and then filtered using a Welch window [138] to mitigate edge effects (or leakage) between segments. Then the discrete Fourier transform is used to compute a frequency domain representation of each segment. Computing the complex absolute value converts the complex representation used by the Fourier transform to a real representation of signal intensity. Finally, each segment is plotted to produce a spectrogram.

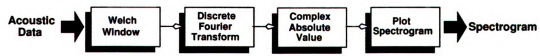


Figure 7.3: A block diagram depicting the operators required to produce a spectrogram.

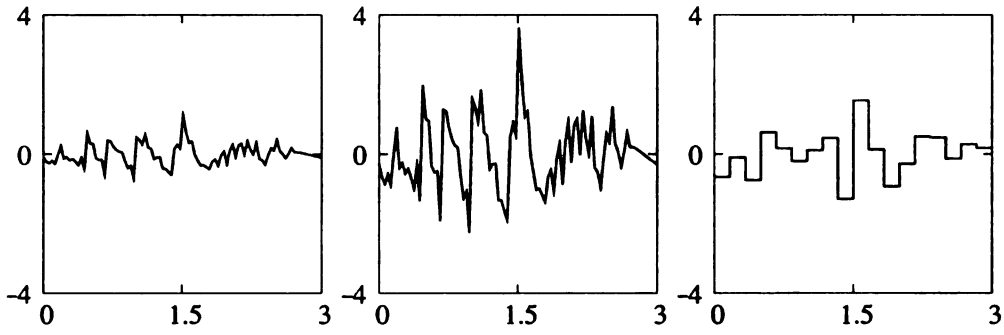
7.1.2 Piecewise Aggregate Approximation

Piecewise aggregate approximation (PAA) was introduced by Keogh et al. [207], and independently by Yi and Faloutsos [208], as a means to reduce the dimensionality of time series. For completeness an overview of PAA is presented here; full details can be found in the papers cited. As shown in Figure 7.4, an original time series sequence, Q, of length n is converted to PAA representation, $\overline{Q}$. First, Q is Z-normalized [210] as follows:

$$\forall i \quad q_i = \frac{q_i - \mu}{\sigma},$$

where μ is the vector mean of original signal, σ is corresponding standard deviation and q_i is the i^{th} element of Q. Second, Q is segmented into $w \leq n$ equal sized subsequences, and the mean of each subsequence computed. $\overline{Q}$ comprises the mean values for all subsequences of Q. Thus, Q is reduced to a sequence $\overline{Q}$ with length w. Each i^{th} horizontal segment of the plot shown in Figure 7.4(c) represents a single element, q_i, of $\overline{Q}$. Thus, the complete PAA algorithm first Z-normalizes Q and then computes the segment means to construct $\overline{Q}$, as depicted in Figure 7.4(c).

Z-normalization and conversion to PAA representation facilitate detection and classification in two ways. First, detection and classification using acoustics in natural environments is often impeded by variance in signal strength due to distance from the sensor station or differences between individual vocalizations. Z-normalization converts two signals that vary only in magnitude to two identical signals, enabling comparison of signals of different strength. Second, conversion to PAA representation



(a) Original signal. (b) Z-normalized signal. (c) Z-normalized signal with PAA.

Figure 7.4: Example signal and results of Z-normalization and subsequent PAA processing.

helps smooth the original signal to facilitate comparison of vocalizations. That is, during classification or detection, signals are typically represented as vectors of values, called *patterns*. For acoustics, many pattern values may represent noise or sounds other than those voiced by a bird. These values do not contribute usefully when using distance metrics, such as Euclidean distance, for pattern comparison. PAA smoothes intra-signal variation and reduces pattern dimensionality, while Z-normalization helps equalize similar acoustic patterns that differ in signal strength.

Figure 7.5 depicts the spectrogram shown in Figure 7.2 after conversion to PAA representation. This spectrogram was constructed by applying PAA to the frequency data comprising each column of the original spectrogram. Despite smoothing and reduction using PAA, these spectrograms are similar in appearance, demonstrating the potential utility of using PAA representation.

As mentioned, we use distance metrics for comparing patterns. For comparison of patterns that have not been reduced using PAA, Euclidean distance can be used. Computing the distance between two patterns reduced using PAA is similar to com-

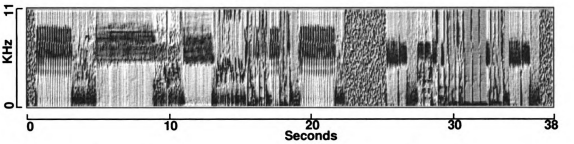


Figure 7.5: Spectrogram of the acoustic signal (see Figure 7.2) after conversion to PAA representation (stretched vertically for clarity).

puting Euclidean distance. PAA distance is defined as:

$$distance_{PAA}(\overline{Q}, \overline{P}) \equiv \sqrt{\frac{n}{w}} \sqrt{\sum_{i=1}^{w} (\overline{q_i} - \overline{p_i})^2},$$

where $\overline{Q}$ and $\overline{P}$ are two patterns reduced using PAA. The terms n and w are the lengths of the original patterns and those after PAA reduction, respectively. PAA distance has been shown to be a tight lower bound on Euclidean distance [207], providing a close estimate of Euclidean distance between the original two patterns despite PAA dimensionality reduction.

7.1.3 Symbolic aggregate approximation

Extending the benefits of PAA is a representation introduced by Lin et al. [5] called Symbolic Aggregate approXimation (SAX). The purpose of SAX is to enable accurate comparison of time series using a symbolic representation. As shown in Figure 7.6, SAX converts a sequence from PAA representation to symbolic representation, where each symbol (we use integers as symbols, others have used alphabetic characters) appears with equal probability based on the assumption that the distribution of time

series subsequences is Gaussian [5]. Thus, each PAA segment is assigned a symbol by dividing the Gaussian probability distribution into α equally probable regions, where α is the alphabet size ($\alpha = 5$ in Figure 7.6). Each PAA segment falls within a specific Gaussian region and is assigned the corresponding symbol.

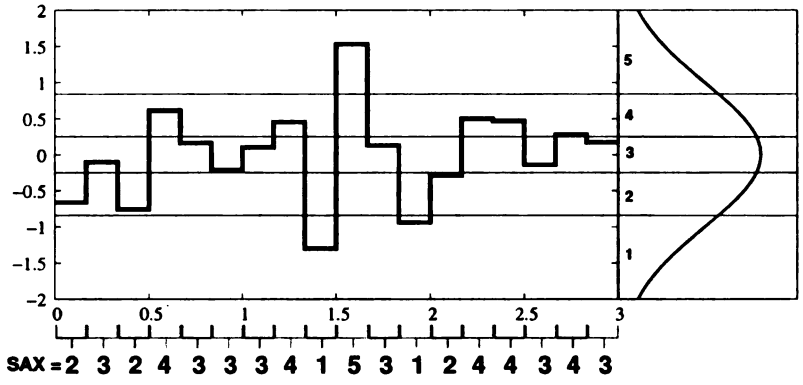


Figure 7.6: Conversion of the example PAA processed signal converted to SAX (adapted from [5]).

Kumar et al. [6] proposed *time series bitmaps* for visualization and anomaly detection in time series. SAX bitmaps are constructed by counting occurrences of symbolic subsequences of length n (e.g., 1, 2 or 3 symbols). Each bitmap can be represented using an n-dimensional matrix, where each cell represents a specific subsequence. An example is shown in Figure 7.7; using subsequences of length $n = 2$, matrix cell $(1, 1)$ contains the count and frequency with which the subsequence $1, 1$ occurs. Frequencies are computed by dividing the subsequence count by the total number of subsequences. For visualization as a bitmap, each cell is assigned a color according to the cell's value. An anomaly score can be computed by comparing two concatenated bitmap matrices using Euclidean distance. The matrices are constructed using two concatenated sliding windows. For each anomaly score computed, both windows are moved for-

ward along the time series one time step and the corresponding matrices recomputed. The distance between the matrices is computed and reported as an anomaly score. Greater distances indicate significant change in the time series. As further discussed in Section 7.2, we use SAX bitmap matrices to compute an anomaly score for acoustic signals, enabling the extraction of bird vocalizations and other acoustic events.

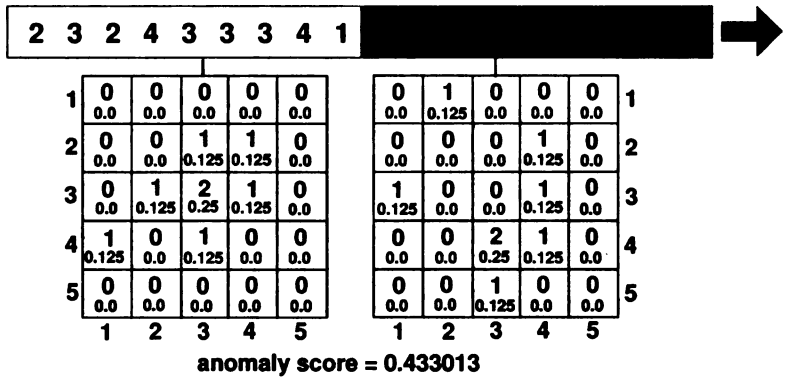


Figure 7.7: Using SAX bitmaps to compute an anomaly score for a signal (see [6] for more information). Number of subsequence occurrences shown over frequency.

7.2 Ensemble Extraction and Processing

A sensor data stream is a time series comprising continuous or periodic sensor readings. Typically, readings taken from a specific sensor can be identified and each reading appears in the time series in the order acquired. Online clustering or detection of interesting sequences benefits from time-efficient, distributed processing that extracts finite candidate sequences from the original time series. Moreover, clustering time-series data using sliding windows has been shown to be ineffective [211], prompting the need for research in ways to extract sequences that can be usefully

clustered. Our goal is to extract potentially recurring sequences that can be used for data mining tasks such as classification or detection.

As noted earlier, we define *ensembles* as time series sequences that recur, though perhaps rarely. Notably, this definition is similar to other time series terms. For instance, a *motif* [212–215] is defined as a sequence that occurs frequently and a *discord* [216] is defined as the sequence that is least similar to all other sequences. A notable limitation for finding a discord in a time series is that the time series must be finite. Our use of ensembles addresses this limitation by using a finite window for computing an anomaly score and thereby detecting a distinct change, or transition, in time series behavior. An anomaly score greater than a specified threshold is considered as indicating the start of an ensemble that continues until the anomaly score falls below the threshold.

Figure 7.8 depicts a typical approach to data acquisition and analysis using a Dynamic River pipeline that targets ecosystem monitoring using acoustics. The interested reader may refer to Appendix B for further information on the Dynamic River operators described in this section. First, audio clips are acquired by a sensor platform and transmitted to a `readout` operator that encapsulates clips as records and then writes the records to `record` for storage. Although additional record processing is possible prior to storage, it is often desirable to retain a copy of the raw data for later study. During analysis, a data `feed` is invoked to read clips from storage and write them to `wav2rec` to encapsulate acoustic data (WAV format in this case) in pipeline records. The remaining operators comprise the process for extracting ensembles and processing them for classification or detection using MESO.

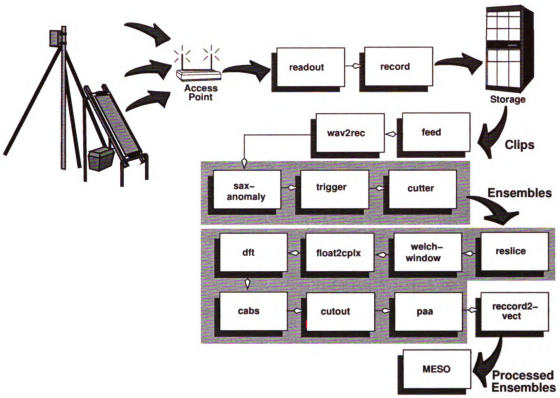


Figure 7.8: Block diagram of pipeline operators for converting acoustic clips into ensembles for detection of bird species.

The pipeline segment, ⇒`[saxanomaly|trigger|cutter]`, transforms records comprising acoustic data into ensembles. The incoming record stream is scoped, with each clip delimited by an `OpenScope/CloseScope` pair. The outgoing record stream comprises ensembles that are also delimited by an `OpenScope/CloseScope` pair. The clip and ensemble scopes are typed, using the `scope_type` record header field, as `scope_clip` or `scope_ensemble` respectively.

The moving average of the SAX anomaly score, as described in Section 7.1.3, is output by `saxanomaly` in addition to the original acoustic data. Parameters, such as the SAX anomaly window size, SAX alphabet size and a moving average window size, can be set to meet the needs of a particular application or data set. The SAX

anomaly window size specifies the number of samples to use for constructing each concatenated matrix used for computing the SAX anomaly score, for a given SAX alphabet. The moving average window size specifies the number of anomaly scores to use for computing a mean anomaly score that is output by **saxanomaly**. Using a moving average smoothes anomaly score "spikes" over a longer period that can be used as a window of anomalous behavior by the **cutter** operator. In our experiments with environmental acoustics, we set the moving average window to 2250 samples, the SAX anomaly window to 100 samples and the SAX alphabet size to 8. Figure 7.9 plots the anomaly score computed by the **saxanomaly** pipeline operator for the signal depicted in Figure 7.2. As shown, larger anomaly scores are computed when time series behavior is changing.

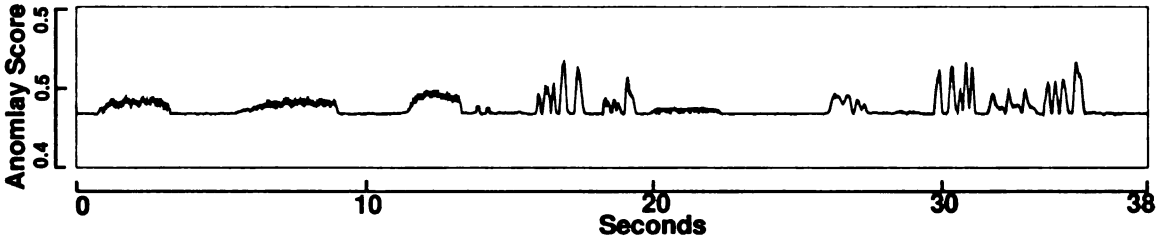


Figure 7.9: Anomaly score generated for the acoustic signal shown in Figure 7.2.

Figure 7.10 depicts the trigger signal output by the **trigger** operator (top) and the corresponding ensembles extracted from the original acoustic signal by the **cutter** operator (bottom). The **trigger** operator transforms the anomaly score output by **saxanomaly** into a trigger signal that has the discrete values of either 0 or 1. The **trigger** operator is adaptive in that it incrementally computes an estimate of the mean anomaly score, μ_0, for values when the trigger value is 0. **Trigger** emits a value of 1 when the anomaly score is more than 5 standard deviations from μ_0 and a

0 otherwise. The number of standard deviations is specific to the particular data set or application.

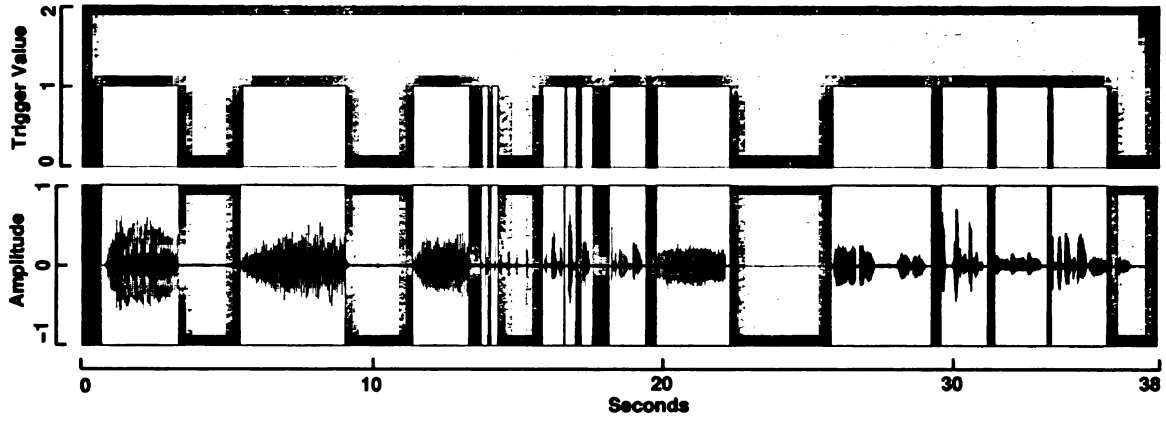


Figure 7.10: Trigger signal and ensembles extracted from the acoustic signal shown in Figure 7.2.

The `cutter` operator reads both the records containing the original acoustic signal and the records emitted by `trigger`. When the trigger signal transitions from 0 to 1, `cutter` emits an `OpenScope` record, designating the start of an ensemble, and begins composing an ensemble. Each ensemble comprises values from the original acoustic signal that correspond to when the trigger value is 1. When the trigger value transitions from 1 to 0, `cutter` emits a `CloseScope` record, and resumes consuming acoustic values until the trigger value again transitions to 1. The record stream, as emitted from `cutter`, comprises clips that contain one or more ensembles.

The pipeline segment, ⇒`[reslice|welchwindow|float2cplx|dft|cabs]`, transforms the amplitude data of each ensemble into a frequency domain (power spectrum) representation in a way similar to that used for producing a spectrogram. First, for each pair of ensemble records, the `reslice` operator constructs a new record comprising the last half of the first record and the second half of the second original record.

This new record is then inserted into the record stream between the two original records. "Reslicing" ensemble records is a method similar to that used by Welch's method [138] for minimizing variance when computing power spectral density (PSD) using finite length periodograms. The remainder of the pipeline segment, starting with `welchwindow`, computes a floating point representation of each ensemble's spectrogram, where each ensemble comprises one or more records of spectral data. If desired, each record could be plotted as a single column of pixels in a spectrogram.

Next, each record of each ensemble is passed to the `cutout` operator. The `cutout` operator selects specific frequency ranges from each record and emits records comprising only these ranges. Data outside of the selected range is discarded. For our classification experiments, the frequency range ≈[1.2kHz,9.6kHz] was extracted using `cutout`. Frequencies above and below this range typically have little data useful for classification or detection of bird species. Moreover, data below this range typically comprises low frequency noise, including the sound of wind and sounds produced by human activity. For our detection experiments, discussed in Section 7.3, `cutout` was used to further reduce the frequency range for better detection of specific species.

The optional `paa` operator reduces each record to a PAA representation as discussed in Section 7.1. For our experiments, we used records that were either reduced by a factor of 10 using PAA or that were not reduced. The effectiveness of using PAA representation for smoothing acoustic spectral data is demonstrated in Section 7.3. Finally, the `record2vect` operator converts pipeline records to vectors of floating point values (patterns), suitable for use in our classification and detection experiments with MESO.

7.3 Assessment

Listed in Table 7.1 are the four-letter species codes and the common names for the 10 bird species whose vocalizations we use in our experiments. Also listed are the number of individual patterns and ensembles extracted from the recorded vocalizations and included in our experimental data sets. For testing classification accuracy, we used four data sets produced from a set of audio clips, and each extracted ensemble contains the vocalization from one of the 10 bird species. Although each ensemble contains the vocalization for only a single species, the clips typically contain other sounds such as those produced by wind and human activity.

Table 7.1: Bird species codes, common names and the number of patterns and ensembles used in the experiments discussed in Section 7.3.

Code	Common name	Patterns	Ensembles
AMGO	American goldfinch	229	42
BCCH	Black capped chickadee	672	68
BLJA	Blue Jay	318	51
DOWO	Downy woodpecker	272	50
HOFI	House finch	223	26
MODO	Mourning dove	338	24
NOCA	Northern cardinal	395	42
RWBL	Red winged blackbird	211	27
TUTI	Tufted titmouse	339	59
WBNU	White breasted nuthatch	676	84

7.3.1 Data Sets and Methodology

Ensemble data sets. Two ensemble data sets, comprising 473 ensembles, were produced using the method described in Section 7.2. The data sets differ in that one was processed with PAA while the other was not. The ensembles produced by

the `cutter` operator were validated by a human listener as being a bird vocalization. The validated ensembles were then fed to the `dft` operator for further processing (refer to Figure 7.8). Each ensemble comprises one or more patterns. Each pattern was constructed by merging 3 frequency domain records. A single pattern represents 0.125 seconds of acoustic data in the range $\approx$[1.2kHz,9.6kHz] and comprises either 1050 features or, when processed with PAA, 105 features. A voting approach is used for testing each ensemble, specifically each pattern belonging to a given ensemble is tested independently and represents a "vote" for the species indicated by the test. The species with the most votes is returned as the recognized species.

Pattern data sets. Each of the two pattern data sets comprises 3,673 patterns extracted from the 473 ensembles in the ensemble data sets. Like the ensemble data sets, each pattern has either 1050 or 105 features and represents 0.125 seconds of acoustic data. Ensemble grouping is not retained and, as such, recognition is based on testing with a single pattern.

Experimental method and assessment. We tested classification accuracy using cross-validation experiments as described by Murthy et al. [170] using a leave-one-out approach [154]. The leave-out-out approach was used due to the high variability found in bird vocalizations and the relatively small size of the data sets. Each experiment is conducted as follows:

1. Randomize the data set. For the ensemble data set, randomize the order of the ensembles. For the pattern data set, randomize the order of the patterns.
2. In turn select each ensemble/pattern as a test pattern, train MESO using all remaining data. Test MESO using the single selected ensemble/pattern.

3. Calculate the classification accuracy by dividing the sum of all correct classifications by the total number of ensemble/patterns.
4. Repeat the preceding steps n times, and calculate the mean and standard deviation for the n iterations.

In our leave-one-out tests, we set n equal to 20. Thus, for each mean and standard deviation calculated, MESO is trained and tested 9,460 times in the case of the ensemble data set and 73,500 times in the case of the pattern data set.

We also executed a resubstitution test, where MESO was both trained and tested using the entire data set. Although lacking statistical independence between training and testing data, resubstitution affords an estimate of the maximum classification accuracy expected for particular data set. Each experiment is conducted as follows:

1. Randomize the data set. For the ensemble data set, randomize the order of the ensembles. For the pattern data set, randomize the order of the patterns.
2. Train and test MESO using all ensembles/patterns.
3. Calculate the classification accuracy by dividing the sum of all correct classifications by the total number of ensemble/patterns.
4. Repeat the preceding steps n times, and calculate the mean and standard deviation for the n iterations.

In our resubstitution tests, we set n equal to 100. Thus, for each mean and standard deviation calculated, MESO is trained and tested 100 times for both the pattern and ensemble data sets.

7.3.2 Classification Results

Table 7.2 summarizes the accuracies and timing results for the four bird song data sets. Resubstitution is greater than 92% accurate for all data sets while leave-one-out results are somewhat less accurate. Given that bird vocalizations are highly variable and that data set sizes are relatively small, we can consider these results promising.

Table 7.2: MESO classification and timing results.

	Data set			
	Pattern	Ensemble	PAA Pattern	PAA Ensemble
Accuracy%				
Leave-one-out	71.5%±0.9%	76.0%±1.1%	80.4%±0.3%	82.2%±0.9%
Resubstitution	92.3%±3.1%	96.3%±2.8%	94.7%±0.8%	97.2%±1.2%
Timing (s)				
Training	57.7±1.1	56.1±1.7	57.7±1.1	56.1±1.7
Testing	57.7±1.9	58.6±2.8	57.7±1.9	58.6±2.8

Accuracy experiments were conducted using cross-validation with the leave-one-out approach and resubstitution. Timing experiments were run using the entire data set for both training and testing. For the PAA data sets, timing results include the time required for conversion to PAA representation. Timing tests were executed on a 2GHz Intel Xenon processor with 1.5GB RAM running Linux.

Shown in Table 7.3 is the confusion matrix [176,217] for classification using PAA patterns and the leave-one-out approach. Matrix columns are labeled with the species predicted by MESO, while rows are labeled with the species that actually produced the original vocalization. The main diagonal (in bold) indicates the percentage of patterns correctly classified. Other cells indicate the percentage of patterns confused with other species. For instance, the intersection of the row labeled AMGO with the column labeled BLJA indicates that 4.66% of blue jay patterns were confused with the American goldfinch. As shown, most patterns are correctly classified, with the northern cardinal most likely to be classified correctly while the American goldfinch is most likely to be confused with another species.

Table 7.4 shows the confusion matrix for classification using PAA ensembles and the leave-one-out approach. Again, most ensembles are correctly classified. Moreover, ensemble classification is typically more accurate than classification using individual patterns. However, the black capped chickadee and the mourning dove are notable

Table 7.3: Confusion matrix for classification using individual PAA patterns.

	Predicted									
	AMGO	BCCH	BLJA	DOWO	HOFI	MODO	NOCA	RWBL	TUTI	WBNU
AMGO	**62.9**	5.5	4.7	0.4	2.0	9.6	1.6	3.4	3.4	6.4
BCCH	2.9	**78.4**	2.7	1.0	5.9	0.6	1.5	1.2	2.8	3.0
BLJA	3.4	6.5	**79.4**	0.9	1.3	1.5	2.8	1.2	0.8	2.1
DOWO	0.2	5.9	1.0	**86.4**	2.2	0.0	0.2	0.1	2.9	0.9
HOFI	1.6	7.4	2.8	2.1	**75.3**	0.6	2.5	0.2	5.5	2.1
MODO	5.6	0.9	1.5	1.0	0.6	**81.6**	1.1	1.0	0.9	5.8
NOCA	3.4	1.7	2.0	0.3	1.8	0.8	**87.6**	0.5	0.7	1.2
RWBL	2.6	3.3	1.8	0.7	1.3	0.9	0.6	**84.3**	2.9	1.6
TUTI	2.9	5.8	2.9	0.6	7.1	0.2	1.8	1.0	**74.7**	3.1
WBNU	4.5	1.9	1.6	0.3	1.2	3.1	1.0	1.0	2.0	**83.3**

exceptions and are misclassified more frequently than when testing with individual patterns. Using ensembles, the red winged blackbird is most likely to be classified correctly, while the mourning dove is most likely to be confused with a different species.

Table 7.4: Confusion matrix for classification using ensembles comprising PAA patterns.

	Predicted									
	AMGO	BCCH	BLJA	DOWO	HOFI	MODO	NOCA	RWBL	TUTI	WBNU
AMGO	**70.3**	7.8	0.5	1.5	0.5	3.8	2.8	4.5	1.7	6.6
BCCH	5.2	**69.2**	4.3	2.5	4.4	0.1	2.6	3.7	2.9	5.2
BLJA	2.1	3.5	**86.0**	0.5		3.4	1.7	0.5	0.2	2.2
DOWO		5.5	0.5	**90.5**	1.1		0.1	0.1	2.2	
HOFI	2.9	1.2	2.3	3.9	**79.3**		6.6		3.7	0.2
MODO	7.6	1.6	1.8	3.7	4.1	**67.0**	6.4	3.1		4.7
NOCA	6.0	0.1	0.1		0.3	0.1	**90.8**	0.6		2.0
RWBL	0.9	0.5		2.8		0.5	0.5	**94.7**	0.2	
TUTI	2.2	2.6	0.7		2.1		1.1		**90.5**	0.9
WBNU	3.4	0.3	0.1	1.2		4.8	2.4	0.4	1.2	**86.1**

Values are percentages with empty cells indicating 0%

7.3.3 Species Detection

The goal of species detection is to indicate that the bird song for a specific bird species is present in an acoustic clip. Detection should maximize the detector's true-positive rate while holding false-positives (where other sounds are identified as the target species) to an acceptably low level. Species detection is useful for automating ecological surveys and for annotating sensor data, with metadata, to ease identification of candidate data sets for study [14, 23, 28]. More generally, detection can be used to alert human operators or automated decision makers of important events. For instance, acoustics has been used for the detection of railroad car bearing failure [218].

Experimental method. For our detection experiments we divided the ensemble and pattern data sets into a training and testing set. The two training sets comprise the patterns (or ensembles) for a single species. For our experiments we used either the black capped chickadee or the white breasted nuthatch for training. The two corresponding testing sets comprise all the patterns for the remaining 9 species after occurrences of the training species had been removed. Each experiment is conducted as follows:

1. Randomize the training set. For the ensemble data set, randomize the order of the ensembles. For the pattern data set, randomize the order of the patterns.
2. Select 10% of the training set and add it to the testing set. Remove the selected patterns/ensembles from the training set.
3. Train MESO using the remaining data in the training set. Test MESO using all the data in the testing set, reporting whether a test ensemble/pattern is correctly identified as the target species.
4. Repeat the preceding steps n times, and calculate the true-positive and false-positive rates over all n iterations.

In our tests, we set n equal to 100. Thus, for each true- and false-positive rate calculated, MESO is trained and tested 100 times. This process was repeated for each of 200 detector settings. Each detector setting specifies a proportion, ρ, of the MESO sensitivity sphere δ grown during training. If the distance between a test pattern and the closest sphere mean is $\leq \rho\delta$, then the test pattern is considered as indicating the presence of the target species and the detector returns true. Otherwise, the pattern is rejected and the detector returns false. We varied ρ over the interval $[0.0, 2.0]$ in steps of 0.01 and calculated the true- and corresponding false-positive rates for each setting. When testing using ensembles, a voting method is again used where the target species is reported as detected only if 50% or more of the votes are for that species.

Detector assessment. Receiver operating characteristic (ROC) curves [219] have been used for evaluating machine learning and pattern recognition techniques [220, 221] when the cost of error is not uniform. ROC curves plot the false-positive rate against the true-positive rate where each point on the curve represents a different setting of detector parameters. As such, if the cost of incorrect detection is high, a detector setting is needed that will hold the false-positive rate low even at the cost of failing to detect the target species in many clips. However, since clips are regularly produced by each sensor platform, failure to detect the target species in some clips will likely be compensated for during subsequent detector operation. Moreover, precision [217] is also computed using the number of true- and false-positives. Precision is the rate of true-positives to the total number of positive predictions, and is defined

as:

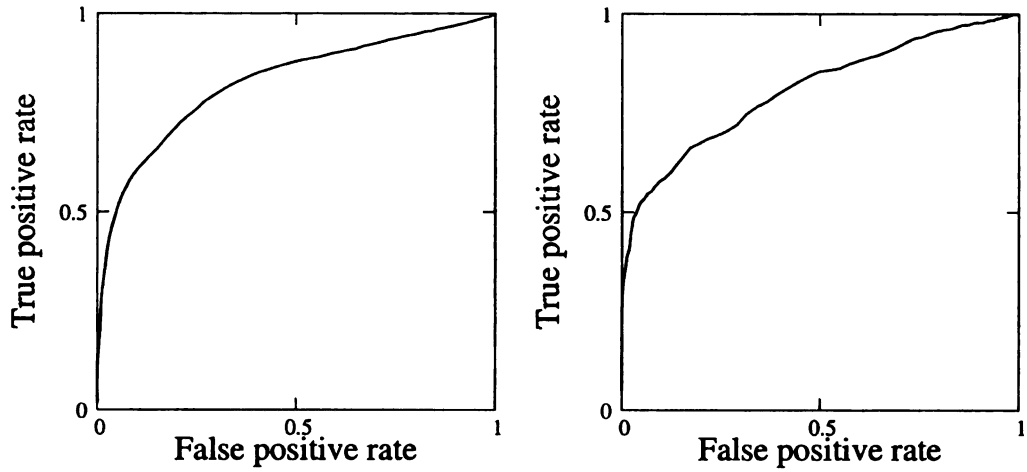
$$precision = \frac{TP}{FP + TP},$$

where TP and FP are the number of true- and false-positive predictions respectively. A score of 1.0 indicates perfect precision, and occurs where there are no false-positives. A larger false-positive count reduces precision.

Figure 7.11 depicts ROC curves for detecting the black capped chickadee and the white breasted nuthatch. For the black capped chickadee, detection is approximately 50% true-positive when the false-positive rate is approximately 4% using either patterns or ensembles. Detection of the white breasted nuthatch is approximately 50% true-positive with a corresponding 1% false-positive rate using either patterns or ensembles. These rates are promising for detection of species in real environments, where many other sounds may be heard. Moreover, continuous clip collection yields future opportunities for detection of a species even when current environmental conditions make detection difficult.

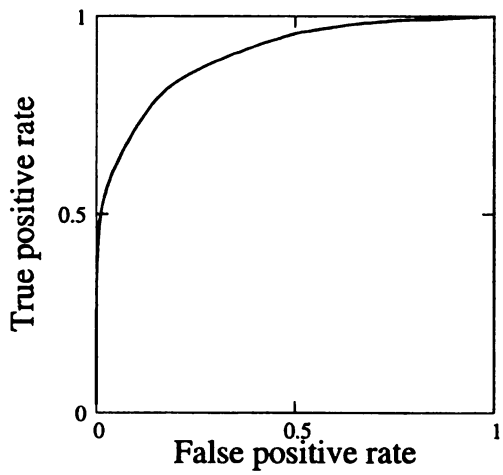
As shown in Figure 7.12, further insight can be gleaned by plotting a ROC curve together with precision. A semi-log scale magnifies their relationship. For the black capped chickadee, the best true-positive rate attained, while maintaining a precision of $\geq$0.9, is approximately 10% and 9% for patterns and ensembles, respectively. Similarly for the white breasted nuthatch, the best true-positive rates attained with a precision of $\geq$0.9 are approximately 28% and 33%.

High variance is particularly notable in the ROC curve shown in Figure 7.11(b)

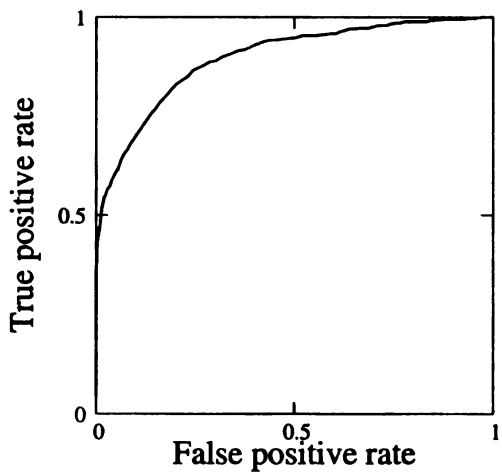


(a) BCCH ROC curve using patterns.

(b) BCCH ROC curve using ensembles.



(c) WBNU ROC curve using patterns.

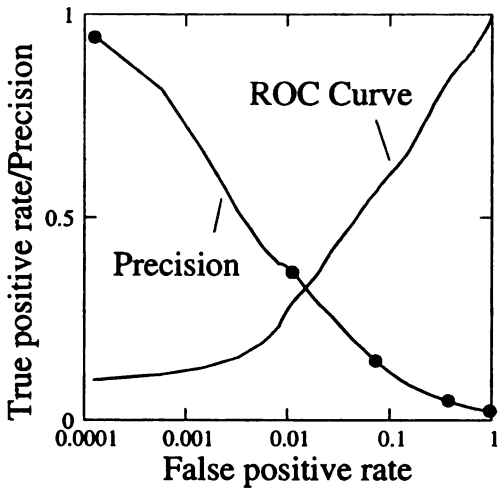


(d) WBNU ROC curve using ensembles.

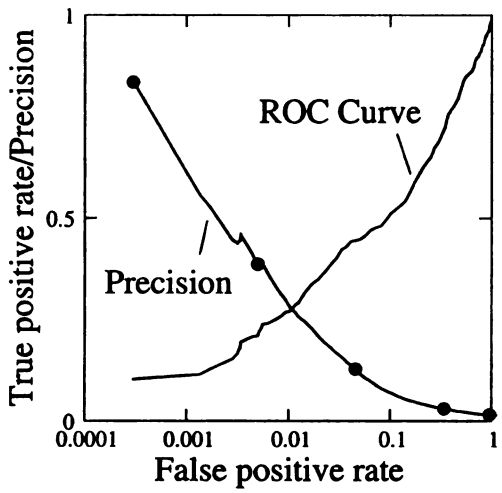
Figure 7.11: ROC curves for detection of the black capped chickadee (BCCH) and white breasted nuthatch (WBNU).

and 7.12(b). This variance is in part due to the small size of the data sets with respect to the variability found in bird vocalizations. Moreover, a significant proportion of the frequency range used may not be useful for detection of the target species. For detection, we can further reduce pattern dimensionality and target a specific frequency range for each species being detected.

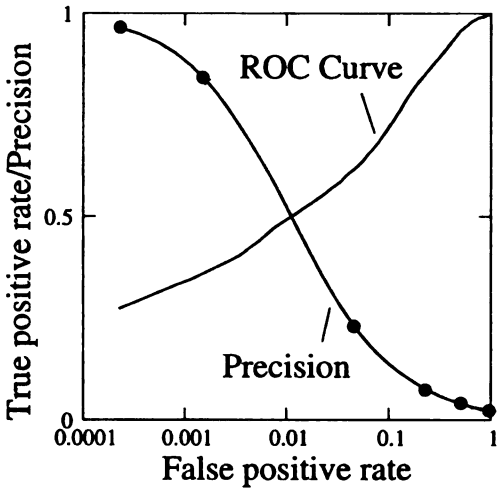
Power spectral density (PSD) histograms plot the power, or intensity, of an acous-



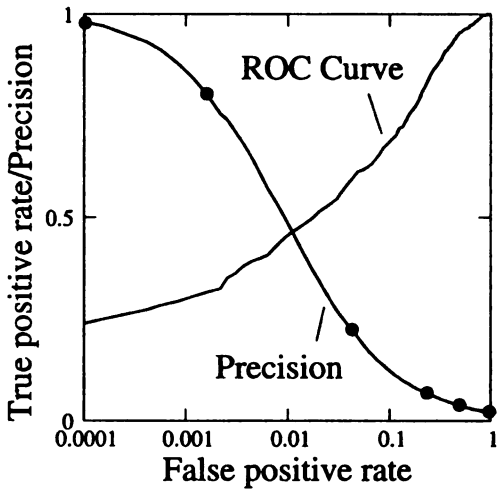
(a) BCCH ROC curve and precision using patterns.



(b) BCCH ROC curve using ensembles.



(c) WBNU ROC curve and precision using patterns.



(d) WBNU ROC curve using ensembles.

Figure 7.12: Semi-log scale ROC curves and precision for detection of the black capped chickadee (BCCH) and white breasted nuthatch (WBNU).

tic signal at different frequencies. Welch's method [138] is a common approach for estimating the PSD of acoustic or other types signals across an entire frequency band. Concisely, Welch's method estimates the PSD of a signal by dividing the original signal into equal sized overlapping sequences. The Fourier transform is then applied to each sequence and the power of the signal computed at each frequency. The average

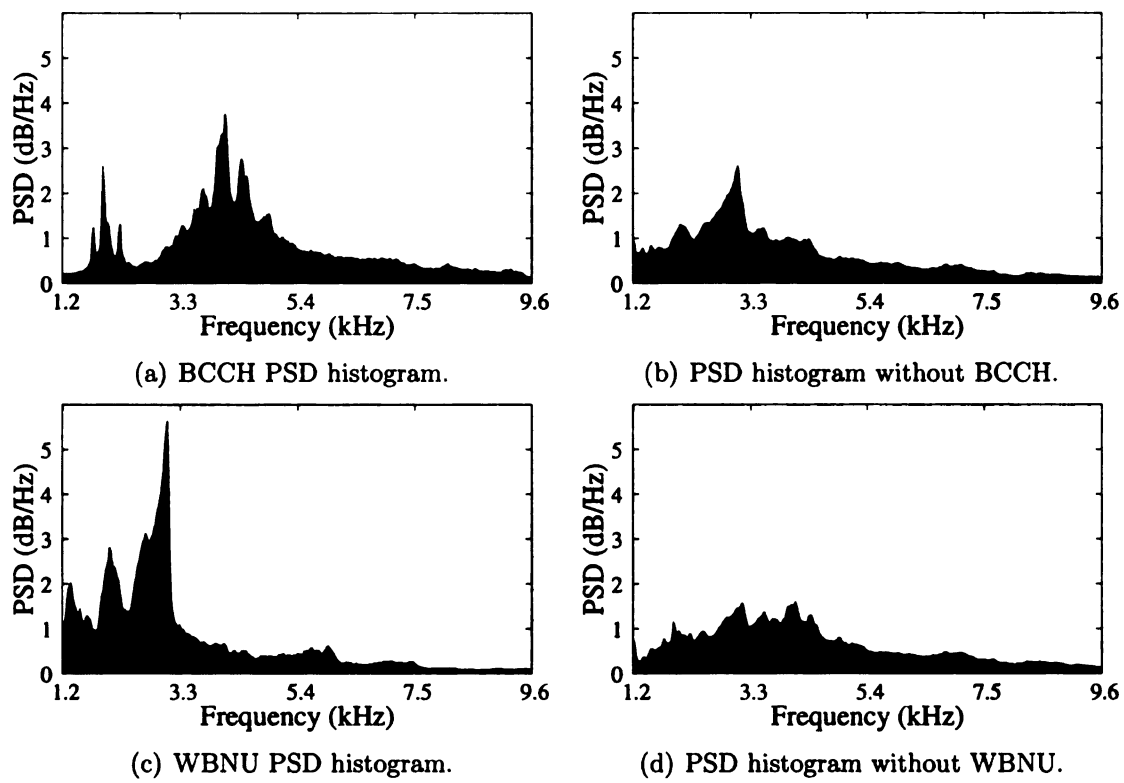
power at each frequency is then computed across all sequences.

Figure 7.13[1] shows PSD histograms for the black capped chickadee and the white breasted nuthatch. For comparison, PSD histograms are also shown for the other 9 bird species remaining after removal of the respective target species. These histograms show that the most useful frequency ranges for detecting a black capped chickadee or white breasted nuthatch in the presence of the other species is afforded by the ranges $\approx$[1.2kHz,6.0kHz] and $\approx$[1.2kHz,4.8kHz] respectively. Thus, the number of pattern features can be further reduced without increasing the false-positive rate. Moreover, the removal of unneeded features may reduce variability in detection rates and improve ROC curve quality.

Figure 7.14 shows the ROC curves for detection experiments using limited frequency ranges. The ranges $\approx$[1.2kHz,6.0kHz] and $\approx$[1.2kHz,4.8kHz] correspond to 60 and 45 pattern features respectively when using PAA. These curves have similar shape and interpretation as those in Figure 7.11 but are smoother indicating a reduction of detector variance. However, when ensembles are used for detection, the ROC curves still show significant variability, indicating that more training data would be valuable for improving detection and further reducing variance.

Figure 7.15 plots the ROC curves and precision using a semi-log scale for limited frequency ranges. For the black capped chickadee, the best true-positive rate attained while maintaining a precision of $\geq$0.9 is approximately 9% and 6% for pat-

[1]Decibels (dB) is a ratio of the difference between a reference signal and the signal under consideration. For measures of intensity dB is defined as $10\log(\frac{S}{S_0})$ where S_0 is the reference signal and S is the signal under consideration. The PSD plots are computed as dB of intensity where S_0 is the normalized mean signal.



(a) BCCH PSD histogram.

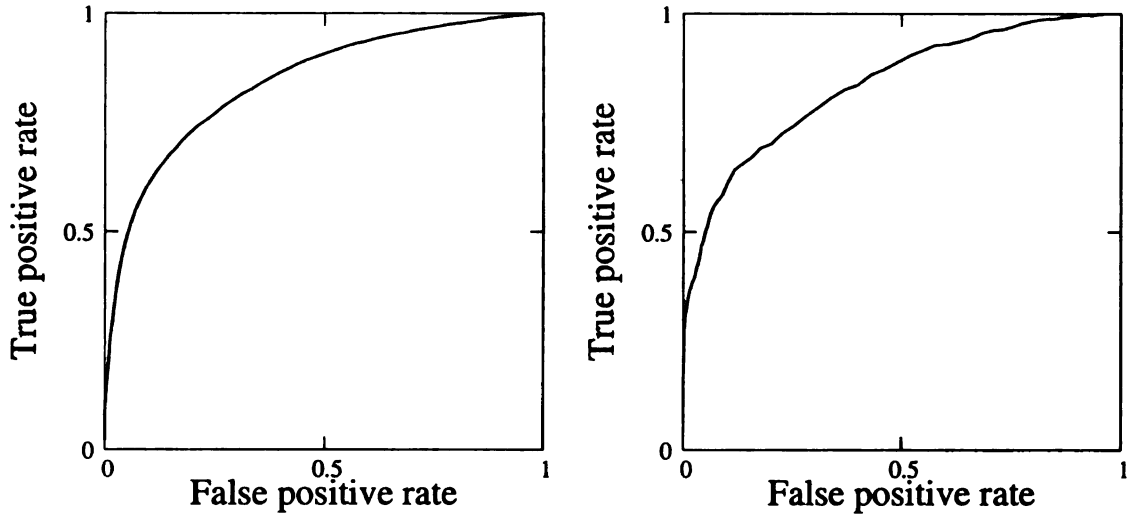
(b) PSD histogram without BCCH.

(c) WBNU PSD histogram.

(d) PSD histogram without WBNU.

Figure 7.13: Power spectral density (PSD) histograms for the black capped chickadee (BCCH) and the white breasted nuthatch (WBNU).

terns and ensembles respectively. Similarly for the white breasted nuthatch, the best true-positive rates attained with a precision of ≥ 0.9 are approximately 29% and 23%. Although reducing the frequency range used for detection smoothed the ROC curves, overall detection did not improve. However, a relatively high true-positive rate is attained with respect to the false-positive rate. Since the count of potential false-positives is much larger than that for true-positives, a relatively small percentage increase in false-positives represents a much greater deterioration of precision. For instance, in our experiments with white breasted nuthatch patterns, only approximately 1.9% of the test set represented the target species. Since the majority of acoustic events sensed in natural environments will not be voiced by the species of



(a) BCCH ROC curve using patterns.

(b) BCCH ROC curve using ensembles.

(c) WBNU ROC curve using patterns.

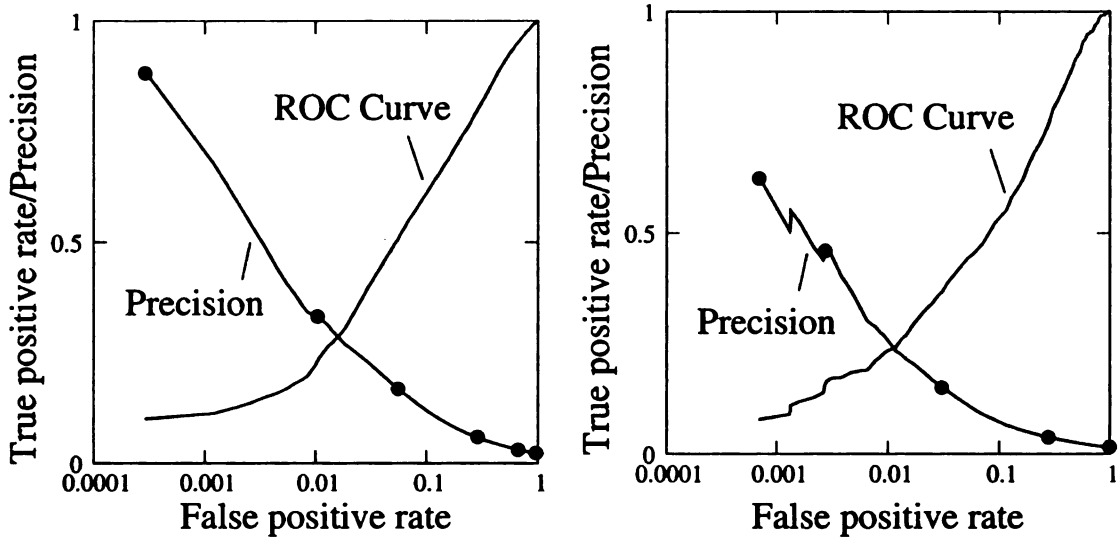
(d) WBNU ROC curve using ensembles.

Figure 7.14: ROC curves for detection of the black capped chickadee (BCCH) and the white breasted nuthatch (WBNU) using only the frequency range ≈[1.2kHz,6.0kHz] and ≈[1.2kHz,4.8kHz] respectively

interest, future research must strive for precise and accurate detection.

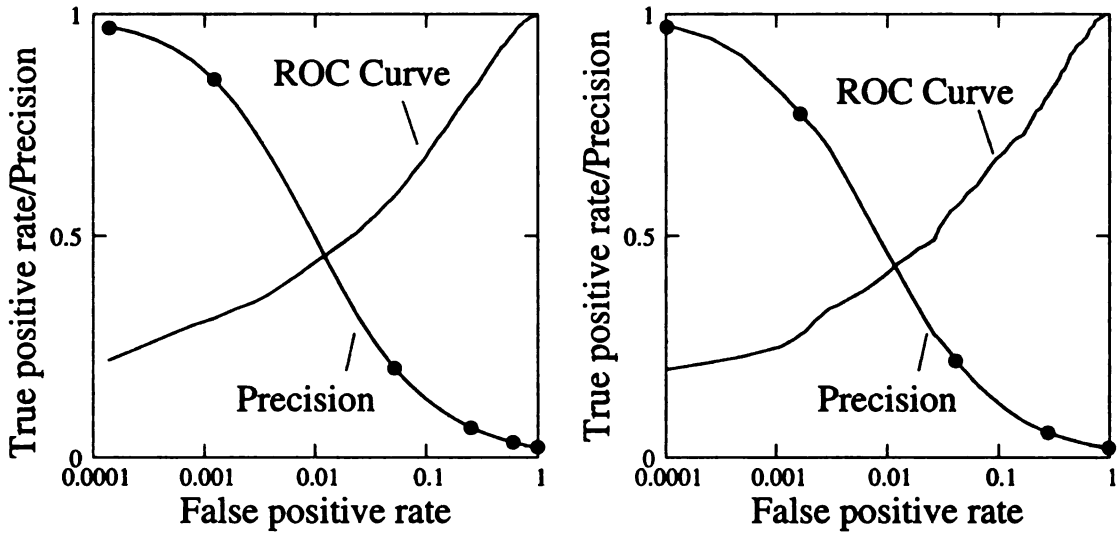
7.4 Related Work

Several research projects address selection of tuples from data streams [222–225]. Such works treat a data stream as a database and optimize query processing for better



(a) BCCH ROC curve using patterns (semi-log scale).

(b) BCCH ROC curve using ensembles.



(c) WBNU ROC curve using patterns (semi-log scale).

(d) WBNU ROC curve using ensembles.

Figure 7.15: Semi-log scale ROC curves and precision for detection of the black capped chickadee (BCCH) and the white breasted nuthatch (WBNU) using only the frequency range ≈[1.2kHz,6.0kHz] and ≈[1.2kHz,4.8kHz] respectively

efficiency. Other works address content-based routing [226], where tuple selection is used to route information based on data stream content. Our work with automated extraction of ensembles and annotation of data stream content may be beneficial to many of these approaches. For example, annotations can be treated as tuples that

describe the underlying data stream and can be used by selection schemes for routing data stream to address application specific requirements.

Recently, there has been increased interest on identifying motifs [212–214, 227] in time series. Motifs are defined as frequently occurring time series sequences. Identification of motifs requires analysis of a time series to determine which subsequences occur frequently. Motifs can be used for the construction of a model that represents the normal behavior of a time series. For instance, the motifs produced by an electrocardiogram may be used as a normal reference for a patient's heartbeat. Moreover, motifs can be clustered in support of time series data mining. On the other hand, a discord [216] is defined as the sequence that is least similar to others. Continuing with our electrocardiogram example, a discord may represent an aberration of a patient's heartbeat. Our work with ensembles complements work on motifs and discords in that ensembles can be considered as candidate motifs or discords. However, rather than focus on the most or least frequent time series patterns, ensembles are locally anomalous patterns that may recur only rarely. Each ensemble may be a motif, a discord or neither. Some approaches to motif and discord identification focus on subsequences of a specific length and require both scanning the time series and comparing subsequences to determine how often each occurs [213, 216]. Others consider variable length subsequences by iteratively increasing the subsequence length and rescanning the time series until a specified maximum length has been reached [215]. Our focus is on the timely, automated processing of continuous streams of sensor data that likely comprise variable length events. As such, processor and memory efficient techniques for extracting and processing ensembles are needed. Moreover, our approach to en-

semble extraction requires only a single scan of a time series and extracts variable length ensembles.

Methods that cluster data stream content to discover a meaningful structuring for the raw data [228,229] may also benefit from our approach for extraction of ensembles. As explained in [211], clustering data streams using sliding windows is ineffective in the general case. However, clustering motifs can be effective. Although ensembles are not necessarily frequently recurring, they are time series sequences that can be treated as candidate motifs. As we have shown, ensembles are useful for classification and detection applications using acoustic data streams.

Several projects have addressed detection and identification using time series data. For instance, MORPHEUS [215] addresses the need for unlabeled data sets that represent normal behavior for training anomaly detectors. MORPHEUS uses a motif oriented approach that extracts frequently occurring subsequences and treats them as normal pattern suitable for training a detector. Agile [230] uses a variable memory Markov model [231] (VMM) to detect transitions in an evolving sensor data stream produced by observing an underlying process. Agile uses a VMM to construct a reference model for a process and then reports a transition when process behavior no longer corresponds with the model. Partridge, et al. [232] demonstrated that traffic analysis of encrypted wireless network packet streams can be realized using signal processing techniques borrowed from acoustics. Using network timing information and coherence analysis, it was possible to determine how traffic was routed and identify client and server nodes. Our approach for ensemble extraction and processing may benefit from leveraging techniques described by these works, and in turn may

may provide an effective alternative for use by MORPHEUS or in place of Agile's use of VMM.

Recently, researchers have used computer vision techniques to help in the recognition of recorded music. For instance, Ke et al. [233] used such techniques to produce signatures from acoustic spectrograms for recorded songs. These signatures facilitated later online recognition of replayed music in the presence of noise or other sounds. Construction of signatures for classification or detection using acoustics complements our work by affording new techniques that may further improve species classification and detection in natural environments. However, recognition of a species of bird is likely more difficult than recognition of a specific recording, due to the natural variation found in bird vocalizations.

Other research groups have addressed classification of organisms based on their vocalizations. Mellinger and Clark [234] addressed classification of whale songs, with specific application to identification of bowhead song end notes, using spectrogram correlation. Fagerlund and Härmä [235] studied parameterization and classification of bird vocalizations, using 10 parameters that were used to describe the inharmonic syllables of 6 bird species. The 10 parameters were used to classify bird species using a k-nearest neighbor (kNN) approach using Euclidean and Mahalanobis distance. Classification accuracy was 49% using Euclidean distance and 71% using Mahalanobis distance. Another study [236] used bird song syllables and dynamic time warping [227] (DTW) to mitigate the impact of varying syllable lengths when computing distances. Syllables were clustered and then used for constructing histograms for each species. The histograms were compared, by computing their mutual corre-

lation, for recognition of 4 bird species. The highest classification accuracy attained was 80%, comparing favorably with our approach. However, we considered 10 species rather than 4 in our classification experiments.

Kogan and Margoliash [237] used DTW-based long continuous song recognition (LCSR) and hidden Markov models (HMM) in a comparative study for recognition of individual birds of a particular species. Specifically, experiments were conducted using the vocalizations of 4 zebra finches and 4 indigo buntings. LCSR requires careful selection of templates for matching vocalizations and other sounds (e.g., cage noises) in recordings. For HMM, a compound model was constructed by training separate HMMs on 3 sound categories: calls, syllables and cage noises. A proportion of each data set is used for HMM training (as much as 59%), while the entire data set is used for testing. Classification accuracy varied widely depending on the recognition method used whether recognition was based on syllables or songs, size of training and testing sets, template selection, and on which individual bird was to be recognized. For zebra finches, accuracy was in the range 4.4%-97.9% and 75.8%-98.6% for HMM and LCSR respectively; similarly for indigo buntings, 21.3%-99.4% and 62.0%-99.2%. Although very good recognition is attained in some cases, high classification accuracy depends on expert knowledge for selection of classifier parameters, and is somewhat optimistic since testing and training data overlap. Moreover, as noted [237], it has been shown that song birds learn their vocalizations with auditory feedback, which introduces variation between vocalizations of different individuals [203,204] facilitating recognition in a controlled environment. Due to environmental challenges, such as noise, it is expected that species detection in natural settings will be more difficult.

However, the LCSR and HMM approaches may benefit from automated ensemble extraction for selection of candidate sounds for training, testing and template construction. Moreover, use of LCSR and HMM techniques may complement our work with species detection and help improve detector precision.

Each of the above classification studies used different sized populations and different species, making direct comparison with our results difficult. However, in general, our method appears to compare well with other methods used for classification of birds. Moreover, none of the studies described above addressed the automated online extraction of acoustic events (ensembles) from streaming data for detection and classification of bird species in natural environments. Ensemble extraction helps reduce the processor and memory requirements for processing continuous data streams by focusing classification and detection tasks on ensemble data.

7.5 Discussion

We have presented a technique for extracting ensembles from acoustic data streams with the goals of classification and detection of bird species. Results of our classification and detection experiments show promise for automating species surveys using acoustics. Moreover, ensemble extraction and processing using distributed pipelines may enable timely annotation and clustering of sensor data streams. Annotation and clustering is a first step for transmuting raw data into usable information and its subsequent use for expanding our knowledge and understanding of our environment and other complex systems. Although our goals, as presented in this paper, are rela-

tively specific, the process for extracting ensembles is general and can be extended to other types of streaming sensor data and other applications. For instance, automated monitoring of a wireless communication channel may indicate that future packet loss rate is likely to increase. In turn, an autonomous decision maker may choose to insert new pipeline operators or modify parameters, enabling continued communication, by adding redundancy and correcting for lost packets. Moreover, automated detection of mechanical problems, such as bearing failure [218], enables early response and prevention of accidents.

Chapter 8

Forecasting Network Packet Loss

The main contribution of this chapter is to expand upon our work in Chapter 7, by investigating automated ensemble extraction for *forecasting* [238–240] network packet loss. Specifically, we extract ensembles from four network traces: one trace was collected as a user roamed about a wireless cell, and three traces were generated using packet loss models. Using these traces, we evaluate how accurately packet loss can be predicted as data is streamed to a receiver. The goal of forecasting is to predict near term transitions in system behavior [241]. We investigate automated forecasting of wireless network packet loss to enable early corrective response by autonomic decision makers. Specifically, we assume packet traces contain *events* comprising multiple sensor readings that collectively describe a period of similar packet loss behavior. For instance, events may capture the vocalization of a specific bird species or a period of high packet loss. Accurate forecasting enables decision makers to invoke preemptive adaptations to sensed conditions shortly after the onset of an event. For instance, a decision maker for a data streaming application can increase FEC redundancy soon

after the application encounters a period of increased packet loss. Forecasting differs from classification and prediction in that forecasting attempts to predict overall event characteristics based on early sensor readings, rather than processing an entire event at once.

The remainder of this chapter is organized as follows. Section 8.1 describes our approach for collecting roaming network traces, and Section 8.2 introduces the packet loss models used in this study. Next, in Section 8.3, a detailed description of our approach for extracting ensembles from network traces is presented. Section 8.4 describes our experimental method, and Section 8.5 presents the results of our experiments using ensemble extraction for forecasting network packet loss. Section 8.6 describes related work. Finally, in Section 8.7, we summarize and discuss the contribution of this chapter.

8.1 Trace Collection and Characterization

Our trace collection scenario is similar to that used in our Xnaut case study in that a stationary workstation transmits a data stream to a wireless access point that forwards the stream to a mobile receiver over an 11Mbps 802.11b wireless network. The transmitting station was a 1.5GHz AMD Athlon workstation running the Linux operating system, and the mobile receiver was a 1.83 GHz Intel Core Duo MacBook Pro running Mac OS X. The workstation transmitted packets comprising 200 bytes of data 40 times per second using the UDP/IP protocol. This produced a data stream with a rate of 8,000 bytes per second, sufficient to support 8 bit vocal communication.

Trace data comprises the sequence number for each packet received and the delay, in microseconds, between the arrival of each packet.

A user roamed about a wireless cell in an outdoor environment, and 5 hours of trace data was collected in half hour increments. The workstation and the access point were both located inside a wood and brick building and transmitted to a roaming receiver outside the building. Receiver motion, physical distance, reflection and occlusions caused by physical objects, such as trees or the body of the user carrying the receiver, all affected packet loss rate. Notably, wireless computer networks typically operate at 2.4GHz that transmits poorly through obstacles containing water, such as dense vegetation. The user performed several different activities while carrying the receiver including: standing in one place, pacing back-and-forth over short distances, and walking over longer distances. Typically, the user stood at one location or paced for approximately 15 to 20 seconds before walking to a new location. Distance from the access point ranged from only a few feet to approximately 90 feet.

8.1.1 Trace Scoring and Sampling

One problem when making decisions using trace data is that the sequence number and inter-packet delay are only updated when a new packet is received. During periods of high loss, the time between packet receptions can become protracted, inhibiting timely response by a decision maker. For this reason, in software, we implemented a synthetic sensor to compute a *trace score* that can be sampled at regular intervals, rather than being read only when a new packet arrives. Our trace score is defined as:

$$score_{trace}(\Delta s, \Delta r) \equiv \begin{cases} 0, & \text{iff } \Delta s = 0 \\ \\ \frac{1+\Delta r}{\Delta s}, & otherwise \end{cases},$$

where Δs is the difference between the current sequence number and the one when the sensor was last read, and Δr is the change in the number of packets received. When there is no change in the sequence number, this function returns 0. When no packets are lost, the sequence number and the number of packets received both increase by one, and the function returns 2. When packets are lost, the sensor returns a value between 0 and 2. In our approach, the sensor executes in a thread separate from that of the receiver. Thus, it is possible for the sensor to read both the sequence number and packet counter before an update of both is complete. A mutex can be used to prevent the sensor from reading during an update. However, we prefer to sample at a specific regular interval, and the use of a mutex may incur additional delay during trace score computation. Our trace score is designed to differentiate between when no new packets were received, and Δs does not increase, and when all packets appear to be lost, and only Δs increases.

In our approach, the sensor sampling frequency must be computed. The Shannon-Nyquist sampling theorem [242–245] holds that a continuous-time signal can only be reconstructed from its samples if the sampling frequency is more than twice the frequency of the signal. We observe that the rate at which the sequence number and inter-packet delay are updated is at most 40 times per second. Thus, the sensor should be sampled at a frequency greater than 80 times per second to avoid aliasing,

where different signals appear to be identical. We sampled the trace score 120 times per second, which produces a signal that repeats the sequence 0,0,2 during periods without packet loss.

8.1.2 Trace Characterization

Figure 8.1 depicts two methods for visualizing network traces using trace scores. The top graph shows a scatter plot of the trace score for a 30 minute roaming trace. The solid lines at the top and bottom comprise the most frequently occurring trace score values, 2 and 0. Points plotted between 2 and 0 indicate that packets have been lost. The bottom graph shows the same trace plotted as a spectrogram. Using a sampling approach for observing packet loss enables the construction of a spectrogram. Construction of a spectrogram from trace score data, is similar to the method for constructing a spectrogram from acoustic data, discussed in Chapter 7.1.1. First, the trace score data is divided into equal sized segments and then it is passed through the pipeline depicted in Figure 7.1.1 to construct a frequency domain representation. Spectrograms are useful for visualizing the rate at which the trace score changes. Comparing the scatter plot and the spectrogram reveals that, during periods of packet loss, trace score values exhibit rapid, broad-spectrum, change. Notably, when roaming, packet loss behavior is significantly affected by the motion of the person carrying the receiver. When the receiver is moving, packet loss is typically more pronounced. As such, the periods of packet loss shown in Figure 8.1 often correspond to periods when the user is pacing or walking from one place to another.

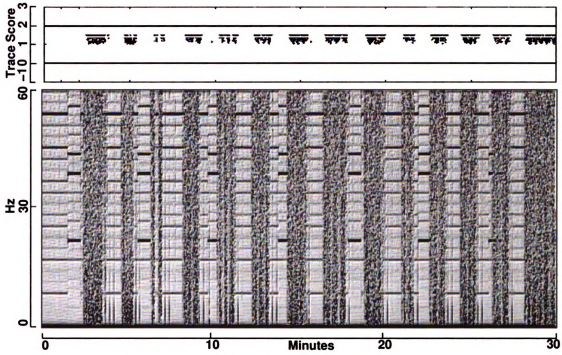


Figure 8.1: Top, a scatter plot of the trace scores for a 30 minute roaming trace. Bottom, a spectrogram of the same roaming trace.

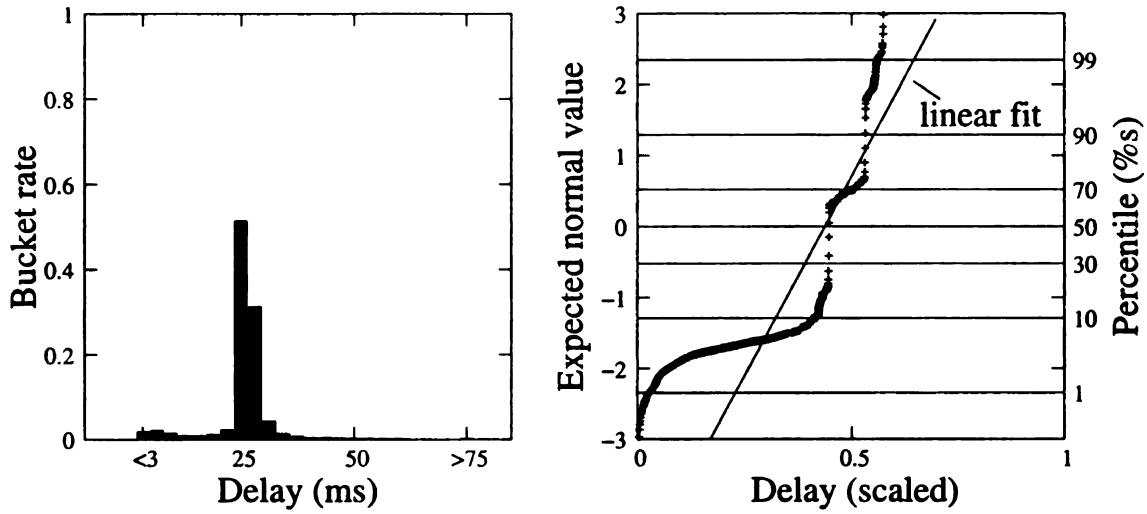
We consider a *burst* to be a period when no packets are lost. Conversely, a *gap* is a period where all packets are lost, terminated by the reception of the first packet of a new burst. To characterize burst and gap periods we plot histograms and probability plots for inter-packet delay, run length and loss rate. Probability plots are produced by plotting the frequency at which data points occur in comparison with a selected distribution. If the plotted points are drawn from this distribution, they appear to have a linear relationship. Histograms and probability plots are computed using the 5 hours of roaming trace data.

Shown in Figure 8.2 are histograms and normal probability plots for burst and gap inter-packet delay on a per packet basis. First, let us consider the distribution of burst inter-packet delays. The histogram, depicted in Figure 8.2(a), shows that the most

common value is approximately 25 milliseconds, as expected for a packet transfer rate of 40 packets per second. Moreover, the computed mean and standard deviation are 24.6±5.0, indicating a similar central value. Although the histogram distribution appears to be somewhat bell shaped, Figure 8.2(b) reveals that the distribution of burst delays does not appear to be Gaussian. Now, let us consider the distribution of gap inter-packet delays. Figure 8.2(c) plots the histogram of gap inter-packet delays. The most common value is approximately 35 milliseconds, and similarly the computed mean and standard deviation are 34.5±5.9. Gap delays tend to be significantly larger than those for bursts. Again, the histogram appears to be bell shaped, and from Figure 8.2(d) the distribution appears to be approximately Gaussian.

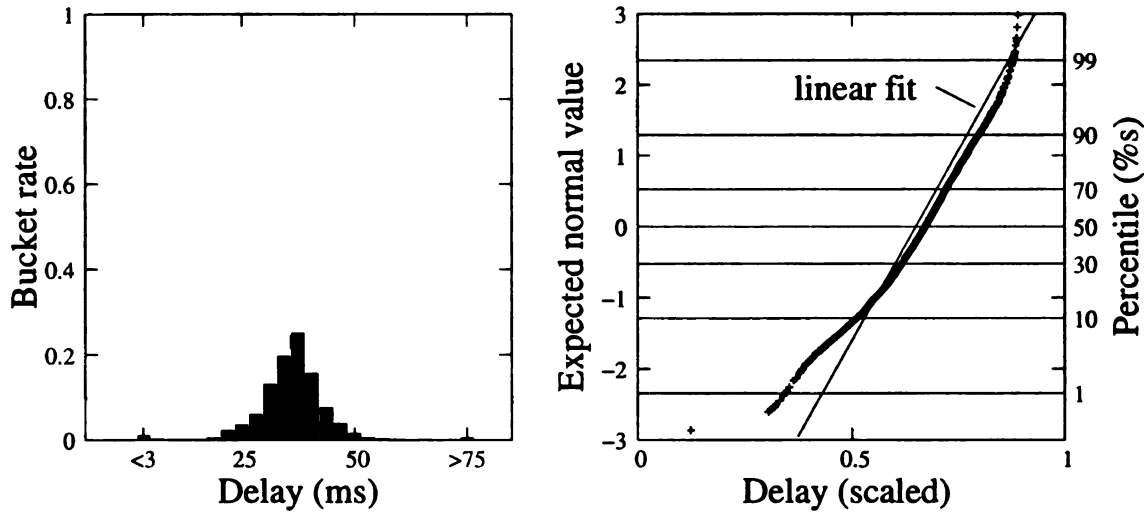
Next, let us consider the distribution of burst and gap run-lengths in terms of packet count. The histogram, shown in Figure 8.3(a), shows that most bursts comprise only a few packets. Moreover, the plot in Figure 8.3(b) shows that burst run-lengths appear to be geometrically distributed except for very long bursts. Similar to bursts, most gaps comprise only a few packets, as shown in Figure 8.3(c). However, as shown in Figure 8.3(d), the distribution of gap run-lengths is not clearly geometric. Intuitively, if the burst run-lengths are geometrically distributed, a geometric distribution of gap run-lengths is expected. However, if individual packet losses are not independent Bernoulli trials, then a geometric distribution may not exist. Notably, packet losses often correlate with conditions that impact packet delivery, such as signal occlusion, distance or receiver motion.

Finally, let us consider the distribution of loss rates for 1 and 5 second segments. We compute these loss rates by dividing the 5 hours of roaming trace data into 1-



(a) Burst histogram.

(b) Burst normal probability.

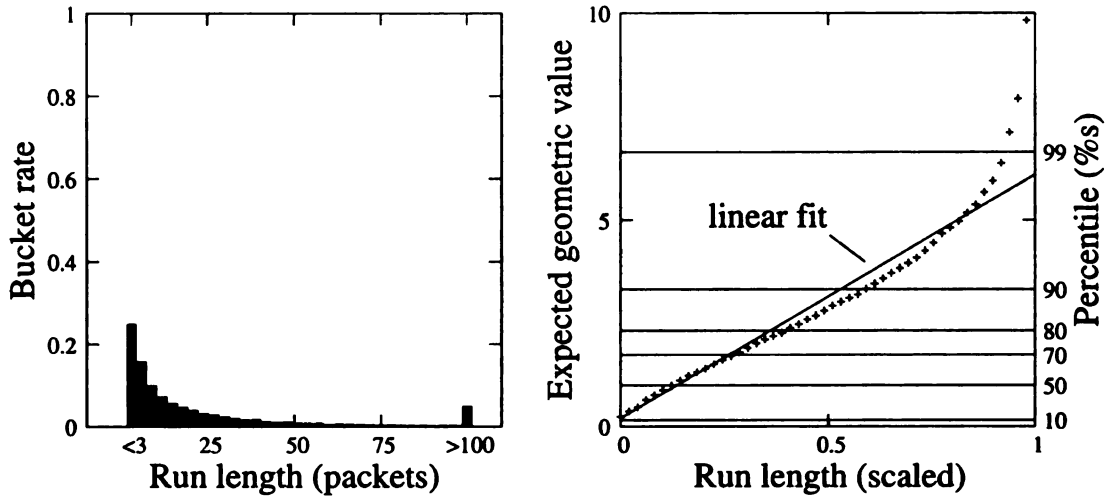


(c) Gap histogram.

(d) Gap normal probability.

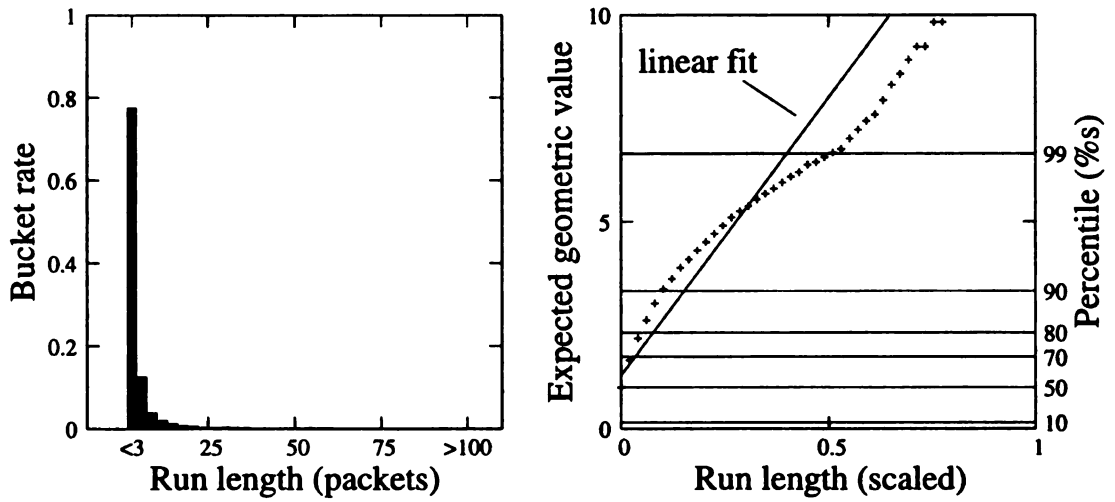
Figure 8.2: Burst and gap delay histograms and normal-probability plots for a roaming receiver. Normal-probability plots represent the delay range [3ms-50ms] scaled by subtracting 3ms and dividing by 50ms.

and 5-second, non-overlapping segments, and compute the loss rate for each. We plot histograms and exponential probability plots in Figure 8.4. The sets of 1 and 5 second loss rates both comprise many segments of little or no loss that are distributed approximately exponentially. In fact, for most segments, no packets were lost, and more than 90% of segments had a loss rate $\leq$1%. As such, a loss rate increase likely correlates with an increase in the number of low loss segments. This suggests that



(a) Burst histogram.

(b) Burst geometric probability.



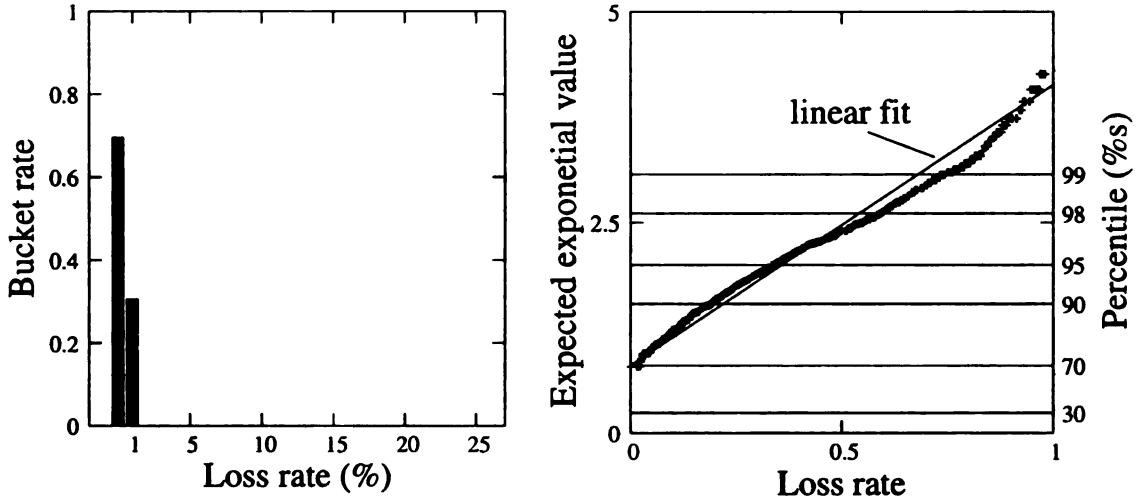
(c) Gap histogram.

(d) Gap geometric probability.

Figure 8.3: Burst and gap run-length histograms for a roaming receiver. Geometric-probability plots are scaled by dividing by the longest run-length.

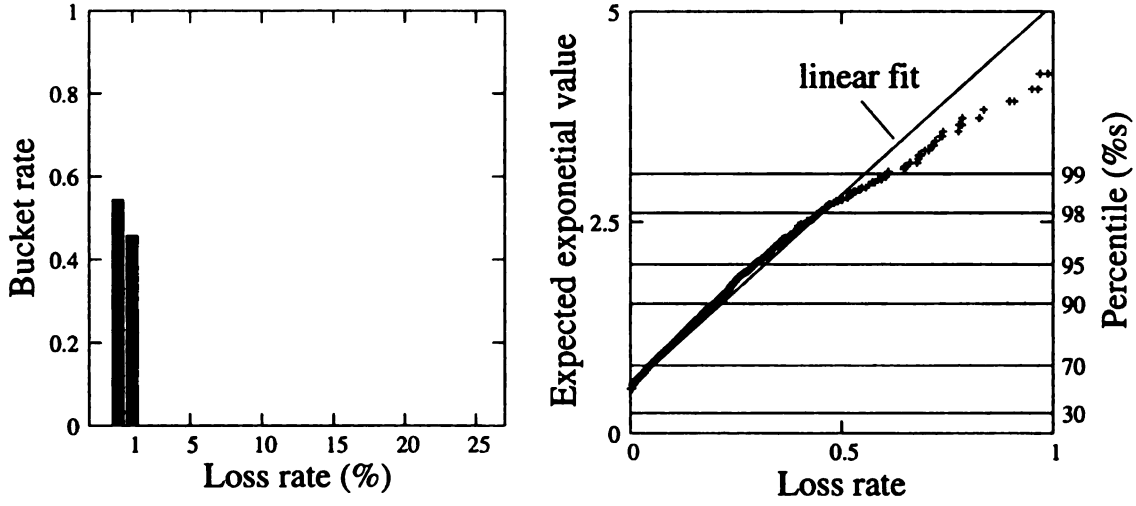
packet loss can be described as comprising distributions of loss rate segments.

In this section, we described a sampling approach for collecting and characterizing computer network packet loss. Our analysis of traces collected while roaming about a wireless cell reveals that lossy periods are characterized by rapid, broad-spectrum change. Moreover, an increase in loss rate likely correlates with a corresponding increase of short, low-loss segments, a result similar to that observed in other works [7,



(a) 1-second loss histogram.

(b) 1-second exponential probability.



(c) 5-second histogram.

(d) 5-second exponential probability.

Figure 8.4: 1-second and 5-second Loss rate histograms and exponential probability plots. Exponential probability plots are scaled by dividing by the largest loss rate.

246–248]. Moreover, the distribution of burst and gap inter-packet delay and run-length may differ significantly.

8.2 Packet Loss Models

In our forecasting experiments, described in Section 8.4, we compare and contrast forecasting packet loss using a real, 5-hour, roaming trace with 2 simulation models

and a stepwise model for generating artificial packet losses. In autonomic software, models can be used to train a decision maker and construct statistical representations of environmental or software behavior offline. Moreover, collecting traces in real environments that provide good coverage of overall behavior can be time consuming and difficult. As such, simulation of environmental conditions and software operation can ease the implementation of autonomic decision makers.

First, we describe the Gilbert-Elliot model [249,250] that is well known and widely used for modeling network packet loss. For our forecasting experiments, we use the simplified Gilbert-Elliot model described by Tang et al. [7]. As depicted in Figure 8.5, a two-state Markov model is used to transition between a burst and gap state in a probabilistic fashion. As shown, the probabilities of remaining in the burst and gap states are α and β, respectively. Moreover, the burst-to-gap transition probability is given by $(1 - \alpha)$ while the gap-to-burst probability is given by $(1 - \beta)$.

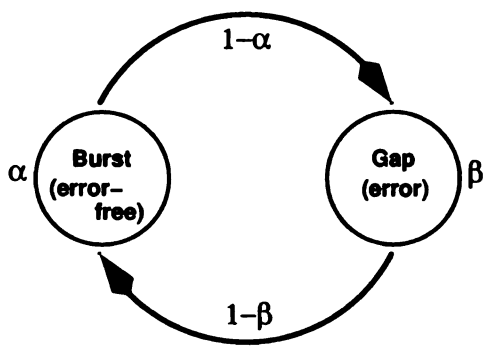


Figure 8.5: Simplified Gilbert-Eliot Model (adapted from [7]).

Since, packet losses are independent Bernoulli trials in this model, both burst and gap run-lengths are geometrically distributed. As such, the burst and gap run-length can be computed when generating a trace by [7]:

$$length_{run}(u,p) = \frac{\log(u)}{\log(1-p)},$$

where u is a uniformly distributed random variable drawn from the interval [0,1], and p is the probability for transitioning out of the burst or gap state, depending on whether a burst or gap run-length is being computed. As described in [247], we compute α and β directly using the run-lengths extracted from the 5 hours of roaming trace data, and set the burst-to-gap and gap-to-burst transition probabilities to 0.0184534 and 0.299966, respectively.

The second model we consider is a trace-based approach for modeling network channel behavior introduced by Nguyen et al. [246]. This approach also uses a two-state model, but differs from the simplified Gilbert-Elliot model by using a composite function comprising the piecewise assembly of pareto and exponential distributions. Using a large number of packet traces collected in a 2Mbps Lucent WaveLan network, linear regression was used to compute distribution parameters that best matched the packet losses exhibited by the traces. The authors showed that this trace-based model was more accurate than a simple Gilbert-Elliot model in terms of accurately representing the distribution of run-lengths computed from the traces. Finally, the third model that we use is the simple probabilistic loss model we used in our case study for adaptive error control, described in Chapter 6. Artificial packet losses are generated by varying the loss rate from 0.0 to 0.8 in steps of size 0.025.

Inter-packet burst and gap delays were added to each of these three models based on histograms computed using the 5-hour roaming trace data. For each generated

packet, a burst or gap delay was randomly selected from a histogram according to the frequency with which each delay period appears. The inclusion of inter-packet delay enabled these generated traces to be sampled in the same way as the real roaming trace. We generated ten hours of trace data using each of these three models for use in our experiments.

8.3 Ensemble Extraction and Processing

Figure 8.6 depicts our approach to automated network trace analysis using a Dynamic River pipeline that targets forecasting of packet loss. First, trace scores are sampled by the `readout` operator that encapsulates trace scores as Dynamic River records. The remaining operators comprise the process for extracting ensembles and processing for forecasting packet loss using MESO.

The pipeline segment, ⇒`[saxanomaly|steptrigger|stepcutter]`, transforms records comprising trace scores into ensembles. The moving average of the SAX anomaly score, as described in Section 7.1.3, is output by `saxanomaly` in addition to the original trace data. This moving average is used as a window of anomalous behavior by the `stepcutter` operator. In our experiments with forecasting packet loss, we set both the moving average window and the SAX anomaly window to 120 and the SAX alphabet size to 8. Figure 8.7 plots the anomaly score computed by the `saxanomaly` pipeline operator for the signal depicted in Figure 8.1.

Figure 8.8 depicts the trigger signal output by the `steptrigger` operator (top) and the corresponding 5-second loss rates (bottom). The `steptrigger` operator trans-

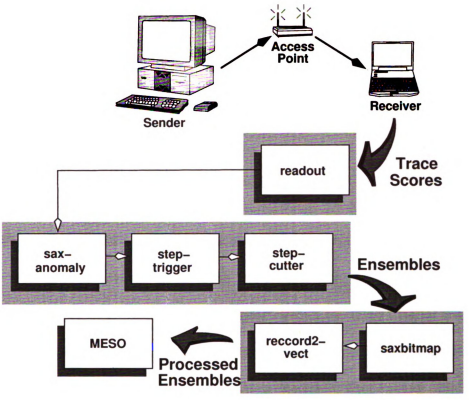


Figure 8.6: Block diagram of pipeline operators for converting network traces into ensembles for forecasting packet loss.

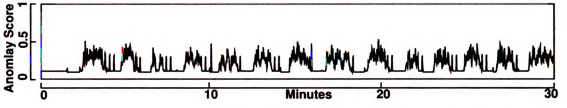


Figure 8.7: Anomaly score generated for the signal shown in Figure 8.1.

forms the anomaly score output by `saxanomaly` into a trigger signal that has discrete, integer values selected based on a threshold setting. If the anomaly score increases or decreases by more than the specified threshold, the trigger score emitted is correspondingly adjusted by dividing the anomaly score by the threshold and rounding to nearest whole number. In our experiments, we set the `steptrigger` threshold to 0.05.

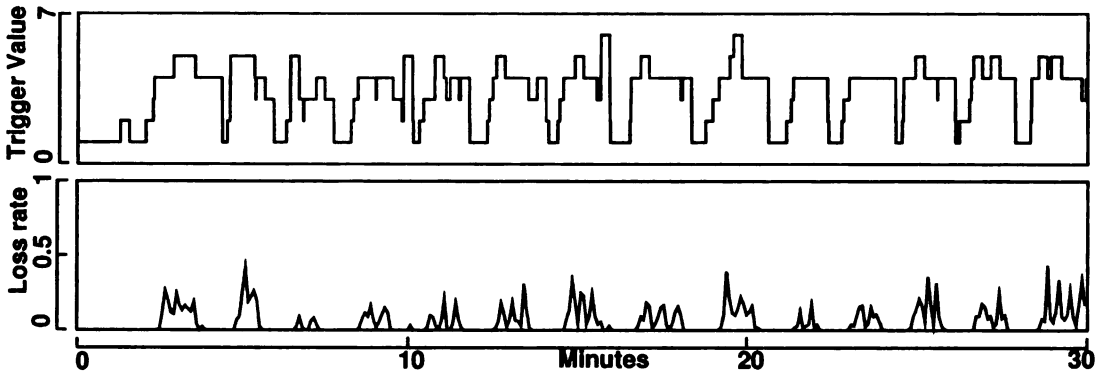


Figure 8.8: Step trigger signal and 5 second loss rates for the trace score shown in Figure 8.1.

The **stepcutter** operator reads both the records containing the original trace scores and the records emitted by **steptrigger**. When the trigger signal transitions to a different value, **stepcutter** first closes the current scope and then emits an **OpenScope** record, designating the start of a new ensemble. **Stepcutter** then begins composing a new ensemble. Each ensemble comprises values of the original trace score data that correspond to periods when the trigger signal had a specific value. The record stream, as emitted from **stepcutter**, comprises ensembles that represent periods of similar packet loss behavior as detected by **saxanomaly**. Each ensemble is then written to **saxbitmap** for conversion to SAX bitmap representation. Finally, this bitmap is passed to **record2vect** for conversion to a vector representation that can be used as a training or testing pattern by MESO. A sequence length of 2 was used for constructing SAX bitmaps for our forecasting experiments.

8.4 Data Sets and Methodology

We divide all the ensembles extracted from a trace into a training and testing set using the mean 5-second loss rate computed for each ensemble. Computed loss rates have a resolution of 0.01. Ensembles are grouped by loss rate and each group is divided evenly between the training and testing sets with extra ensembles, from odd numbered groups, assigned for training. Moreover, we retain only ensembles that are at least 10 seconds in length for our forecasting experiments. The data sets used in our forecasting experiments differ from those used for our classification and detection experiments, described in Chapter 7, in that ensembles are not comprised of patterns. Instead, patterns are constructed by computing a SAX bitmap for one or more seconds of ensemble data. As shown in Table 8.1, each ensemble is used to produce one or more patterns for processing by MESO. Training patterns are constructed by computing a SAX bitmap on a second-by-second basis for each ensemble. That is, a SAX bitmap is computed using the first second of ensemble data, then the first two seconds of ensemble data and so forth. For training, all second-by-second bitmaps are computed for each training ensemble. Testing patterns are computed for each testing ensemble, on a second-by-second basis, for the first 1 through 10 seconds of ensemble data. Table 8.1 gives the name of each data set, a brief description, the number of training and testing ensembles used to produced MESO patterns, and the number of training and testing patterns produced.

For our experiments, we labeled the training and testing patterns in two ways. First, we assigned each pattern a metadata label comprising the minimum, maximum

Table 8.1: Total training and testing ensemble counts.

Data set	Description	Ensemble count		Pattern count	
		Training	Testing	Training	Testing
Roam	5-hour roaming trace	351	328	8238	328
Gsim	Gilbert-Elliot model	802	797	15111	797
Wlsim	Trace-based wireless model	735	712	16529	712
Ploss	Step-wise probability model	544	498	17854	498

and mean 5-second loss rate computed using the trace data associated with each ensemble. Loss rates are computed by dividing trace data into discrete 5-second periods and computing the loss rate for each. Table 8.2 shows the number of training and testing patterns for several ranges of mean loss rates. For the first three data sets, most of patterns have relatively low loss rates. For the ploss data set, packet losses are generated systematically. As such, the ploss data set has a more balanced distribution of patterns across loss rate categories.

Second, for each pattern we selected the FEC (n, k) combination that provides the least redundancy that is greater than or equal to the mean ensemble loss rate. Decision makers for autonomic software, such as the Xnaut, can use these FEC labels to invoke corresponding adaptive actions. We selected each FEC code from the following set of 13 (n, k) combinations:

$$\begin{array}{lllll} (1,1) & (10,9) & (5,4) & (4,3) & (3,2) \\ (4,2) & (6,2) & (8,2) & (10,2) & (12,2) \\ (14,2) & (16,2) & (18,2). & & \end{array}$$

The number of training and testing patterns assigned a particular FEC code is shown in Table 8.3 (continued on the next page). For the first three data sets, most patterns are assigned an FEC code that provides relatively little redundancy, while the ploss data set has a more evenly balanced assignment.

Table 8.2: Training and testing data set characterization.

Data set	Loss% (r)	Pattern count	
		Training	Testing
Roam	$r = 0$	3599 (43.7%)	161 (49.1%)
	$0 < r \leq 20$	4002 (48.6%)	149 (45.4%)
	$20 < r \leq 40$	508 (6.2%)	16 (4.9%)
	$40 < r \leq 60$	78 (0.9%)	1 (0.3%)
	$60 < r \leq 80$	30 (0.4%	0 (0.0%)
	$80 < r \leq 100$	21 (0.3%)	1 (0.3%)
Gsim	$r = 0$	161 (1.1%)	12 (1.5%)
	$0 < r \leq 20$	14950 (98.9%)	785 (98.5%)
	$20 < r \leq 40$	0 (0.0%)	0 (0.0%)
	$40 < r \leq 60$	0 (0.0%)	0 (0.0%)
	$60 < r \leq 80$	0 (0.0%)	0 (0.0%)
	$80 < r \leq 100$	0 (0.0%)	0 (0.0%)
Wlsim	$r = 0$	10656 (64.5%)	497 (69.8%)
	$0 < r \leq 20$	2713 (16.4%)	112 (15.7%)
	$20 < r \leq 40$	3149 (19.1%)	103 (14.5%)
	$40 < r \leq 60$	11 (0.1%	0 (0.0%)
	$60 < r \leq 80$	0 (0.0%)	0 (0.0%)
	$80 < r \leq 100$	0 (0.0%)	0 (0.0%)
Ploss	$r = 0$	319 (1.8%	16 (3.2%)
	$0 < r \leq 20$	3199 (17.9%)	105 (21.1%)
	$20 < r \leq 40$	4212 (23.6%)	120 (24.1%)
	$40 < r \leq 60$	4881 (27.3%)	157 (31.5%)
	$60 < r \leq 80$	5121 (28.7%)	96 (19.3%)
	$80 < r \leq 100$	122 (0.7%)	4 (0.8%)

Experimental method. For our forecasting experiments we select one data set for training MESO and select either the same data set or a different one for testing. Training uses the entire set of training data while testing uses SAX bitmap patterns constructed using only the first 1 to 10 seconds from each testing set ensemble. Note, when the same data set is used for both training and testing, training and testing data do not overlap. Each experiment is conducted as follows:

1. Randomize the patterns in the training set.
2. Train MESO using all the data in the randomized training set.
3. Using the ensembles in the test set, construct a SAX bitmap test pattern for each ensemble using only the first t seconds of ensemble data.

Table 8.3: Training and testing data set characterization labeled with FEC codes.

Data set	FEC(n,k)	Pattern count	
		Training	Testing
Roam	(1,1)	3620 (43.9%)	162 (49.4%)
	(10,9)	2955 (35.9%)	111 (33.8%)
	(5,4)	1047 (12.7%)	38 (11.6%)
	(4,3)	300 (3.6%)	9 (2.7%)
	(3,2)	116 (1.4%)	7 (2.1%)
	(4,2)	155 (1.9%)	1 (0.3%)
	(6,2)	30 (0.4%)	0 (0.0%)
	(8,2)	15 (0.2%)	0 (0.0%)
	(10,2)	0 (0.0%)	0 (0.0%)
	(12,2)	0 (0.0%)	0 (0.0%)
	(14,2)	0 (0.0%)	0 (0.0%)
Gsim	(1,1)	161 (1.1%)	12 (1.5%)
	(10,9)	14792 (97.9%)	778 (97.6%)
	(5,4)	158 (1.1%)	7 (0.9%)
	(4,3)	0 (0.0%)	0 (0.0%)
	(3,2)	0 (0.0%)	0 (0.0%)
	(4,2)	0 (0.0%)	0 (0.0%)
	(6,2)	0 (0.0%)	0 (0.0%)
	(8,2)	0 (0.0%)	0 (0.0%)
	(10,2)	0 (0.0%)	0 (0.0%)
	(12,2)	0 (0.0%)	0 (0.0%)
	(14,2)	0 (0.0%)	0 (0.0%)

4. Test MESO using all the SAX bitmap test patterns, reporting both the predicted and actual minimum, maximum and mean ensemble loss rates and FEC code metadata.
5. Repeat the preceding steps n times, and evaluate using the evaluation metrics discussed in Section 8.4.1 over all n iterations.

In our tests, we set n equal to 100 and t ranges from 1 to 10 seconds. Thus, MESO is trained and tested 100 times for each experiment and 10 experiments, one for each time period, are completed for each training/testing data set combination.

Table 8.3: (cont'd)

Data set	FEC(n,k)	Pattern count	
		Training	Testing
Wlsim	(1,1)	10656 (64.5%)	497 (69.8%)
	(10,9)	1937 (11.7%)	89 (12.5%)
	(5,4)	776 (4.7%)	23 (3.2%)
	(4,3)	1037 (6.3%)	28 (3.9%)
	(3,2)	1891 (11.4%)	65 (9.1%)
	(4,2)	232 (1.4%)	10 (1.4%)
	(6,2)	0 (0.0%)	0 (0.0%)
	(8,2)	0 (0.0%)	0 (0.0%)
	(10,2)	0 (0.0%)	0 (0.0%)
	(12,2)	0 (0.0%)	0 (0.0%)
	(14,2)	0 (0.0%)	0 (0.0%)
Ploss	(1,1)	319 (1.8%)	16 (3.2%)
	(10,9)	1722 (9.6%)	56 (11.2%)
	(5,4)	1477 (8.3%)	49 (9.8%)
	(4,3)	1073 (6.0%)	27 (5.4%)
	(3,2)	1564 (8.8%)	44 (8.8%)
	(4,2)	3998 (22.4%)	139 (27.9%)
	(6,2)	3800 (21.3%)	88 (17.7%)
	(8,2)	2643 (14.8%)	44 (8.8%)
	(10,2)	1136 (6.4%)	31 (6.2%)
	(12,2)	122 (0.7%)	4 (0.8%)
	(14,2)	0 (0.0%)	0 (0.0%)

8.4.1 Evaluation Metrics

In these experiments, the goal is to accurately predict overall ensemble packet loss behavior using only the first few seconds of ensemble data, potentially enabling timely response by an autonomic decision maker. We evaluate forecast accuracy in three ways. First, using the mean ensemble loss rate, we evaluate accuracy using loss rate *margins*. A margin specifies how much the predicted loss rate can differ from the actual mean and still be counted as an accurate forecast. For instance, a prediction will be considered accurate, using a 2% margin with an actual mean loss rate of 20%, if the predicted rate is in the range [18%-22%]. Second, using the FEC code metadata

labels, we evaluate how accurately MESO predicts the optimal code assigned to each test pattern. Finally, we evaluate using the minimum and maximum loss rates using *coverage* and *precision*, discussed next.

Coverage. Packet loss behavior, as captured using ensembles, can be described as a period when the loss rate varies only over a limited range. In our experiments, we use ranges with an upper and lower bound set to the maximum and minimum 5-second loss rate. Changing FEC codes to address changes in the mean loss rate may provide insufficient redundancy for a wide range of loss rates. That is, many packets may still be lost when the loss rate increases above the mean. A decision maker that can accurately forecast packet loss ranges may choose to add greater redundancy for a period when loss rate varies widely while conserving bandwidth during a period with a narrow range. We evaluate forecasting accuracy, using packet loss ranges, with *coverage* and *precision* metrics. Coverage is the proportion of the actual range that is covered by the predicted range. A value of 1 indicates complete coverage, while 0 indicates no coverage of the actual range. Formally, we define coverage as:

$$coverage(C_L, C_U, T_L, T_U) \equiv \begin{cases} 0, & \text{if } (C_U - C_L) < 0 \\ \dfrac{C_U - C_L + 1}{T_U - T_L + 1}, & otherwise \end{cases},$$

where T_U and T_L are the upper and lower loss percentages for the actual (true) ensemble range, and C_U and C_L are the upper and lower values of the coverage range. We define C_U and C_L as:

$$C_U(T_U, P_U) \equiv \begin{cases} P_U, & \text{iff } P_U \leq T_U \\ \\ T_U, & otherwise \end{cases} \qquad C_L(T_L, P_L) \equiv \begin{cases} P_L, & \text{iff } P_L \geq T_L \\ \\ T_L, & otherwise \end{cases},$$

where P_U and P_L are the upper and lower loss percentage for the predicted range.

Precision. Notably, a very wide predicted range can completely cover the actual range, but be imprecise. That is, the range predicted can be much wider than the actual range, and may cause a decision maker to add too much redundancy and waste bandwidth. We further evaluate forecasting loss ranges using a measure of precision defined as:

$$precision(T_L, T_U, P_L, P_U) \equiv 1 - \frac{\mid T_U - P_U \mid + \mid T_L - P_L \mid}{200},$$

that computes how closely the predicated range matches the actual range. A precision of 1 indicates a perfectly precise forecast.

8.5 Assessment

Figure 8.9 shows the forecasting accuracies when using mean ensemble loss rates with margins. Tabulated data, used for producing the plots included in this section, can be found in Appendix C. In these experiments, training and testing patterns came from the same data set. As shown, MESO accuracy increases with an increase in margin size and as more ensemble data is used for creating a test pattern. For a 0% margin, accuracy is typically low for all data sets. With a 5% margin, accuracy is

minimally 44.1% for the ploss data set when test patterns are constructed using only 1 second of data. However, when test patterns are constructed using 10 seconds of data, accuracy is more than 80.5% for all data sets. Since packet loss behavior is very dynamic while roaming, we consider these results promising.

Figure 8.10 shows the forecasting accuracies when MESO was trained with patterns produced using generated data and tested using real data from the roam data set. Again, mean ensemble loss rates and margins are used for evaluation. As shown, accuracy typically increases with margin size. However, as longer periods are used for constructing test patterns, accuracy does not show clear improvement when training with the gsim or wlsim data sets. Moreover, variance is significant for all data sets and standard deviations are often >5%. However, when training with the ploss data set, accuracy increases with an increase in margin size and the period used for constructing test patterns. We attribute the utility of training with ploss data to the systematic approach used for generating packet losses. This systematic approach provides a better representation of the wide range of packet loss rates found in a real roaming trace.

Figure 8.11(a) shows forecasting accuracy when patterns are labeled with the 13 FEC codes introduced in Section 8.4. In all cases except gsim, accuracy increases as longer periods are used for constructing testing patterns. Using discrete FEC codes, the highest accuracy attained was 96.5% for the gsim data set, while for the roam data set, an accuracy of 71.8% was attained when test patterns were constructed using 10 seconds of ensemble data.

Results when training using generated data and testing on data collected while

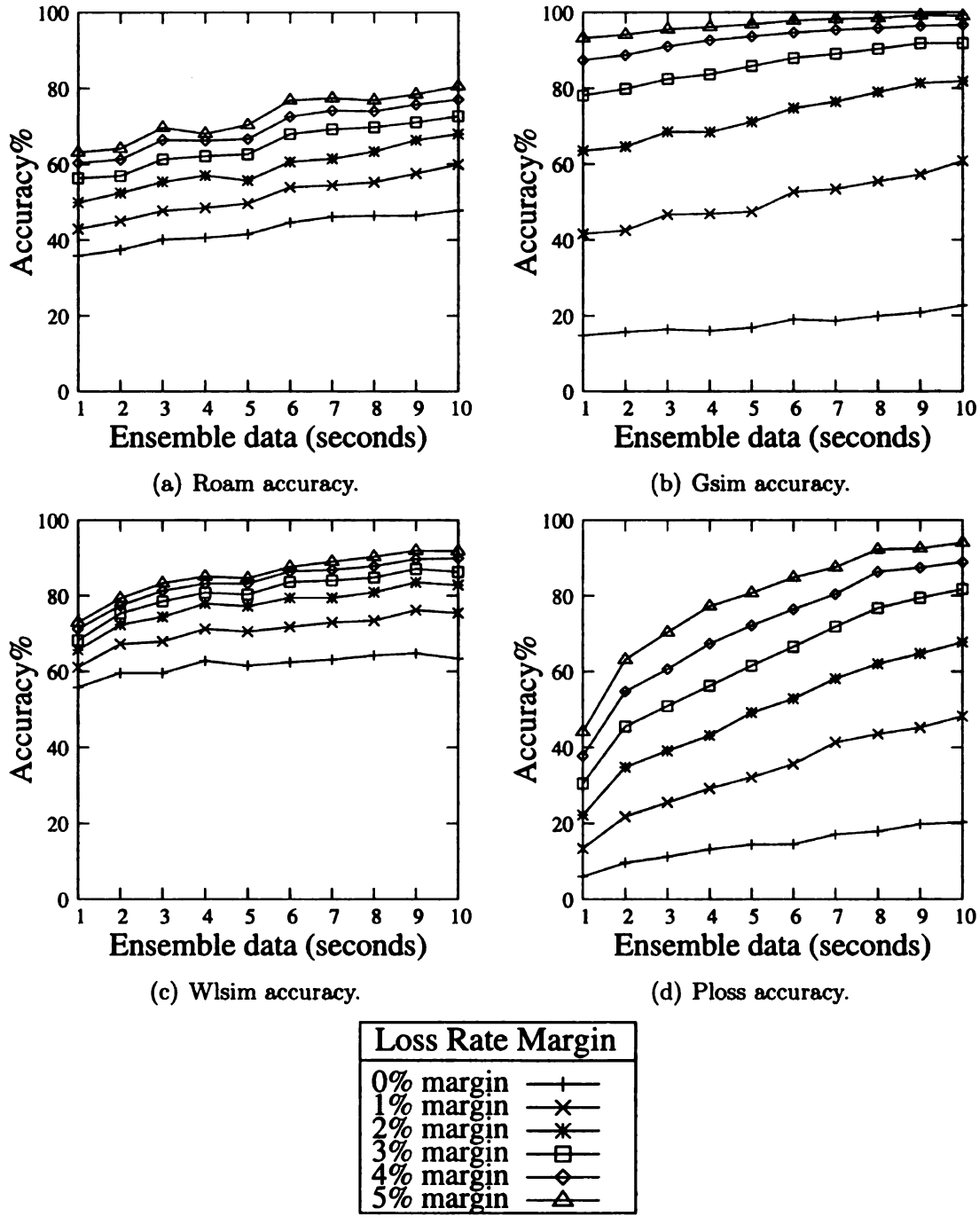
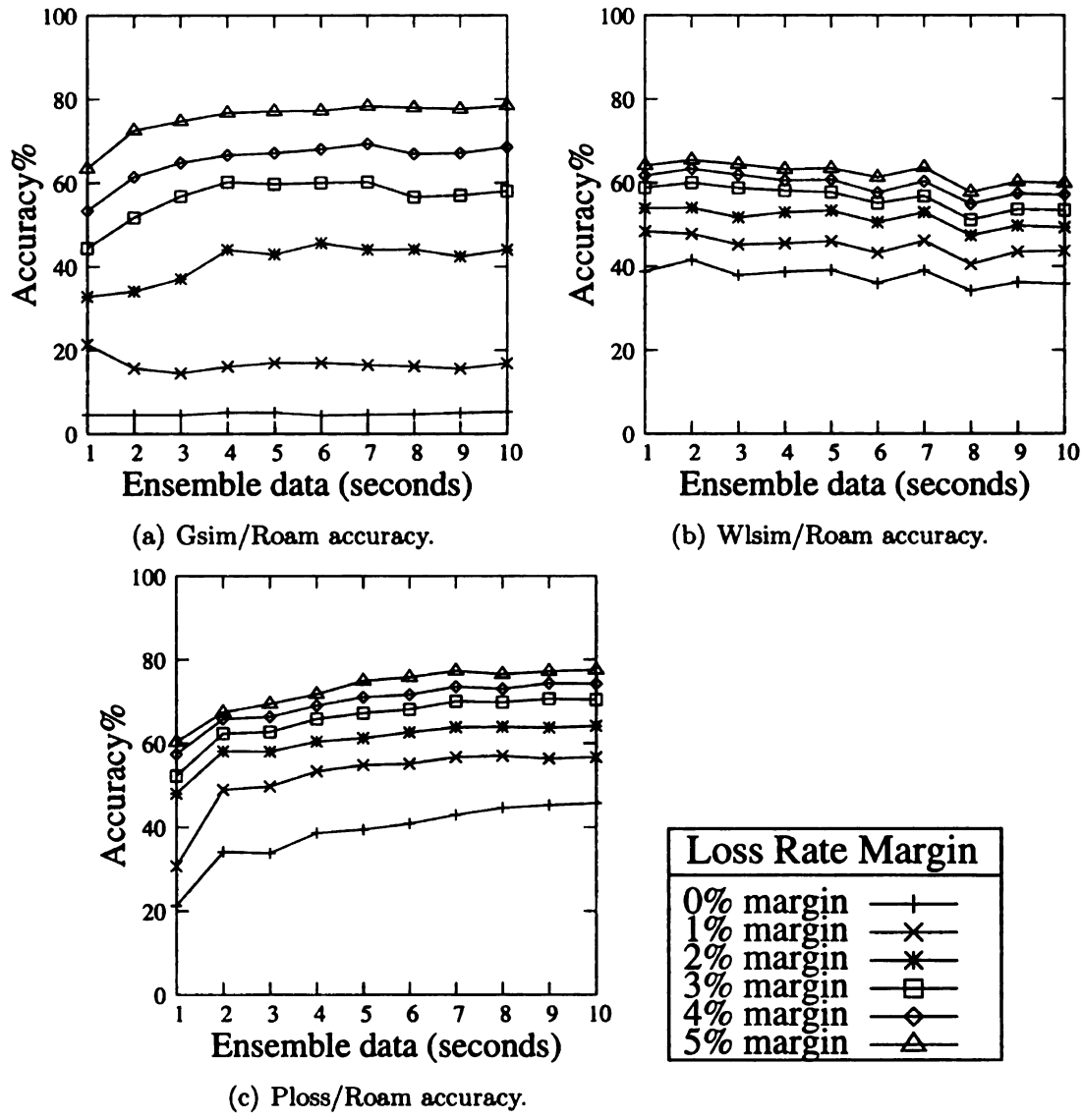


Figure 8.9: MESO forecasting accuracy with loss rate margin. Experiments conducted using the method described in Section 8.4.

roaming are shown in Figure 8.11(b). Data patterns are labeled with FEC code labels. Similar to our results when using margins, when longer periods are used for constructing testing patterns, gsim and wlsim show little improvement, while training



(a) Gsim/Roam accuracy.

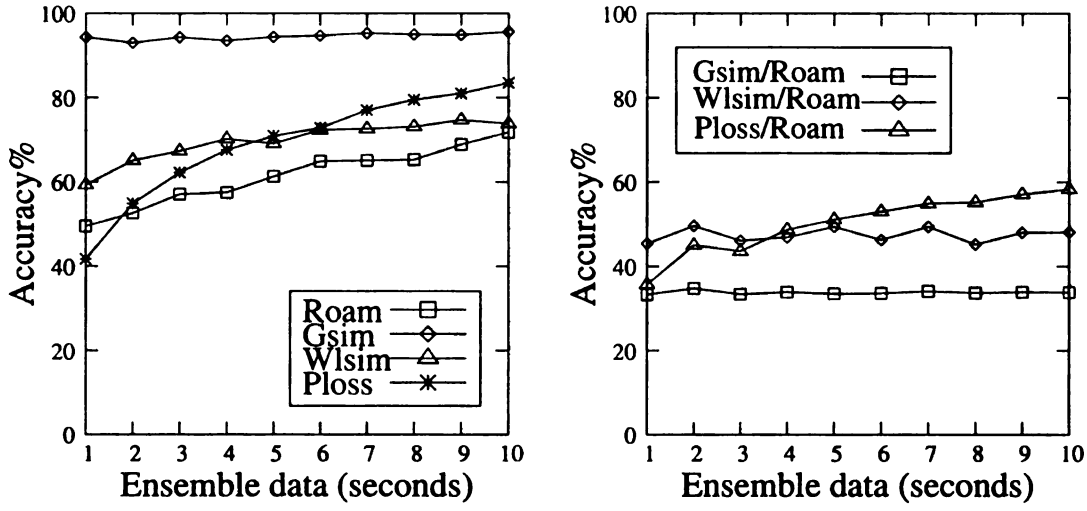
(b) Wlsim/Roam accuracy.

(c) Ploss/Roam accuracy.

Figure 8.10: MESO forecasting accuracy when trained using generated data. Experiments conducted using the method described in Section 8.4.

with ploss data shows a moderate, corresponding increase in accuracy.

For comparison with our Xnaut case study, described in Chapter 6, we relabeled the data sets using the FEC codes used by the Xnaut decision maker during autonomous operation atop a wireless network. Again, the FEC code selected as a label for each pattern is the FEC code that provides the least redundancy that is greater than or equal to the mean ensemble loss rate. Again, for completeness, the FEC



(a) Trained and tested using the same data set. (b) Trained and tested on different data sets.

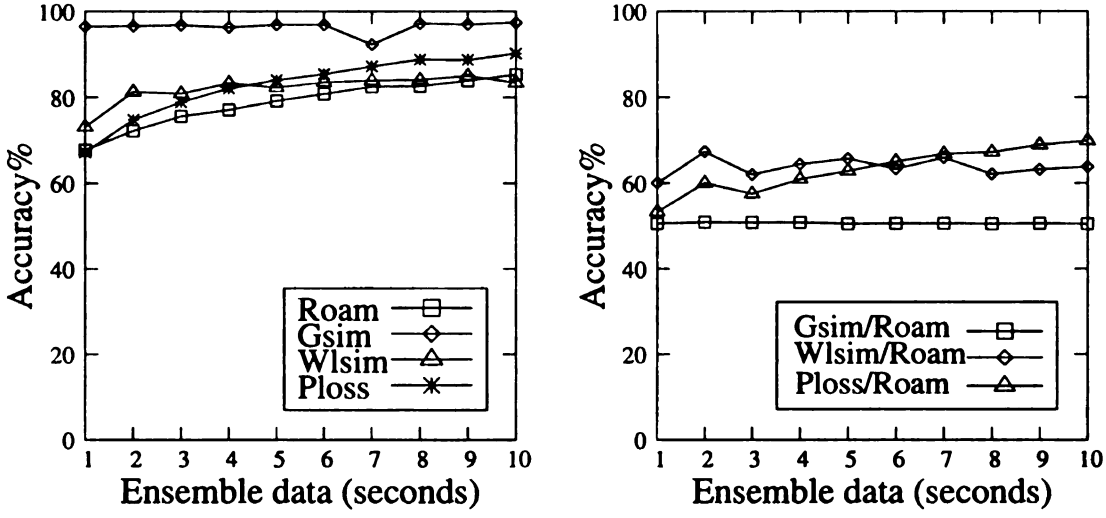
Figure 8.11: MESO forecasting accuracy with FEC code labels. Left, results are produced by training and testing using patterns from the same data set. Right, results are produced by training on generated data and testing on data from the roam data set. Note, when using the same data set for training and testing, training and testing data does not overlap. Experiments conducted using the method described in Section 8.4.

(n, k) combinations used by the Xnaut are:

$$(1,1) \quad (4,2) \quad (6,2) \quad (8,2) \quad (10,2)$$
$$(12,2) \quad (14,2) \quad (16,2) \quad (18,2).$$

Figure 8.12(a) shows MESO forecasting accuracy when using Xnaut FEC code labels. Notably, accuracy increases as longer periods are used for constructing test patterns. Moreover, by reducing the number of FEC codes, overall accuracy has also increased. Again, gsim has the highest accuracy, 97.4%, when 10 seconds of data is used for constructing testing patterns. Similarly, the roam data set attains an accuracy of 85.3%. These results suggest that, for highly dynamic environments, adaptive actions that address a wide range of conditions may be more viable than those that only target specific situations. Figure 8.12(b), plots our results when

training on generated data and testing using roam data set patterns. Again, only when training with ploss data does accuracy improve.



(a) Trained and tested using the same data set. (b) Trained and tested on different data sets.

Figure 8.12: MESO forecasting accuracy with Xnaut FEC code labels. Left, results are produced by training and testing using patterns from the same data set. Right, results are produced by training on generated data and testing on data from the roam data set. Note, when using the same data set for training and testing, training and testing data does not overlap. Experiments conducted using the method described in Section 8.4.

Depicted in Table 8.4 are our results when evaluating MESO accuracy using coverage and precision. In these experiments, training and testing patterns come from the same data set. For all data sets, both coverage and precision improve with an increase in the period used for constructing test patterns. The best coverage is 0.705, attained using the ploss data set, while 0.616 is attained using the roam data set. The highest precision, 0.976, is attained using the gsim data set, while a precision of 0.955 is attained using the roam data set.

Table 8.5 depicts MESO coverage and precision when generated data is used for training and the roam data set is used for testing. As the period for constructing

Table 8.4: MESO forecasting coverage and precision.

Coverage				
	Data set			
Seconds	Roam	Gsim	Wlsim	Ploss
1 second	0.377±0.037	0.592±0.012	0.382±0.013	0.323±0.011
2 second	0.476±0.044	0.621±0.010	0.496±0.016	0.460±0.011
3 second	0.473±0.029	0.628±0.008	0.560±0.017	0.514±0.011
4 second	0.458±0.021	0.620±0.009	0.586±0.014	0.554±0.010
5 second	0.501±0.021	0.636±0.009	0.593±0.012	0.585±0.011
6 second	0.549±0.018	0.648±0.009	0.631±0.013	0.608±0.011
7 second	0.590±0.017	0.652±0.009	0.637±0.011	0.647±0.009
8 second	0.592±0.019	0.658±0.007	0.661±0.012	0.667±0.009
9 second	0.605±0.019	0.673±0.007	0.685±0.011	0.690±0.009
10 second	0.616±0.016	0.687±0.007	0.692±0.011	0.705±0.008
Precision				
	Data set			
Seconds	Roam	Gsim	Wlsim	Ploss
1 second	0.924±0.008	0.971±0.001	0.937±0.002	0.914±0.002
2 second	0.929±0.006	0.971±0.001	0.954±0.002	0.942±0.001
3 second	0.938±0.004	0.973±0.000	0.964±0.002	0.951±0.001
4 second	0.939±0.005	0.973±0.001	0.967±0.001	0.958±0.001
5 second	0.941±0.003	0.974±0.000	0.968±0.001	0.959±0.001
6 second	0.951±0.002	0.975±0.000	0.970±0.001	0.962±0.001
7 second	0.950±0.003	0.975±0.000	0.972±0.001	0.965±0.001
8 second	0.952±0.002	0.975±0.000	0.974±0.001	0.967±0.001
9 second	0.955±0.002	0.976±0.000	0.975±0.001	0.967±0.001
10 second	0.955±0.002	0.976±0.000	0.974±0.001	0.968±0.001

Experiment conducted using the method described in Section 8.4.

test patterns increases, a corresponding improvement in coverage and precision does not occur when training with gsim or wlsim. For ploss, a moderate improvement in precision occurs with no corresponding increase in coverage.

In this section, we evaluated forecasting packet loss in three ways using three generated and one real roaming trace. We also explored training with generated traces while testing with real roaming data. Since collection of data in real environments can be time consuming and difficult, training a decision maker under simulation can ease the implementation of autonomous software. Our results show that when training

Table 8.5: MESO forecasting coverage and precision when trained on generated data.

Coverage			
	Data set		
Seconds	Gsim/Roam	Wlsim/Roam	Ploss/Roam
1 second	0.299±0.013	0.229±0.025	0.151±0.013
2 second	0.318±0.024	0.342±0.049	0.139±0.011
3 second	0.297±0.021	0.333±0.034	0.129±0.012
4 second	0.295±0.022	0.380±0.040	0.138±0.009
5 second	0.292±0.023	0.393±0.042	0.150±0.011
6 second	0.266±0.020	0.417±0.046	0.152±0.010
7 second	0.281±0.026	0.434±0.050	0.173±0.014
8 second	0.279±0.024	0.442±0.060	0.169±0.013
9 second	0.271±0.025	0.400±0.056	0.167±0.015
10 second	0.270±0.028	0.408±0.054	0.168±0.016
Precision			
	Data set		
Seconds	Gsim/Roam	Wlsim/Roam	Ploss/Roam
1 second	0.935±0.002	0.926±0.006	0.905±0.011
2 second	0.939±0.003	0.934±0.006	0.931±0.004
3 second	0.940±0.003	0.934±0.006	0.936±0.003
4 second	0.941±0.003	0.933±0.012	0.943±0.003
5 second	0.941±0.003	0.933±0.013	0.948±0.002
6 second	0.941±0.003	0.926±0.016	0.950±0.003
7 second	0.941±0.002	0.932±0.016	0.954±0.002
8 second	0.941±0.003	0.924±0.020	0.955±0.002
9 second	0.940±0.004	0.928±0.018	0.955±0.002
10 second	0.941±0.004	0.925±0.017	0.955±0.002

Experiment conducted using the method described in Section 8.4.

and testing using data collected in the same way, forecasting accuracy improves as more ensemble data is used for testing. However, when we trained with generated data and tested with real data, accuracy improved only when training with the ploss data set. Also, when data sets were labeled with FEC (n, k) combinations, higher accuracy was attained than that using loss-rate margins. In short, simulations used for training must accurately represent the conditions found under real operation. In lieu of accurate simulation, training with artificially generated data, that systematically covers a broad range of conditions, may enable better decision making. Moreover,

when adapting in highly dynamic environments, actions that address a wide range of behaviors may be more successful that those that only target specific situations.

8.6 Related Work

Forecasting has been studied in many fields, such as for predicting economic trends [251] or river flow [252]. Many of these studies use techniques based on moving averages (MA) or auto-regression (AR) to construct a time-series model that can be used to predict future time-series evolution. Other approaches, such as the auto-regressive moving average, or ARMA, combine techniques to improve predication accuracy [240]. These techniques rely on the correlation between historic, time-series values to accurately predict the future. While these approaches have been successful in forecasting stationary or slowly-decaying time-series, they are error prone when time series behavior changes rapidly [253]. To address this problem, Ilow [254] used fractional auto-regressive integrated moving average (FARIMA) predictors in conjunction with cepstral [255] techniques to improve estimation of ARIMA parameters and include short-term data dependence. Results demonstrate that this model better predicts network bandwidth requirements. Notably, packet loss, as a user roams about a wireless cell, is highly dynamic and changes suddenly when the user begins walking or the wireless signal is occluded. Thus, historic packet losses may correlate poorly with those in the near term. Ensembles use anomaly detection to automatically recognize transitions in the time-series and extract periods that exhibit similar behavior. Rather than assume that near term behavior correlates with that of the im-

mediate past, our approach forecasts overall ensemble characteristics based on early ensemble data, enabling timely automated decision making.

Cui et al. [253] investigate Gaussian, ARMA and FARIMA predictors in a virtual private network (VPN). The authors investigate dynamically resizing the bandwidth of VPN links based on predicted bandwidth usage. They observe that a Gaussian predictor is more accurate than either the ARMA or FARIMA predictors, attributed to the Gaussian predictor's greater sensitivity to rapid change in bandwidth usage. That is, ARMA and FARIMA predictors depend heavily on historical data, and change slowly due to regression and averaging using past data. Thus, the authors propose a more responsive method, called linear predictor with dynamic error compensation (L-PREDEC), to improve predictor sensitivity in the face of rapid change. Like our technique, L-PREDEC addresses the need for forecasting in dynamic environments that may change rapidly. Instead, we use perceptual memory in conjunction with ensemble extraction rather than linear models that adapt to correct prediction errors. Moreover, further study is required to understand how linear models can be used for decision making when software must respond to changes in very dynamic environments.

Papadopouli et al. [256] study several models for forecasting traffic load on the University of North Carolina at Chapel Hill's wireless network. The simple network management protocol (SNMP) was used to query 488 wireless access points, every five minutes for 63 days. They target middle term forecasting (2-weeks ahead) on individual access points using mean bandwidth and recent history or other predictors, such as ARIMA. Evaluation used a prediction tolerance interval, computed as a

percentage of the mean bandwidth usage. The percentage of correct predictions was typically <35% when using a tolerance interval of 25%. Our work complements studies that investigate forecasting network traffic by enabling automated extraction of ensembles that can be processed using clustering and classification techniques. Moreover, our study addresses near term forecasting of packet loss when roaming, rather than middle term forecasting of network bandwidth.

Other projects have investigated the use of artificial neural networks (NNs) [252, 257] and hidden Markov models (HMMs) [258] for forecasting time-series. Atiya et al. [252] studied forecasting the flow of the River Nile in Egypt using a neural network. The goal was to predict the river flow rate for 10 or more days into the future based on historical data. A artificial neural network was trained for 4000 iterations and tested, producing "fairly accurate forecasts." Fraser et al. [258], investigated HMMs during participation in a Santa Fe Institute and NATO sponsored competition. The contest goal was to explore and compare different forecasting methods. The authors provide an overview of HMM techniques, and investigate hidden filter and mixed state HMMs for forecasting a numerically generated time series constructed for the competition. These HMMs produced good predictions for both short and longer term forecasts. MESO complements these works by investigating forecasting for automated decision making and adaptation of autonomic software. It is likely that HMMs and other approaches can be used to enable autonomic decision making in software, perhaps in conjunction with perceptual memory.

8.7 Discussion

We have presented a technique for extracting ensembles from wireless network traces with the goal of forecasting future ensemble packet loss behavior. Forecasting enables timely, autonomous response by software decision makers in autonomic computer systems. Moreover, automated ensemble extraction enables capture of discrete periods of similar behavior, as described by the distribution of time series values. Results of our forecasting experiments show promise. We learned that if an autonomic decision maker is trained under simulation, care must be taken that the simulation accurately produces conditions similar to those found when the software is operating autonomously. Alternatively, models that systematically generate artificial data may provide better coverage of conditions found under autonomous operation than those provided by simulation models.

Chapter 9

Conclusions and Future Work

Integration of adaptive mechanisms, state maintenance and automated decision making enables implementation of autonomic software. In this dissertation, we investigated the design and integration of these methods and applied them to data streaming applications, an important class of software that includes communications, mobile computing, command-and-control, and environmental monitoring. In this chapter, we summarize our specific contributions and discuss future work.

9.1 Contributions

This research has produced five main contributions:

A programming model that separates intercession and introspection. In the first part of our study, we described and evaluated a programming model that separates intercession and introspection. We developed Adaptive Java, an extension to Java that incorporates programming language constructs to support instrumentation

and dynamic recomposition. Our group used Adaptive Java to design and evaluate the MetaSocket component, whose behavior can be adapted in response to changing network conditions by enabling structural reconfiguration. In addition, we developed Dynamic River, a distributed, data-stream pipeline platform that enables dynamic recomposition of pipeline operators across multiple hosts. Our studies show that the integration of mechanisms that enable both introspection and and intercession are important to the design of autonomic and adaptive software. Where mechanisms for intercession enable dynamic recomposition of software, those for introspection support instrumentation and sensor data collection, which in turn enable automated decision making.

State maintenance in adaptive software. In the second part of our study, we investigated state maintenance at the program and pipeline level. Where *Perimorph* transferred nontransient state during recomposition, *Dynamic River* used protocols in conjunction with *data stream scope* to resynchronize processing following operator reconfiguration. Perimorph addresses collateral change by enabling the declaration of sets of program modifications that must occur atomically when an application is adapted. In a case study, we demonstrate that externalizing state supports application handoff between different devices in a mobile computing environment while enabling recomposition to meet the resource capabilities of different devices. In addition, when data stream processing is distributed among networked hosts, Dynamic River supports graceful recovery in the face of software, host or network failure.

Perceptual memory. In the third part of our study, we investigated the effect of perceptual memory on the autonomous decision making process. Storing and retrieving external stimuli and associated meta information in dynamic environments is typically incremental, data intensive and time sensitive. Moreover, storage and recall must be efficient and avoid impacting the function of the application being adapted while enabling the decision maker to make correct and timely decisions. We have designed and implemented the perceptual memory system, MESO, to address these requirements. We showed that MESO accurately and quickly retrieves prior experience that can be used to help an autonomous decision maker enhance and optimize an underlying adaptable application.

An integrated, autonomic application. In the fourth part of our study, we investigated and evaluated the integration and application of adaptive mechanisms, state maintenance and decision making to adaptive software and data streaming. We conducted a case study using an autonomous, adaptive application, called Xnaut. We show that a decision maker can learn through interaction with a user to operate completely autonomously while attempting to adapt and optimize an underlying mobile computing application. When we examined which pattern features were most significant for deciding which adaptations to invoke, we discovered that instantaneous sensor readings were of little importance for making decisions that balance packet loss with bandwidth consumption. Instead, statistical measures computed over multiple instantaneous readings prove most useful when deciding how and when to adapt the application.

Automated analysis of sensor data streams. Finally, we introduced a technique for automated extraction and analysis of ensembles from sensor data streams. We investigated the utility of using ensembles for classification and detection of bird species using acoustic data streams and for forecasting near-term, packet-loss behavior when streaming data to a mobile receiver. Our investigations showed that ensembles improve classification, detection and forecasting of time-series events, such as those that can be used by autonomic decision makers when adapting an application.

Figure 9.1 summarizes the contributions discussed in this dissertation as a whole, and shows how they are related and how they fit into our integrated approach to autonomous computation. Three major elements must be considered during design and implementation: adaptive mechanisms, state maintenance, and decision making. Our investigations uncovered abstractions that enabled each of these elements to be further refined.

State maintenance can be further refined by normalized state transfer and collateral change. Using Perimorph, we studied state maintenance for component-based software and learned that normalized state extraction can be used to transfer state to a new component during component exchange. However, dynamic recomposition must also be orchestrated to avoid incorrect program operation or data loss. Perimorph enabled collateral change to be codified using factor sets. On the other hand, Dynamic River provides protocols, in conjunction with data stream scoping, to orchestrate redeployment and recomposition of data stream processing.

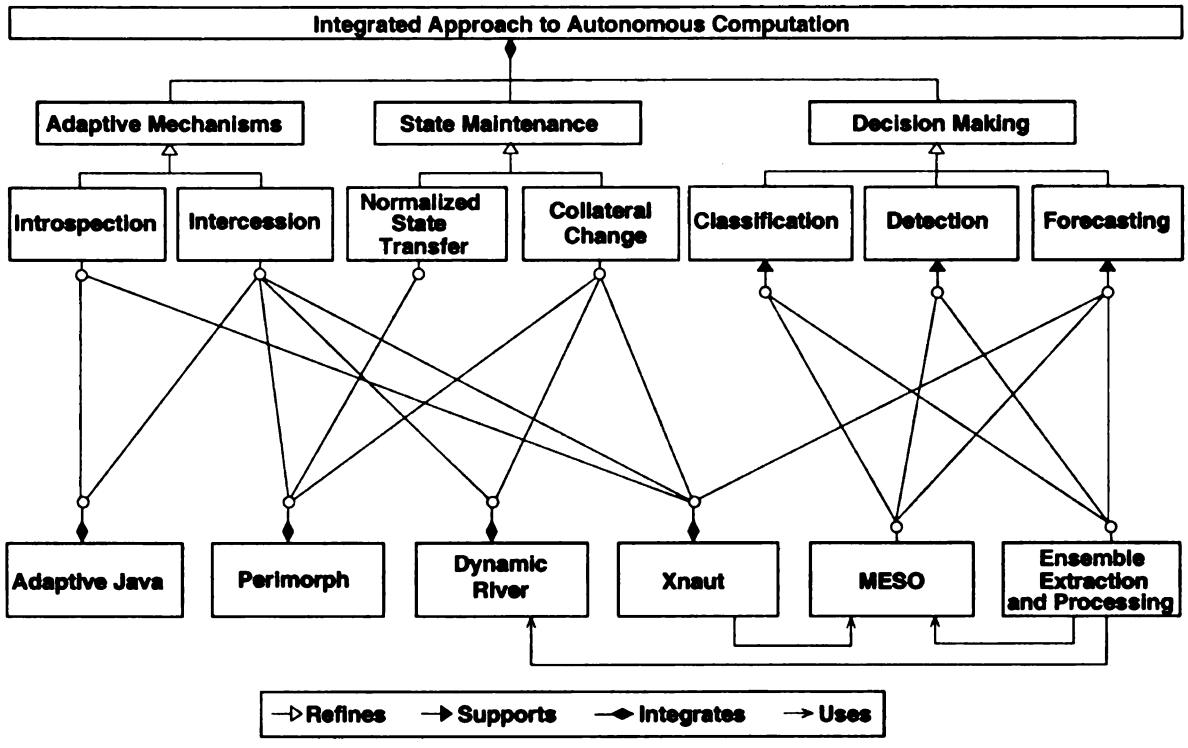


Figure 9.1: Achievements presented in this dissertation viewed as a whole.

Decision making using perceptual memory can be further refined as supporting classification, detection or forecasting. Using acoustic data streams, we studied classification and detection to better understand the difficulties of distilling information, useful for making decisions, from raw data. An analysis of the pattern features used for our Xnaut case study revealed that metrics computed using multiple instantaneous sensor readings were most significant. This lesson prompted further analysis of sensor data to better support autonomous decision making, and led to the development of our technique for extracting ensembles from sensor data streams. We learned that perceptual memory, used in conjunction with automated ensemble extraction and processing, can enable decision making for autonomous software.

9.2 Future Work

The work presented in this dissertation has revealed directions for further research. We identify five research topics that need to be addressed (and in some cases are already under study) in the design of autonomic software.

Adaptive protocols and techniques. The set of protocols and techniques described in this dissertation can be expanded. For instance, programmers that must implement an adaptable application will benefit from design patterns that describe accepted methods for dynamically recomposing applications. Documented protocols and other methods for orchestrating runtime recomposition and capturing collateral change are also needed. Moreover, there are many different application classes, and each may require different techniques to enable autonomous adaptation. For instance, protocols that target distributed recomposition of pipeline operators may not be effective when recomposing components.

Safety and security when adapting. Runtime composition complicates ensuring the safety and security of an application [52, 259]. Ensuring safety requires that the software continues to execute acceptably during and after an adaptation. That is, when software undergoes structural change during execution it often exhibits complex and changing interactions that can be difficult to coordinate. Such interaction problems can be difficult to characterize, particularly when the number of possible compositions is large. Nonviable compositions need to be recognized prior to their use to ensure that the adaptation recommended by a decision maker preserves the

correctness and safety properties required of the original system. Ensuring security requires that an adaptation does not expose a breach that can be exploited by a malicious entity. Similar to safety, ensuring security is often difficult due to the complex interactions that exist between different parts of a running program. Autonomic software needs methods for ensuring that the adaptive process will produce a safe and secure system.

Extending ensemble extraction and processing to other domains. We presented ensemble extraction as applied specifically to acoustic data streams and network traces. However, the process for extracting ensembles is general and can be extended to other types of streaming sensor data and other application domains. For instance, automated detection of mechanical problems, such as bearing failure [218], enables preemptive response and prevention of accidents. Moreover, forecasting of catastrophic events, such as the eruption of a volcano, enables early response or evacuation. We anticipate that automated ensemble extraction may prove to be a useful analysis tool in many disciplines.

Ensemble extraction from multiple sensor streams. Currently, we have extracted ensembles from data streams comprising a single signal. Although acoustic data streams are data rich, extracting ensembles from multiple correlated data streams may enhance classification, detection and forecasting of time series events. For instance, species identification may be more accurate when acoustic data is coupled with geographic, weather or other information about the environment. Moreover,

monitoring the health of an organization's or nation's cyberinfrastructure will require the acquisition and correlation of data from many sensors to capture the complex behavior afforded by multiple interacting components, systems and networks. Autonomous decision makers will require accurate and timely extraction, detection and identification of sensed events to correctly respond to faults and attacks.

Automated programming for adaptation. Approaches that use genetic programming [260, 261] or digital evolution [262] operate on sequences of instructions that are reminiscent of computer machine code or higher level language constructs. Programs evolved in these languages can be interpreted much like a traditional computer program. Decision makers might use such automated approaches to construct adaptable components that address specific sensed conditions or to better interpret sensor data. That is, a decision maker that must detect and respond to the song of a specific bird species could request the automated construction of a customized detector. Moreover, some sensor data streams may have a syntactic structure that is better captured as a series of programmatic steps. For instance, many bird songs have a syntax that is useful for recognizing individual birds and members of a specific species. Approaches like digital evolution may enable automated generation of components and tools for analyzing and adapting to sensed conditions and improve decision making.

Appendices

Appendix A

Perimorph Data Dictionary of Major Components

This appendix provides an elided data dictionary for the major components of the Perimorph API. This data dictionary describes the public and protected component interfaces, including private methods and variables only when they may improve understanding. Some of the less used methods and components, such as those that implement visitors or throwable errors, have been omitted to make this appendix more concise.

A.1 BaseComponentDefinition

Description:	Basic component definition that implements functionality needed by all Perimorph components. This class is abstract and should be inherited by application specific components.
Relationships:	Extends: Serializable

BaseComponentDefinition() [protected]

Description: Default constructor for this class.
Returns: this

BaseComponentDefinition(String aName) `[public]`

Description: Constructor that specifies a name for this component instance.

Parameters:

- *aName* – A name for this component instance.

Returns: this

void setName(String aName) `[protected]`

Description: Set the name for this component instance.

Parameters:

- *aName* – A name for this component instance.

Returns: None.

String getName() `[public]`

Description: Return the name for this component instance.

Returns: The name for this component instance.

ObjectReference getReference() `[protected]`

Description: Return an `ObjectReference` for this component. An ObjectReference acts as a proxy for this component, enabling indirect execution of component factors.

Returns: An `ObjectReference` for this component.

A.2 BaseFactor

Description:	Basic factor definition that implements functionality needed by all Perimorph factors. This class is abstract and should be inherited by application specific factors.
Relationships:	Extends: Serializable

Scope self() `[protected]`

Description: Return this factor's current execution scope. This scope can contain component specific variables and method parameters.

Returns: The current execution scope.

boolean invoke() `[public, abstract]`

Description: Execute this factor's function using the current factor execution scope.

Returns: True if execution was successful.

A.3 BaseFactorContext

Description: This class defines a factor specific context.

Atrributes:

- `BaseFactor factor [public]`
  Factor associated with this context.
- `FactorsetVars variables [public]`
  Factor specific variables.

Relationships: Extends: Serializable

BaseFactorContext(BaseFactor aFactor,FactorsetVars aVars) `[public]`

Description: Constructor that specifies the factor and variables belonging to this context. The factor specified is the one that will use this context.

Parameters:

- *aFactor* – The factor that will use this context.
- *aVars* – The variables that will be used by `aFactor`.

Returns: this

A.4 BaseFactorset

Description: A set of factors and variables that can be associated with components. This class is abstract and should be inherited by application specific factor sets.

Atrributes:

- `long refcnt [private]`
  Reference count for this factor set.
- `HashMap factorcollection [private]`
  Collection of named factors.
- `FactorsetVars variables [protected]`
  Set of factor set variables.
- `String factorsetname [protected]`
  Name for this factor set.

Relationships: Extends: Serializable

BaseFactorset(String aName) `[public]`

Description: Construct a factor set with the specified name.

Parameters:

- *aName* – A name for this factor set.

Returns: this

void setName(String aName) `[protected]`

Description: Set this factor set's name.

Parameters:

- *aName* – A name for this factor set.

Returns: None.

String getName() `[public]`

Description: Get this factor set's name.

Returns: The name for this factor set.

FactorsetVars getVariables() `[public]`

Description: Get the variables associated with this factor set.

Returns: The variables associated with this factor set.

void addFactor(BaseFactor aFactor) `[public]`

Description: Add a factor to this factor set.

Parameters:

- *aFactor* – The factor to add to this factor set.

Returns: None.

void removeFactor(String aName) `[public]`

Description: Remove a factor from this factor set by name.

Parameters:

- *aName* – The name of the factor to remove.

Returns: None.

BaseFactor getFactor(String aName) `[public]`

Description: Get a factor from this factor set by name.

Parameters:

- *aName* – The name of the factor to retrieve.

Returns: The factor requested.

void initialize() `[protected, abstract]`

Description: Initialize this factor set.

Returns: None.

void activate() `[protected, abstract]`

Description: Activate this factor set. That is, application specific bootstrap processing can be codified by this method. For instance, code for initial display of graphical interfaces might be included in this method.

Returns: None.

void deactivate() [protected, abstract]

Description: Deactivate this factor set. That is, application specific component termination processing can be codified by this method. For instance, code for removal of graphical interfaces from the desktop might be included in this method.

Returns: None.

long getRefCount() [public]

Description: Get the current reference count for this factor set. The `FactorManager` typically activates and deactivates factor sets when the reference count rises above or drops below zero, respectively.

Returns: The current reference count.

void incRefCount() [protected]

Description: Increment the reference count for this factor set.

Returns: None.

void decRefCount() [protected]

Description: Decrement the reference count for this factor set.

Returns: None.

void setState(Object aState) [protected, abstract]

Description: Set this factor set's nontransient state using a state memento.

Parameters:

- *aState* – A state memento for this factor set.

Returns: None.

Object getState() [public, abstract]

Description: Extract this factor's nontransient state as a memento.

Returns: The state memento for this factor set.

A.5 BaseInterface

Description:	Basic interface definition that contains invocations and pre and post factors. This component enables reconfiguration of a component's interface.
Atrributes:	• `HashMap invocation_map [private]` Hash map containing invocations.
Relationships:	Extends: Serializable

void addInvocation(String aSigAlias) `[public]`

Description: Add an invocation to this interface.

Parameters:

- *aSigAlias* – The signature of the invocation (stored in `invocation_map`).

Returns: None.

void removePreFactor(String aSigAlias,String Factor) `[public]`

Description: Remove a pre factor from an invocation.

Parameters:

- *aSigAlias* – The signature of the invocation.
- *aFactor* – The name of the pre factor to remove.

Returns: None.

void removePostFactor(String aSigAlias,String Factor) `[public]`

Description: Remove a post factor from an invocation.

Parameters:

- *aSigAlias* – The signature of the invocation.
- *aFactor* – The name of the post factor to remove.

Returns: None.

void prependPreFactor(String aSigAlias,String Factor) `[public]`

Description: Prepend a pre factor to an invocation.

Parameters:

- *aSigAlias* – The signature of the invocation.
- *aFactor* – The name of the pre factor to prepend.

Returns: None.

void appendPreFactor(String aSigAlias,String Factor) `[public]`

Description: Prepend a pre factor to an invocation.

Parameters:

- *aSigAlias* – The signature of the invocation.
- *aFactor* – The name of the pre factor to append.

Returns: None.

void replacePreFactor(String aSigAlias,String Factor,String aNewFactor) `[public]`

Description: Replace an invocation pre factor.

Parameters:

- *aSigAlias* – The signature of the invocation.
- *aFactor* – The name of the pre factor to replace.
- *aNewFactor* – The name of the replacement pre factor.

Returns: None.

void prependPostFactor(String aSigAlias,String Factor) `[public]`

Description: Prepend a post factor to an invocation.

Parameters:

- *aSigAlias* – The signature of the invocation.
- *aFactor* – The name of the post factor to prepend.

Returns: None.

void appendPostFactor(String aSigAlias,String Factor) `[public]`

Description: Prepend a post factor to an invocation.

Parameters:

- *aSigAlias* – The signature of the invocation.
- *aFactor* – The name of the post factor to append.

Returns: None.

void replacePostFactor(String aSigAlias,String Factor,String aNewFactor) `[public]`

Description: Replace an invocation post factor.

Parameters:

- *aSigAlias* – The signature of the invocation.
- *aFactor* – The name of the post factor to replace.
- *aNewFactor* – The name of the replacement pre factor.

Returns: None.

void invokeFactors(String aSigAlias) `[public]`

Description: Execute an invocation's pre and post factors.

Parameters:

- *aSigAlias* – The signature of the invocation.

Returns: None.

A.6 ComponentFactorStore

Description:	A repository of pre and post factors and their component associations.
Atrributes:	• `HashMap pre_factorcollection [private]` Collection of component pre factors. • `HashMap post_factorcollection [private]` Collection of component post factors.
Relationships:	Extends: Serializable

ComponentFactorStore() [protected]

Description: Default constructor.

Returns: this

void removePreFactor(String aComp,String aSig,String aFactor) [protected, static]

Description: Remove a pre factor from a component.

Parameters:

- *aComp* – The component name from which to remove a pre factor.
- *aSig* – The interface signature from which to remove a pre factor.
- *aFactor* – The name of the factor to remove.

Returns: None.

void removePostFactor(String aComp,String aSig,String aFactor) [protected, static]

Description: Remove a post factor from a component.

Parameters:

- *aComp* – The component name from which to remove a post factor.
- *aSig* – The interface signature from which to remove a post factor.
- *aFactor* – The name of the factor to remove.

Returns: None.

void prependPreFactor(String aComp,String aSig,String aFactor) `[protected, static]`

Description: Prepend a pre factor to a component.

Parameters:

- *aComp* – The name of the component to use when prepending a pre factor.
- *aSig* – The interface signature to use when prepending a pre factor.
- *aFactor* – The name of the factor to prepend.

Returns: None.

void appendPreFactor(String aComp,String aSig,String aFactor) `[protected, static]`

Description: Append a pre factor to a component.

Parameters:

- *aComp* – The name of the component to use when appending a pre factor.
- *aSig* – The interface signature to use when appending a pre factor.
- *aFactor* – The name of the factor to append.

Returns: None.

void replacePreFactor(String aComp,String aSig,String aFactor,String aNewFactor) `[protected, static]`

Description: Replace a component pre factor.

Parameters:

- *aComp* – The name of the component to use during factor replacement.
- *aSig* – The interface signature to use during factor replacement.
- *aFactor* – The name of the factor to replace.
- *aNewFactor* – The name of the replacement factor.

Returns: None.

void prependPostFactor(String aComp,String aSig,String aFactor) `[protected, static]`

Description: Prepend a post factor to a component.

Parameters:

- *aComp* – The name of the component to use when prepending a post factor.
- *aSig* – The interface signature to use when prepending a post factor.
- *aFactor* – The name of the factor to prepend.

Returns: None.

void appendPostFactor(String aComp,String aSig,String aFactor) `[protected, static]`

Description: Append a post factor to a component.

Parameters:

- *aComp* – The name of the component to use when appending a post factor.
- *aSig* – The interface signature to use when appending a post factor.
- *aFactor* – The name of the factor to append.

Returns: None.

void replacePostFactor(String aComp,String aSig,String aFactor,String aNewFactor) `[protected, static]`

Description: Replace a component post factor.

Parameters:

- *aComp* – The name of the component to use during factor replacement.
- *aSig* – The interface signature to use during factor replacement.
- *aFactor* – The name of the factor to replace.
- *aNewFactor* – The name of the replacement factor.

Returns: None.

Vector getPreFactors(String aComp,String aSig) `[protected, static]`

Description: Get the pre factors associated with a specific component and interface signature.

Parameters:

- *aComp* – The name of the component.
- *aSig* – The interface signature.

Returns: A vector containing the pre factors for the specified component.

Vector getPostFactors(String aComp,String aSig) `[protected, static]`

Description: Get the post factors associated with a specific component and interface signature.

Parameters:

- *aComp* – The name of the component.
- *aSig* – The interface signature.

Returns: A vector containing the post factors for the specified component.

A.7 ComponentManager

Description:	This class enables public access and manipulation of the ComponentStore.
Relationships:	Extends: Object

ComponentManager() `[protected]`

Description: Default constructor.

Returns: this

void addComponent(BaseComponentDefinition aComponent) `[public, static]`

Description: Add a component to the component store.

Parameters:

- *aComponent* – The component to add to the component store.

Returns: None.

void loadComponent(String aName) `[public, static]`

Description: Dynamically loads the component with the name specified and adds it to the component store.

Parameters:

- *aName* – The name of the component to load.

Returns: None.

void removeComponent(String aComponent) `[public, static]`

Description: Removes a component from the component store.

Parameters:

- *aComponent* – The name of the component to remove.

Returns: None.

BaseComponentDefinition getComponent(String aComponent) `[public, static]`

Description: Retrieve a component from the component store.

Parameters:

- *aComponent* – The name of the component to retrieve.

Returns: The component with the specified name.

void replicateComponent(String aFromComponent,String aToComponent) `[public, static]`

Description: Replicate a component under a new name and add it to the component store.

Parameters:

- *aFromComponent* – The name of the component to replicate.
- *aToComponent* – The name of the new component.

Returns: None.

ObjectReference newInstance(String aFromComponent,aToComponent,Class[] argtyps,Object[] args) `[public, static]`

Description: Create a new instance of a component and instantiate it with the arguments specified.

Parameters:

- *aFromComponent* – The name of the component for which to create a new instance.
- *aToComponent* – The name of the new component instance.
- *argtyps* – The parameter types to use when constructing the new instance.
- *args* – The parameter values to use when constructing the new instance.

Returns: A reference to the new instance.

ObjectReference getReference(String aComp) `[public, static]`

Description: Get a reference to an existing component.

Parameters:

- *aComp* – The name of the component for which to retrieve a reference.

Returns: A reference to a component.

void addInterface(String aComp,BaseInterface aISet) `[public, static]`

Description: Adds an interface to a component.

Parameters:

- *aComp* – The name of the affected component.
- *aISet* – The interface to add.

Returns: None.

void prependPreFactor(String aComp,String aSig,String aFactor) [protected, static]

Description: Prepend a pre factor to a component.

Parameters:

- *aComp* – The name of the component to use when prepending a pre factor.
- *aSig* – The interface signature to use when prepending a pre factor.
- *aFactor* – The name of the factor to prepend.

Returns: None.

void appendPreFactor(String aComp,String aSig,String aFactor) [protected, static]

Description: Append a pre factor to a component.

Parameters:

- *aComp* – The name of the component to use when appending a pre factor.
- *aSig* – The interface signature to use when appending a pre factor.
- *aFactor* – The name of the factor to append.

Returns: None.

void replacePreFactor(String aComp,String aSig,String aFactor,String aNewFactor) [protected, static]

Description: Replace a component pre factor.

Parameters:

- *aComp* – The name of the component to use during factor replacement.
- *aSig* – The interface signature to use during factor replacement.
- *aFactor* – The name of the factor to replace.
- *aNewFactor* – The name of the replacement factor.

Returns: None.

void prependPostFactor(String aComp,String aSig,String aFactor) [protected, static]

Description: Prepend a post factor to a component.

Parameters:

- *aComp* – The name of the component to use when prepending a post factor.
- *aSig* – The interface signature to use when prepending a post factor.
- *aFactor* – The name of the factor to prepend.

Returns: None.

void appendPostFactor(String aComp,String aSig,String aFactor) `[protected, static]`

Description: Append a post factor to a component.

Parameters:

- *aComp* – The name of the component to use when appending a post factor.
- *aSig* – The interface signature to use when appending a post factor.
- *aFactor* – The name of the factor to append.

Returns: None.

void replacePostFactor(String aComp,String aSig,String aFactor,String aNewFactor) `[protected, static]`

Description: Replace a component post factor.

Parameters:

- *aComp* – The name of the component to use during factor replacement.
- *aSig* – The interface signature to use during factor replacement.
- *aFactor* – The name of the factor to replace.
- *aNewFactor* – The name of the replacement factor.

Returns: None.

void removePreFactor(String aComp,String aSigAlias,String Factor) `[public]`

Description: Remove a pre factor from a component's invocation.

Parameters:

- *aComp* – The name of the component to use during factor removal.
- *aSigAlias* – The signature of the invocation.
- *aFactor* – The name of the pre factor to remove.

Returns: None.

void removePostFactor(String aComp,String aSigAlias,String Factor) `[public]`

Description: Remove a post factor from a component's invocation.

Parameters:

- *aComp* – The name of the component to use during factor removal.
- *aSigAlias* – The signature of the invocation.
- *aFactor* – The name of the post factor to remove.

Returns: None.

Object invoke(String aComp,ObjectReference aThis,String aName,Class[] argtyps,Object[] args) `[public, static]`

Description: Invoke a component method using a specific reference.

Parameters:

- *aComp* – The name of the component.
- *aThis* – The reference to use during execution.
- *aName* – The name of the interface invocation to invoke.
- *argtyps* – The invocation parameter types.
- *args* – The invocation parameter values.

Returns: None.

A.8 ComponentStore

Description:	Repository of Perimorph components.
Atrributes:	• `HashMap componentcollection [private]` Collection of named components.
Relationships:	Extends: Serializable

ComponentStore() `[protected]`

Description: Default constructor.

Returns: this

void addComponent(BaseComponentDefinition aComponent) `[protected, static]`

Description: Add a component to the repository.

Parameters:

- *aComponent* – The component to add to the repository.

Returns: None.

void removeComponent(String aComponent) `[public, static]`

Description: Removes a component from the repository.

Parameters:

- *aComponent* – The name of the component to remove.

Returns: None.

BaseComponentDefinition getComponent(String aComponent) `[public, static]`

Description: Retrieve a component from the repository.

Parameters:

- *aComponent* – The name of the component to retrieve.

Returns: The requested component.

A.9 DynamicLoadManager

Description:	Enables the dynamic loading of factors and components.
Relationships:	Extends: ClassLoader

BaseFactor loadFactor(String aName) `[public, static]`

Description: Dynamically load a Perimorph factor.

Parameters:

- *aName* – The name of the factor to load.

Returns: An instance of the dynamically loaded factor.

BaseFactorset loadFactorset(String aName) `[public, static]`

Description: Dynamically load a Perimorph factor set.

Parameters:

- *aName* – The name of the factor set to load.

Returns: An instance of the dynamically loaded factor set.

BaseComponentDefinition loadComponentDefinition(String aName) `[public, static]`

Description: Dynamically load a Perimorph component.

Parameters:

- *aName* – The name of the component to load.

Returns: An instance of the dynamically loaded component.

A.10 FactorManager

Description:	This class enables public access and manipulation of the FactorStore.
Relationships:	Extends: Object

FactorManager() `[protected]`

Description: Default constructor.

Returns: this

void addFactorset(BaseFactorset aFactorset) [public, static]

Description: Add a factor set to the factor store.

Parameters:

- *aFactorset* – The factor set to add to the factor store.

Returns: None.

void loadFactorset(String aName) [public, static]

Description: Dynamically load a serialized factor set and add it to the factor store.

Parameters:

- *aName* – The name of the factor set to dynamically load.

Returns: None.

void removeFactorset(String aFactorset) [public, static]

Description: Removes a factor set from the factor store.

Parameters:

- *aName* – The name of the factor set to remove.

Returns: None.

BaseFactorset getFactorset(String aFactorset) [public, static]

Description: Retrieve a factor set from the factor store.

Parameters:

- *aFactorset* – The name of the factor set to retrieve.

Returns: The factor set retrieved from the factor store.

void assignFactorset(String aLHFactorset,String aRHFactorset) [public, static]

Description: Assign one factor set to another.

Parameters:

- *aLHFactorset* – The left-hand factor set (the target of the assignment).
- *aRHFactorset* – The right-hand factor set.

Returns: None.

void replicateFactorset(String aFromFactorset,String aToFactorset) [public, static]

Description: Replicate a factor set under a new name.

Parameters:

- *aFromFactorset* – The name of the source factor set to be replicated.
- *aToFactorset* – The name of the destination factor set.

Returns: None.

void removeFactor(String aName) `[public, static]`

Description: Removes a factor from the factor store.

Parameters:

- *aName* – The name of the factor to remove.

Returns: None.

void addFactor(String aFactorset,BaseFactor aFactor) `[public, static]`

Description: Add a factor to a factor set.

Parameters:

- *aFactorset* – The name of the factor set.
- *aFactor* – The name of the factor to add to the specified factor set.

Returns: None.

void loadFactor(String aFactorset,String aName) `[public, static]`

Description: Dynamically load a factor and add it to a factor set.

Parameters:

- *aFactorset* – The name of the affected factor set.
- *aName* – The name of the factor do dynamically load.

Returns: None.

BaseFactorContext getFactor(String aName) `[public, static]`

Description: Retrieve a factor and its context from the factor store.

Parameters:

- *aName* – The name of the factor to retrieve.

Returns: The retrieve factor and its context.

void removeComponentPreFactor(String aComp,String aSig,String aFactor) `[public, static]`

Description: Removes a pre factor from a component interface.

Parameters:

- *aComp* – The name of the component.
- *aSig* – The interface signature from which to remove a factor.
- *aFactor* – The name of the factor to remove.

Returns: None.

void removeComponentPostFactor(String aComp,String aSig,String aFactor) `[public, static]`

Description: Removes a post factor from a component interface.

Parameters:

- *aComp* – The name of the component.
- *aSig* – The interface signature from which to remove a factor.
- *aFactor* – The name of the factor to remove.

Returns: None.

void prependComponentPreFactor(String aComp,String aSig,String aFactor) `[public, static]`

Description: Prepend a pre factor to a component interface.

Parameters:

- *aComp* – The name of the component.
- *aSig* – The affected interface signature.
- *aFactor* – The name of the factor to prepend.

Returns: None.

void appendComponentPreFactor(String aComp,String aSig,String aFactor) `[public, static]`

Description: Append a pre factor to a component interface.

Parameters:

- *aComp* – The name of the component.
- *aSig* – The affected interface signature.
- *aFactor* – The name of the factor to append.

Returns: None.

void replaceComponentPreFactor(String aComp,String aSig,String aFactor,String aNewFactor) `[public, static]`

Description: Replace component interface pre factor.

Parameters:

- *aComp* – The name of the component.
- *aSig* – The affected interface signature.
- *aFactor* – The name of the factor to replace.
- *aNewFactor* – The replacement factor.

Returns: None.

void prependComponentPostFactor(String aComp,String aSig,String aFactor) `[public, static]`

Description: Prepend a post factor to a component interface.

Parameters:

- *aComp* – The name of the component.
- *aSig* – The affected interface signature.
- *aFactor* – The name of the factor to prepend.

Returns: None.

void appendComponentPostFactor(String aComp,String aSig,String aFactor) `[public, static]`

Description: Append a post factor to a component interface.

Parameters:

- *aComp* – The name of the component.
- *aSig* – The signature of the affected interface.
- *aFactor* – The name of the factor to append.

Returns: None.

void replaceComponentPostFactor(String aComp,String aSig,String aFactor,String aNewFactor) `[public, static]`

Description: Replace component interface post factor.

Parameters:

- *aComp* – The name of the component.
- *aSig* – The affected interface signature.
- *aFactor* – The name of the factor to replace.
- *aNewFactor* – The replacement factor.

Returns: None.

Vector getComponentPreFactors(String aComp,String aSig) `[public, static]`

Description: Retrieve a component's pre factors.

Parameters:

- *aComp* – The name of the target component.
- *aSig* – The target interface signature.

Returns: A vector containing the specified component's pre factors.

Vector getComponentPostFactors(String aComp,String aSig) `[public, static]`

Description: Retrieve a component's post factors.

Parameters:

- *aComp* – The name of the target component.
- *aSig* – The target interface signature.

Returns: A vector containing the specified component's post factors.

void activateFactorset(String aName) [public, static]

Description: Activate a factor set.

Parameters:

- *aName* – The name of the factor set to activate.

Returns: None.

void deactivateFactorset(String aName) [public, static]

Description: Deactivate a factor set.

Parameters:

- *aName* – The name of the factor set to deactivate.

Returns: None.

A.11 FactorStore

Description:	A repository of Perimorph factors and factor sets.
Atrributes:	• `HashMap factorcollection [private]` Collection of factors.
Relationships:	Extends: Serializable

FactorStore() [protected]

Description: Default constructor.

Returns: this

void addFactorset(BaseFactorset aFactorset) [protected, static]

Description: Add a factor set to the repository.

Parameters:

- *aFactorset* – The factor set to add to the repository.

Returns: None.

void removeFactorset(String aFactorset) [protected, static]

Description: Remove a factor set from the repository.

Parameters:

- *aFactorset* – The name of the factor set to remove.

Returns: None.

BaseFactorset getFactorset(String aFactorset) [protected, static]

Description: Get a factor set from the repository.

Parameters:

- *aFactorset* – The name of the factor set to retrieve.

Returns: The factor set retrieved.

void removeFactor(String aName) `[protected, static]`

Description: Removes a factor from the repository.

Parameters:

- *aName* – The name of the factor to remove.

Returns: None.

void addFactor(String aFactorset,BaseFactor aFactor) `[protected, static]`

Description: Adds a factor to a factor set.

Parameters:

- *aFactorset* – The name of the affected factor set.
- *aFactor* – The factor to add to the specified factor set.

Returns: None.

BaseFactorContext getFactor(String aName) `[protected, static]`

Description: Get a factor and its context from the repository.

Parameters:

- *aName* – The name of the factor to retrieve.

Returns: The retrieved factor and its context.

A.12 FactorsetVars

Description:	A set of variables specific to a particular factor set.
Atrributes:	• `HashMap variables_map [private]` Set of named factor set variables.
Relationships:	Extends: Serializable

void setVar(String aName,boolean aBool) `[public]`

Description: Assign a boolean value to a named variable.

Parameters:

- *aName* – The variable name.
- *aBool* – The boolean value.

Returns: None.

void setVar(String aName,boolean[] aBool) `[public]`

Description: Assign a boolean array to a named variable.

Parameters:

- *aName* – The variable name.
- *aBool* – The boolean array.

Returns: None.

void setVar(String aName,byte aByte) `[public]`

Description: Assign a byte value to a named variable.

Parameters:

- *aName* – The variable name.
- *aByte* – The boolean value.

Returns: None.

void setVar(String aName,byte[] aByte) `[public]`

Description: Assign a byte array to a named variable.

Parameters:

- *aName* – The variable name.
- *aByte* – The byte array.

Returns: None.

void setVar(String aName,char aChar) `[public]`

Description: Assign a character value to a named variable.

Parameters:

- *aName* – The variable name.
- *aChar* – The character value.

Returns: None.

void setVar(String aName,char[] aChar) `[public]`

Description: Assign a character array to a named variable.

Parameters:

- *aName* – The variable name.
- *aChar* – The character array.

Returns: None.

void setVar(String aName,short aShort) `[public]`

Description: Assign a short integer value to a named variable.

Parameters:

- *aName* – The variable name.
- *aShort* – The short integer value.

Returns: None.

void setVar(String aName,short[] aShort) `[public]`

Description: Assign a short integer array to a named variable.

Parameters:

- *aName* – The variable name.
- *aShort* – The short integer array.

Returns: None.

void setVar(String aName,int aInt) `[public]`

Description: Assign an integer value to a named variable.

Parameters:

- *aName* – The variable name.
- *aInt* – The integer value.

Returns: None.

void setVar(String aName,int[] aInt) `[public]`

Description: Assign an integer array to a named variable.

Parameters:

- *aName* – The variable name.
- *aInt* – The integer array.

Returns: None.

void setVar(String aName,long aLong) `[public]`

Description: Assign a long integer value to a named variable.

Parameters:

- *aName* – The variable name.
- *aLong* – The long integer value.

Returns: None.

void setVar(String aName,long[] aLong) `[public]`

Description: Assign a long integer array to a named variable.

Parameters:

- *aName* – The variable name.
- *aLong* – The long integer array.

Returns: None.

void setVar(String aName,float aFloat) `[public]`

Description: Assign a float value to a named variable.

Parameters:

- *aName* – The variable name.
- *aFloat* – The float value.

Returns: None.

void setVar(String aName,float[] aFloat) `[public]`

Description: Assign a float array to a named variable.

Parameters:

- *aName* – The variable name.
- *aFloat* – The float array.

Returns: None.

void setVar(String aName,double aDouble) `[public]`

Description: Assign a double value to a named variable.

Parameters:

- *aName* – The variable name.
- *aDouble* – The double value.

Returns: None.

void setVar(String aName,double[] aDouble) `[public]`

Description: Assign a double array to a named variable.

Parameters:

- *aName* – The variable name.
- *aDouble* – The double array.

Returns: None.

void setVar(String aName,Object aObj) `[public]`

Description: Assign an Object to a named variable.

Parameters:

- *aName* – The variable name.
- *aObj* – The Object.

Returns: None.

void setVar(String aName,Object[] aObj) `[public]`

Description: Assign an Object array to a named variable.

Parameters:

- *aName* – The variable name.
- *aObj* – The Object array.

Returns: None.

Object getObject(String aName) `[public]`

Description: Retrieve the value of a named variable as an Object.

Parameters:

- *aName* – The variable name.

Returns: The value of the named variable.

Object[] getObjectArray(String aName) `[public]`

Description: Retrieve the value of a named variable as an Object array.

Parameters:

- *aName* – The variable name.

Returns: The named variable value array.

boolean getBoolean(String aName) `[public]`

Description: Retrieve the value of a named variable as a boolean.

Parameters:

- *aName* – The variable name.

Returns: The boolean value.

boolean[] getBooleanArray(String aName) `[public]`

Description: Retrieve the value of a named variable as a boolean array.

Parameters:

- *aName* – The variable name.

Returns: The boolean array of values.

byte getByte(String aName) `[public]`

Description: Retrieve the value of a named variable as a byte.

Parameters:

- *aName* – The variable name.

Returns: The byte value.

byte[] getByteArray(String aName) `[public]`

Description: Retrieve the value of a named variable as a byte array.

Parameters:

- *aName* – The variable name.

Returns: The byte array of values.

char getChar(String aName) `[public]`

Description: Retrieve the value of a named variable as a character.

Parameters:

- *aName* – The variable name.

Returns: The character value.

char[] getCharArray(String aName) `[public]`

Description: Retrieve the value of a named variable as a character array.

Parameters:

- *aName* – The variable name.

Returns: The character array of values.

short getShort(String aName) `[public]`

Description: Retrieve the value of a named variable as a short integer.

Parameters:

- *aName* – The variable name.

Returns: The short integer value.

short[] getShortArray(String aName) `[public]`

Description: Retrieve the value of a named variable as a short integer array.

Parameters:

- *aName* – The variable name.

Returns: The short integer array of values.

int getInt(String aName) `[public]`

Description: Retrieve the value of a named variable as an integer.

Parameters:

- *aName* – The variable name.

Returns: The integer value.

int[] getIntArray(String aName) `[public]`

Description: Retrieve the value of a named variable as an integer array.

Parameters:

- *aName* – The variable name.

Returns: The integer array of values.

long getLong(String aName) `[public]`

Description: Retrieve the value of a named variable as a long integer.

Parameters:

- *aName* – The variable name.

Returns: The long integer value.

long[] getLongArray(String aName) `[public]`

Description: Retrieve the value of a named variable as a long integer array.

Parameters:

- *aName* – The variable name.

Returns: The long integer array of values.

float getFloat(String aName) `[public]`

Description: Retrieve the value of a named variable as a float.

Parameters:

- *aName* – The variable name.

Returns: The float value.

float[] getFloatArray(String aName) `[public]`

Description: Retrieve the value of a named variable as a float array.

Parameters:

- *aName* – The variable name.

Returns: The float array of values.

double getDouble(String aName) `[public]`

Description: Retrieve the value of a named variable as a double.

Parameters:

- *aName* – The variable name.

Returns: The double value.

double[] getDoubleArray(String aName) `[public]`

Description: Retrieve the value of a named variable as a double array.

Parameters:

- *aName* – The variable name.

Returns: The double array of values.

boolean isVar(String aName) `[public, static]`

Description: Checks if a named variable exists.

Parameters:

- *aName* – The name of a variable to check.

Returns: True if the named variable exists.

A.13 ObjectReference

Description:	A proxy that refers to the a specific component by name.
Atrributes:	• `String componentname [private]` Name of the component referred to by this proxy.
Relationships:	Extends: Serializable

ObjectReference() `[protected]`

Description: Default constructor.

Returns: this

ObjectReference(String aName) `[public]`

Description: Constructor with the name of the component to refer to.

Returns: this

void setName(String aName) `[protected]`

Description: Set the name of the component to which this proxy refers.

Parameters:

- *aName* – The name of the component.

Returns: None.

String getName() `[public]`

Description: Get the name of component to which this proxy refers.

Returns: The name of the component to which this proxy refers.

Object invoke(String aName,Class[] argtyps,Object[] args) `[public]`

Description: Execute a component invocation.

Parameters:

- *aName* – The signature of the invocation to execute.
- *argtyps* – The invocation parameter types.
- *args* – The invocation parameter values.

Returns: None.

A.14 Parameters

Description:	Parameters associated with invocation scope.
Relationships:	Extends: Object

void setParameter(String aName,boolean aBool) `[public]`

Description: Set a named boolean parameter.

Parameters:

- *aName* – The parameter name.
- *aBool* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,boolean[] aBool) `[public]`

Description: Set a named boolean array parameter.

Parameters:

- *aName* – The parameter name.
- *aBool* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,boolean aByte) `[public]`

Description: Set a named byte parameter.

Parameters:

- *aName* – The parameter name.
- *aByte* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,byte[] aByte) `[public]`

Description: Set a named byte array parameter.

Parameters:

- *aName* – The parameter name.
- *aByte* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,boolean aChar) `[public]`

Description: Set a named character parameter.

Parameters:

- *aName* – The parameter name.
- *aChar* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,char[] aChar) `[public]`

Description: Set a named character array parameter.

Parameters:

- *aName* – The parameter name.
- *aChar* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,short aShort) `[public]`

Description: Set a named short integer parameter.

Parameters:

- *aName* – The parameter name.
- *aShort* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,short[] aShort) `[public]`

Description: Set a named short integer array parameter.

Parameters:

- *aName* – The parameter name.
- *aShort* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,int aInt) `[public]`

Description: Set a named integer parameter.

Parameters:

- *aName* – The parameter name.
- *aInt* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,int[] aInt) `[public]`

Description: Set a named integer array parameter.

Parameters:

- *aName* – The parameter name.
- *aInt* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,long aLong) `[public]`

Description: Set a named long integer parameter.

Parameters:

- *aName* – The parameter name.
- *aLong* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,long[] aLong) `[public]`

Description: Set a named long integer array parameter.

Parameters:

- *aName* – The parameter name.
- *aLong* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,float aFloat) `[public]`

Description: Set a named float parameter.

Parameters:

- *aName* – The parameter name.
- *aFloat* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,float[] aFloat) `[public]`

Description: Set a named float array parameter.

Parameters:

- *aName* – The parameter name.
- *aFloat* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,double aDouble) `[public]`

Description: Set a named double parameter.

Parameters:

- *aName* – The parameter name.
- *aDouble* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,double[] aDouble) `[public]`

Description: Set a named double array parameter.

Parameters:

- *aName* – The parameter name.
- *aDouble* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,Object aObj) `[public]`

Description: Set a named Object parameter.

Parameters:

- *aName* – The parameter name.
- *aObj* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,Object[] aObj) `[public]`

Description: Set a named Object array parameter.

Parameters:

- *aName* – The parameter name.
- *aObj* – The value to assign to the parameter.

Returns: None.

Object getParameter(String aName) `[public]`

Description: Get a named Object parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as an Object.

Object getParameter(int aIdx) `[public]`

Description: Get a named Object parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as an Object.

boolean getBoolean(String aName) `[public]`

Description: Get a named boolean parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a boolean.

boolean[] getBooleanArray(String aName) `[public]`

Description: Get a named boolean array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a boolean array.

Object getBoolean(int aIdx) `[public]`

Description: Get a named boolean parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a boolean.

boolean[] getBooleanArray(int aIdx) `[public]`

Description: Get a named boolean array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a boolean array.

byte getByte(String aName) `[public]`

Description: Get a named byte parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a byte.

byte[] getByteArray(String aName) `[public]`

Description: Get a named byte array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a byte array.

byte getByte(int aIdx) `[public]`

Description: Get a named byte parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a byte.

byte[] getByteArray(int aIdx) `[public]`

Description: Get a named byte array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a byte array.

char getChar(String aName) `[public]`

Description: Get a named char parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a character.

char[] getCharArray(String aName) `[public]`

Description: Get a named character array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a character array.

char getChar(int aIdx) `[public]`

Description: Get a named character parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a character.

char[] getCharArray(int aIdx) `[public]`

Description: Get a named character array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a character array.

short getShort(String aName) `[public]`

Description: Get a named short integer parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a short integer.

short[] getShortArray(String aName) `[public]`

Description: Get a named short integer array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a short integer array.

short getShort(int aIdx) `[public]`

Description: Get a named short integer parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a short integer.

short[] getShortArray(int aIdx) `[public]`

Description: Get a named short integer array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a short integer array.

int getInt(String aName) [public]

Description: Get a named integer parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as an integer.

int[] getIntArray(String aName) [public]

Description: Get a named integer array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a integer array.

int getInt(int aIdx) [public]

Description: Get a named integer parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as an integer.

int[] getIntArray(int aIdx) [public]

Description: Get a named integer array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as an integer array.

long getLong(String aName) [public]

Description: Get a named long integer parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a long integer.

long[] getLongArray(String aName) [public]

Description: Get a named long integer array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a long integer array.

long getLong(int aIdx) [public]

Description: Get a named long integer parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a long integer.

float getFloat(String aName) `[public]`

Description: Retrieve the value of a named variable as a float.

Parameters:

- *aName* – The variable name.

Returns: The float value.

float[] getFloatArray(String aName) `[public]`

Description: Retrieve the value of a named variable as a float array.

Parameters:

- *aName* – The variable name.

Returns: The float array of values.

double getDouble(String aName) `[public]`

Description: Retrieve the value of a named variable as a double.

Parameters:

- *aName* – The variable name.

Returns: The double value.

double[] getDoubleArray(String aName) `[public]`

Description: Retrieve the value of a named variable as a double array.

Parameters:

- *aName* – The variable name.

Returns: The double array of values.

boolean isVar(String aName) `[public, static]`

Description: Checks if a named variable exists.

Parameters:

- *aName* – The name of a variable to check.

Returns: True if the named variable exists.

A.13 ObjectReference

Description:	A proxy that refers to the a specific component by name.
Atrributes:	• `String componentname [private]` Name of the component referred to by this proxy.
Relationships:	Extends: Serializable

ObjectReference() `[protected]`

Description: Default constructor.

Returns: this

ObjectReference(String aName) `[public]`

Description: Constructor with the name of the component to refer to.

Returns: this

void setName(String aName) `[protected]`

Description: Set the name of the component to which this proxy refers.

Parameters:

- *aName* – The name of the component.

Returns: None.

String getName() `[public]`

Description: Get the name of component to which this proxy refers.

Returns: The name of the component to which this proxy refers.

Object invoke(String aName,Class[] argtyps,Object[] args) `[public]`

Description: Execute a component invocation.

Parameters:

- *aName* – The signature of the invocation to execute.
- *argtyps* – The invocation parameter types.
- *args* – The invocation parameter values.

Returns: None.

A.14 Parameters

Description:	Parameters associated with invocation scope.
Relationships:	Extends: Object

void setParameter(String aName,boolean aBool) `[public]`

Description: Set a named boolean parameter.

Parameters:

- *aName* – The parameter name.
- *aBool* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,boolean[] aBool) `[public]`

Description: Set a named boolean array parameter.

Parameters:

- *aName* – The parameter name.
- *aBool* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,boolean aByte) [public]

Description: Set a named byte parameter.

Parameters:

- *aName* – The parameter name.
- *aByte* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,byte[] aByte) [public]

Description: Set a named byte array parameter.

Parameters:

- *aName* – The parameter name.
- *aByte* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,boolean aChar) [public]

Description: Set a named character parameter.

Parameters:

- *aName* – The parameter name.
- *aChar* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,char[] aChar) [public]

Description: Set a named character array parameter.

Parameters:

- *aName* – The parameter name.
- *aChar* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,short aShort) [public]

Description: Set a named short integer parameter.

Parameters:

- *aName* – The parameter name.
- *aShort* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,short[] aShort) [public]

Description: Set a named short integer array parameter.

Parameters:

- *aName* – The parameter name.
- *aShort* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,int aInt) `[public]`

Description: Set a named integer parameter.

Parameters:

- *aName* – The parameter name.
- *aInt* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,int[] aInt) `[public]`

Description: Set a named integer array parameter.

Parameters:

- *aName* – The parameter name.
- *aInt* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,long aLong) `[public]`

Description: Set a named long integer parameter.

Parameters:

- *aName* – The parameter name.
- *aLong* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,long[] aLong) `[public]`

Description: Set a named long integer array parameter.

Parameters:

- *aName* – The parameter name.
- *aLong* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,float aFloat) `[public]`

Description: Set a named float parameter.

Parameters:

- *aName* – The parameter name.
- *aFloat* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,float[] aFloat) `[public]`

Description: Set a named float array parameter.

Parameters:

- *aName* – The parameter name.
- *aFloat* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,double aDouble) `[public]`

Description: Set a named double parameter.

Parameters:

- *aName* – The parameter name.
- *aDouble* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,double[] aDouble) `[public]`

Description: Set a named double array parameter.

Parameters:

- *aName* – The parameter name.
- *aDouble* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,Object aObj) `[public]`

Description: Set a named Object parameter.

Parameters:

- *aName* – The parameter name.
- *aObj* – The value to assign to the parameter.

Returns: None.

void setParameter(String aName,Object[] aObj) `[public]`

Description: Set a named Object array parameter.

Parameters:

- *aName* – The parameter name.
- *aObj* – The value to assign to the parameter.

Returns: None.

Object getParameter(String aName) `[public]`

Description: Get a named Object parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as an Object.

Object getParameter(int aIdx) `[public]`

Description: Get a named Object parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as an Object.

boolean getBoolean(String aName) `[public]`

Description: Get a named boolean parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a boolean.

boolean[] getBooleanArray(String aName) `[public]`

Description: Get a named boolean array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a boolean array.

Object getBoolean(int aIdx) `[public]`

Description: Get a named boolean parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a boolean.

boolean[] getBooleanArray(int aIdx) `[public]`

Description: Get a named boolean array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a boolean array.

byte getByte(String aName) `[public]`

Description: Get a named byte parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a byte.

byte[] getByteArray(String aName) `[public]`

Description: Get a named byte array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a byte array.

byte getByte(int aIdx) `[public]`

Description: Get a named byte parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a byte.

byte[] getByteArray(int aIdx) `[public]`

Description: Get a named byte array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a byte array.

char getChar(String aName) `[public]`

Description: Get a named char parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a character.

char[] getCharArray(String aName) `[public]`

Description: Get a named character array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a character array.

char getChar(int aIdx) `[public]`

Description: Get a named character parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a character.

char[] getCharArray(int aIdx) `[public]`

Description: Get a named character array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a character array.

short getShort(String aName) `[public]`

Description: Get a named short integer parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a short integer.

short[] getShortArray(String aName) `[public]`

Description: Get a named short integer array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a short integer array.

short getShort(int aIdx) `[public]`

Description: Get a named short integer parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a short integer.

short[] getShortArray(int aIdx) `[public]`

Description: Get a named short integer array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a short integer array.

int getInt(String aName) `[public]`

Description: Get a named integer parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as an integer.

int[] getIntArray(String aName) `[public]`

Description: Get a named integer array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a integer array.

int getInt(int aIdx) `[public]`

Description: Get a named integer parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as an integer.

int[] getIntArray(int aIdx) `[public]`

Description: Get a named integer array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as an integer array.

long getLong(String aName) `[public]`

Description: Get a named long integer parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a long integer.

long[] getLongArray(String aName) `[public]`

Description: Get a named long integer array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a long integer array.

long getLong(int aIdx) `[public]`

Description: Get a named long integer parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a long integer.

long[] getLongArray(int aIdx) `[public]`

Description: Get a named long integer array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a long integer array.

float getFloat(String aName) `[public]`

Description: Get a named float parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a float.

float[] getFloatArray(String aName) `[public]`

Description: Get a named float array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a float array.

float getFloat(int aIdx) `[public]`

Description: Get a named float parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a float.

float[] getFloatArray(int aIdx) `[public]`

Description: Get a named float array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a float array.

double getDouble(String aName) `[public]`

Description: Get a named double parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a double.

double[] getDoubleArray(String aName) `[public]`

Description: Get a named double array parameter by name.

Parameters:

- *aName* – The parameter name.

Returns: The named parameter as a double array.

double getDouble(int aIdx) `[public]`

Description: Get a named double parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a double.

double[] getDoubleArray(int aIdx) `[public]`

Description: Get a named double array parameter at an index.

Parameters:

- *aIdx* – The parameter index.

Returns: The parameter as a double array.

boolean isParameter(String aName) `[public, static]`

Description: Checks if a named parameter exists.

Parameters:

- *aName* – The name of a parameter to check.

Returns: True if the named parameter exists.

A.15 ReturnCode

Description:	Invocation return code. Encapsulates the value returned by an invocation with type information.
Atrributes:	• `short rctype [public]` Return code type. • `Object retcode [public]` Return code value.
Relationships:	Extends: Object

A.16 Scope

Description:	Invocation execution scope data. This data is available to each factor executed as part of an invocation execution.
Atrributes:	• `ObjectReference self_component [public]` Proxy reference to the component using this scope. • `ReturnCode return_code [public]` The invocation return code associated with this scope. • `Parameters parameters [public]` The parameters passed to this invocation. • `FactorsetVars variables [public]` The factorset variables associated with this invocation.
Relationships:	Extends: Object

A.17 ScopeManager

Description:	This class enables modification of the scope store.
Relationships:	Extends: Object

ScopeStore() [protected]

Description: Default constructor.

Returns: this

void push(Scope aScope) [public, static]

Description: Push a scope on the scope stack (ScopeStore).

Parameters:

- *aScope* – The scope to push on the scope stack.

Returns: None.

Scope pop() [public, static]

Description: Pop a scope off the top of the scope stack (ScopeStore).

Returns: The scope from the top of the scope stack or null if the stack is empty.

Scope peek() [public, static]

Description: Peek at the scope on the top of the scope stack (ScopeStore) without removing the scope from the stack.

Returns: The scope on the top of the scope stack or null if the stack is empty.

A.18 ScopeStore

Description:	A repository of Perimorph execute scopes. When a interface method is invoked, an execution scope is pushed onto the execution stack that can be retrieved by the factors during execution. Each thread has its own execution stack.
Atrributes:	• `HashMap scopemap [private]` Collection that stores scope stacks on a per thread basis.
Relationships:	Extends: Object

void push(Scope aScope) [public, static]

Description: Push a scope on the scope stack. A different scope stack is maintained for each execution thread.

Parameters:

- *aScope* – The scope to push on the scope stack.

Returns: None.

Scope pop() `[public, static]`

Description: Pop a scope off the top of the current thread's scope stack.

Returns: The scope from the top of the scope stack or null if the stack is empty.

Scope peek() `[public, static]`

Description: Peek at the scope on the top of the current thread's scope stack without removing the scope from the stack.

Returns: The scope on the top of the scope stack or null if the stack is empty.

Appendix B

Dynamic River Operators and Support Programs

In distributed, data-streaming systems, adaptation is possible if operators can be introduced into the data stream transparently or gracefully redeployed to different hosts. This appendix briefly describes the usage of the Dynamic River support programs and pipeline operators referred to in this dissertation. Dynamic River, an extension to the DaSH data acquisition system [137], comprises operators designed for processing sensor data streams and enables sets of operators to be dynamically relocated to more suitable hosts to better meet quality-of-service requirements. The programs described here represent an incomplete list; many other operators are available. Currently, Dynamic River provides 60 operators for processing or routing data and more operators are being implemented on a regular basis to meet the needs of different applications.

B.1 Basic Pipeline Operators

B.1.1 Asciionramp

Enable the injection of ASCII data into the data stream. The `asciionramp` operator enables ASCII output from USERCOMMAND (scripts or other programs) to be easily

injected into the data stream. The output of USERCOMMAND is automatically packaged into records prior to being injected into the data stream. The essential characteristics of **asciionramp** are as follows:

- Records appearing on **asciionramp**'s **stdin** are transferred to **stdout**.
- Once a BEGINRUN record has been read from **stdin**, the BEGINRUN record is written to **stdout** and USERCOMMAND executed. USERCOMMAND's **stdout** is connected to a pipe, enabling **asciionramp** to read data produced by USERCOMMAND. USERCOMMAND's **stderr** is also connected to a pipe, enabling **asciionramp** to read error messages produced by USERCOMMAND and relay them to **asciionramp**'s **stderr**.
- Data produced by USERCOMMAND is automatically packaged into records and written to **asciionramp**'s **stdout**.
- If USERCOMMAND exits and --restart is not specified, **asciionramp** continues reading records from **stdin** and writing them to **stdout**.
- If **asciionramp**'s **stdin** is closed, **asciionramp** kills USERCOMMAND (if necessary) and then exits.

Synopsis:

```
Usage:  asciionramp --packet=PACKET [OPTION]...  USERCOMMAND
```

Options:

- **--version** Output version information and exit
- **--help** Display this help and exit
- **--oneliner** Produce a new record for each line of output produced by USERCOMMAND and write it to **stdout**.
- **--separator=CHARACTER** Collect output from USERCOMMAND until CHARACTER is received. The collected output, excluding the separator, is packaged into a single record and written to **stdout**.
- **--restart[=RETRIES]** (default: infinite restarts) Restart USERBINARY each time it exits. If RETRIES is specified, only restart the program RETRIES times.
- **--packet=PACKET** Mandatory. PACKET can be either a single positive integer or a symbolic name (e.g., Physics). Records produced from the output of USERBINARY will be assigned a type of PACKET.

Examples:

- `readout --run=45 | asciionramp --packet=104 --oneliner timestamp.sh | segmenter /evt/data/output`
  Inject a time stamp into the data stream. Each line output by timestamp.sh is packaged into a record and written to `stdout`.

- `readout --run=46 | asciionramp --packet=103 --restart periodic.sh | segmenter /evt/data/output`
  Inject periodically produced data into the data stream. The entire output of periodic.sh is packaged as a single record and written to `stdout`. When periodic.sh terminates, it is restarted.

- `readout --run=47 | asciionramp --packet=105 --separator=''\f'' --restart thresholds.sh | segmenter /evt/data/output`
  Collect output from thresholds.sh until a form feed character is received. The collected output, excluding the form feed character, is packaged into a single record and written to `stdout`. When thresholds.sh terminates, it is restarted.

B.1.2 Binary2record

Convert binary data directly into DaSH records. This operator is not intended as an injection operator, but as a primary source that can consume binary data from `stdin` and write DaSH records on `stdout`. Data read from `stdin` is automatically packaged into records prior to being written to `stdout`. The essential characteristics of `binary2record` are as follows:

- Binary2record first emits a BEGINRUN record on `stdout`.

- Data read from `stdin` is expected to have a structure that includes a leading byte count, indicating the number of data bytes produced, followed by the data. The leading byte count should be a 4 byte integer in host byte order.

- Data read from `stdin` is automatically packaged into records and written to `binary2record`'s `stdout`.

- If `binary2record`'s `stdin` is closed, or `binary2record` receives a SIGINT, `binary2record` will emit a ENDRUN record on `stdout` and then exit.

Synopsis:

`Usage: binary2record --run=INTEGER --packet=INTEGER [OPTION]...`

Options:

- `--version` Output version information and exit

- `--help` Display this help and exit

- `--run=INTEGER` Mandatory. Specify an experiment run number

- `--packet=[PACKET|typed]` Mandatory. Record type to use for records produced. PACKET may be an integer or the symbolic name for a well known packet type such as Physics. If the argument is the string "typed," then the binary records read are assumed to have an host-byte-order integer field, indicating the record type, following the length.

- `--title=STRING` Experiment title string.

Examples:

- `binary2record --run=45 --packet=104 --title=''Experiment number 45''`
  Produce records using run number 45 and title "Experiment number 45." Records produced from binary data read from **stdin** will have a record type of 104.

B.1.3 Binaryonramp

Enable the injection of binary data into the data stream. The **binaryonramp** operator enables binary output from USERBINARY (executable programs) to be easily injected into the data stream. The output of USERBINARY is automatically packaged into records prior to being injected into the data stream. The essential characteristics of **binaryonramp** are as follows:

- Records appearing on **binaryonramp**'s **stdin** are transferred to **stdout**.

- Once a BEGINRUN record has been read from **stdin**, the BEGINRUN record is written to **stdout** and USERBINARY executed. USERBINARY's **stdout** is connected to a pipe, enabling **binaryonramp** to read data produced by USERBINARY. USERBINARY's **stderr** is also connected to a pipe, enabling **binaryonramp** to read error messages produced by USERBINARY and relay them to **binaryonramp**'s **stderr**.

- Data produced by USERBINARY is expected to produce output that includes a leading byte count, indicating the number of data bytes produced, followed by the data. The leading byte count should be a 4 byte integer in host byte order.

- Data produced by USERBINARY is automatically packaged into records and written to **binaryonramp**'s **stdout**.

- If USERBINARY exits and --restart is not specified, **binaryonramp** continues reading records from **stdin** and writing them to **stdout**.

- If **binaryonramp**'s **stdin** is closed, **binaryonramp** kills USERBINARY (if necessary) and then exits.

Synopsis:

```
Usage:  binaryonramp --packet=PACKET [OPTION]...  USERBINARY
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--restart[=RETRIES]` (default: infinite restarts) Restart USERBINARY each time it exits. If RETRIES is specified, only restart the program RETRIES times.
- `--packet=[PACKET|typed]` Mandatory. PACKET can be either a single positive integer or a symbolic name (e.g., Physics). Records produced from the output of USERBINARY will be assigned a type of PACKET. If the argument is the string "typed," then the output of USERBINARY is assumed to have an host-byte-order integer field, indicating the record type, following the length.

Examples:

- `readout --run=45 | binaryonramp --packet=104 mybinary | segmenter /evt/data/output`
  Inject output from `mybinary` into the data stream. Records produced from data emitted by `mybinary` will have a type of 104. If `mybinary` terminates, it will not be restarted.
- `readout --run=46 | binaryonramp --packet=103 --restart mybinary | segmenter /evt/data/output`
  Inject output from `mybinary` into the data stream. Records produced from data emitted by `mybinary` will have a type of 103. When `mybinary` terminates, it is restarted.

B.1.4 Cabsf

Read records that contain an array of complex float values and calculate the complex absolute value for each element. Input format is (r,i),(r,i),(r,i), . . .

Synopsis:

```
Usage:  cabsf [OPTION]...
```

Options:

- `--version` Output version information and exit.
- `--help` Display this help and exit.

- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as DATA. Default type is DATA.
- `--pass` Pass all input records through to output without modification. This option only applies to records that are processed by this segment as indicated by `--packet`. Records that are not processed are automatically passed through to output without modification.
- `--emit=PACKET` Records that are created by this operator are emitted with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as DATA. Default type is DATA.
- `--half` Only output the first half of the data in the output array (with FFT data, the second half is a mirror image of the first half and is not used for plotting spectrograms).

Examples:

- `cat audio.dat | wav2rec --samples=768 | float2cplx | dft | cabsf > cabsf.recs`
  Read the `sox .dat` file from `stdin` and pipe it through the discrete Fourier transform (`dft`) and then into `cabsf` to create a file of power spectrum records.

B.1.5 Cutout

Read records that contain an array of floating point values and reformat them such that emitted records contain only a specified range of values.

Synopsis:

`Usage: cutout --range=INTEGER,INTEGER [OPTION]...`

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--range=INTEGER,INTEGER` Mandatory. Specify the accepted range as array indices greater than or equal to zero.

Examples:

- `cat audio.wav | wav2rec --samples=768 | cutout --range=0,99 > cutout.recs`
  Use `cutout` to create a file of data where only the first 100 values are retained for each original record.

B.1.6 Cutter

Read records that contain an array of time series float values and a trigger signal. Cut the time series into pieces based on when the trigger signal exceeds a threshold value.

Synopsis:

```
Usage:  cutter [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--pass` Pass all input records through to output without modification. The option only applies to records that are processed by this segment as indicated by `--packet` or `--signal`. Records that are not processed are automatically passed through to output without modification.
- `--emit=PACKET` Records that are created by this process are emitted with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--signal=PACKET` Records with the type of PACKET are used as the signal for cutting data records (`--packet`). PACKET may be an integer or the symbolic name for a well known packet type such as TRIGGER. Default type is TRIGGER.
- `--threshold=REAL` Set the signal threshold value for cutting data records. Default is 1.0.
- `--all` Cut sections for both high and low triggers (above and below/equal the threshold). Default is to only cut sections for high triggers.

Examples:

- `cat audio.wav | wav2rec --samples=768 | trigger --pass --emit=trigger | cutter --signal=trigger --threshold=0.5 > cutter.recs`
  Pipe audio.wav through `trigger` to add trigger records to the data stream that can be used to cut data records into pieces. Then use `cutter` to cut the data stream based on the trigger signal.

B.1.7 Daqcat

Output the records from an experiment that has previously been saved to disk. Experimental data is stored in segmented files to avoid exceeding the maximum file size of the filesystem. To simplify processing experimental data, `daqcat` understands how to output the records for a particular run starting at the first segment and terminating at the last segment.

Synopsis:

```
Usage: daqcat --run=INTEGER [OPTION]... DIRECTORY
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--run=INTEGER` Mandatory. Specify an experiment run number

Examples:

- `daqcat --run=45 /evt/data/output`
  Cat experiment with run number 45 found in directory `/evt/data/output`
- `daqcat --run=123 /evt/data/output | recorddump` Cat experiment with run number 123 found in directory `/evt/data/output` into operator `recorddump`.

B.1.8 Daqtail

Output the records from an experiment that has previously been saved to disk. Experimental data is stored in segmented files to avoid exceeding the maximum file size of the filesystem. To simplify processing experimental data, `daqtail` understands how to output the records for the current experiment as the are appended to the current segment.

Creation of new runs and new run segments in the specified directory is understood by `daqtail`. When new runs are added to the directory, `daqtail` will detect that a new run has been started and automatically resume processing with this new run. When new segments are created for the current run, `daqtail` will detect that a new segment has been created and continue processing with this new segment.

Unless the --catchup option has been specified, processing begins by outputting the last record of the most recent segment.

Synopsis:

```
Usage: daqtail [OPTION]... DIRECTORY
```

Options:

- `--version` Output version information and exit

- `--help` Display this help and exit
- `--catchup` Start processing with the first record of the first segment of the current run.
- `--collate=[ctime|runnumber]` (default: ctime) Collate the event files found in DIRECTORY according to either file ctime (inode change time) or by run-number.
- `--file` Treat DIRECTORY as a file. The file specified will be tailed until an ENDRUN, BADEND or CONTINUE record is encountered. Note that --collate has no effect when tailing a specific file. If --catchup is specified, processing begins at the start, instead of the end, of the specified file.

Examples:

- `daqtail /evt/data/output`
  Tail most recent experiment found in directory /evt/data/output
- `daqtail --catchup /evt/data/output | recorddump`
  Tail most recent experiment found in directory /evt/data/output into operator `recorddump`. Processing begins with the first segment of the most recent run.
- `daqtail --collate=runnumber /evt/data/output`
  Tail the most recent experiment found in directory /evt/data/output. Collate using run numbers instead of ctime.
- `daqtail --file experiment.data`
  Tail the file `experiment.data` and exit when an ENDRUN, BADEND or CONTINUE record is encountered.

B.1.9 Daqtee

`Daqtee` writes records from an experiment to two separate operators enabling online filtering and data processing while retaining the unfiltered or processed data. Records written to the secondary (tee) data stream can be selected a priori using the --packet flag. That is, `daqtee` considers each record read from `stdin`. If the record's packet type matches one of the specified packet types, then the record will be written to the tee data stream. All records read from `stdin` are unconditionally written to the main data stream on `stdout`. If the --packet flag is not specified, then all records read from stdin are also written to the tee data stream.

Packet types can be specified as either a single positive integer, a range of positive integers or by using one of the known symbolic names for some well known packet types. Currently, the list of symbolic well known packet types includes: Physics (a.k.a, Data), Scaler, SnapSc, StateVar, RunVar, PktDoc, ParamDescript and Trigger.

The essential characteristics of `daqtee` are as follows:

- `Daqtee` passes all records read on `stdin` to `stdout`.

- **daqtee** runs a client program with **stdin** connected to a pipe to which it write a copy of the records specified using the --packet flag. If, the --packet flag is not specified then a copy of all the records received on **stdin** will be written to the pipe.
- If PROGRAM breaks the pipe, **daqtee** degenerates to writing all records read on **stdin** to **stdout**.
- If an EOF is received on **stdin**, **daqtee** exits.

Synopsis:

```
Usage:  daqtee [--packet=PACKETS] ...  [OPTION]...  PROGRAM
```

Options:

- **--version** Output version information and exit
- **--help** Display this help and exit
- **--packet=PACKETS** If more than one occurrence of this flag appears on the command line, all packets and ranges specified will be considered. PACKETS can be either a single positive integer, a range of positive integers (e.g., 101-118 or 101,118) or a symbolic name (e.g., Physics). All ranges are inclusive.

Examples:

- **readout --run=123 | daqtee --packet=101 myfilter | segmenter /evt/data/output**
  Output all records read on **stdin** to **stdout** and write a copy of records that have a packet type of 101 to **myfilter**.
- **readout --run=890 | daqtee --packet=PktDoc recorddump | segmenter /evt/data/output**
  Output all records read on **stdin** to **stdout** and write a copy of records that have a packet type of PktDoc to **recorddump**.

B.1.10 Dft

Read records containing an array of complex values, in (real,imaginary) format, and compute the discrete Fourier transform for each record.

Synopsis:

```
Usage:  dft [OPTION]...
```

Options:

- **--version** Output version information and exit
- **--help** Display this help and exit

- `--pow2` Use a Fourier transform optimized for arrays of length power of 2.
- `--invert` Execute an inverse Fourier transform instead of a forward transform.
- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--pass` Pass all input records through to output without modification. The option only applies to records that are processed by this segment as indicated by `--packet`. Records that are not processed are automatically passed through to output without modification.
- `--emit=PACKET` Records that are created by this process are emitted with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.

Examples:

- `cat audio.wav | wav2rec --samples=768 | float2cplx --pow2 | dft --pow2 >fft.recs`
  Use dft to create a file of Fourier transformation records from an audio file in WAV format.

B.1.11 Feed

Read DaSH records from storage and write them to `stdout`. As discussed in this dissertation, this operator is abstract. See `daqcat` or `daqtail` for a concrete example of a `feed` operator.

B.1.12 Float2cplx

Read records that contain an array of float values and reformat the array as an array of complex numbers. The output ordering is (real,imaginary),(real,imaginary),(real,imaginary),...,(real,imaginary) with the imaginary part set to 0.0.

Synopsis:

```
Usage: float2cplx [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--pow2` Structure complex data as an array of length power of 2 padded with zeros.

- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.

- `--pass` Pass all input records through to output without modification. The option only applies to record that are processed by this segment as indicated by `--packet`. Records that are not processed are automatically passed through to output without modification.

- `--emit=PACKET` Records that are created by this process are emitted with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.

Examples:

- `cat audio.wav | wav2rec --samples=768 | float2cplx --pow2 | dft --pow2 >fft.recs`
  Use float2cplx to create a record stream of audio data suitable for processing by `dft`.

B.1.13 Paa

Read records that contain an array of time-series, float values and compute the piecewise aggregate approximation (PAA) for each data record.

Synopsis:

`Usage: paa [OPTION]...`

Options:

- `--version` Output version information and exit

- `--help` Display this help and exit

- `--pass` Pass all input records through to output without modification. The option only applies to records that are processed by this segment as indicated by `--packet`. Records that are not processed are automatically passed through to output without modification.

- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.

- `--emit=PACKET` Records that are created by this process are emitted with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.

- `--segments=INTEGER` Set the number of segments to use for creating a PAA (Piecewise Aggregate Approximation) of the time series.

Examples:

- `cat audio.wav | wav2rec --samples=768 | paa > paa.recs`
  Use paa to compute the piecewise aggregate approximation of audio WAV data.

B.1.14 Ratemeter

Reads a DaSH record stream from **stdin** and either consume the record or pass it through to **stdout**. Calculate rate statistics and print them to **stderr**. The size of the record header is not included when calculating statistics.

Synopsis:

```
Usage: ratemeter [--consume] [--every=INTEGER]... [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--every=SECONDS` An attempt will be made to output statistics to **stderr** every INTEGER seconds of wall clock time. That is, statistics will output after the specified number of seconds have passed and a new record has been read.
- `--consume` Records will be consumed rather written to **stdout**.

Examples:

- **readout --run=123 | ratemeter --every=5 | recorddump**
  Calculate statistics for run 123, outputting statistics on **stderr** every 5 seconds. Records are then written to **stdout** for consumption by **recorddump**.
- **readout --run=890 | ratemeter --consume --every=2048**
  Calculate statistics for run 890, outputting statistics on **stderr** every 10 seconds. Records are then consumed.

B.1.15 Raw2record

Convert raw data directly into DaSH records. This operator is not intended as an injection operator, but as a primary source that can consume raw data from **stdin** and write DaSH records on **stdout**. Data read from **stdin** is automatically packaged into records prior to being written to **stdout**. The essential characteristics of **raw2record** are as follows:

- Raw2record first emits a BEGINRUN record on stdout.
- Data read from **stdin** is treated as a stream of bytes.

- Data read from `stdin` is automatically packaged into records and written to `raw2record`'s `stdout`.
- If `raw2record`'s `stdin` is closed, or `raw2record` receives a SIGINT, `raw2record` will emit a ENDRUN record on `stdout` and then exit.

Synopsis:

```
Usage: raw2record --run=INTEGER --packet=INTEGER [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--run=INTEGER` Mandatory. Specify an experiment run number
- `--packet=INTEGER` Mandatory. Record type to use for records produced.
- `--title=STRING` Experiment title string
- `--size=INTEGER` Size of data in bytes to include in each record. Default 8192 bytes.

Examples:

- `raw2record --run=45 --packet=104 --title=''Experiment number 45''` Produce records using run number 45 and title "Experiment number 45." Records produced from raw data read from `stdin` will have a record type of 104.
- `raw2record --size=1024 --run=51 --packet=Data --title=''Experiment number 51''` Produce records using run number 51 and title "Experiment number 51." Records produced from raw data read from `stdin` will have a record type of DATA. Each record will contain 1024 bytes of raw data.

B.1.16 Readout

Read experiment input from data acquisition hardware and output event records. Each execution of `readout` begins with a BEGINRUN record and ends with either an ENDRUN or BADEND record depending on how `readout` was terminated. If `readout` is terminated with a SIGINT (Ctrl-C), then `readout` emits an ENDRUN record. If `readout` is terminated by a SIGTERM, then `readout` will emit a BADEND record prior to exiting. All other records are of type DATA.

Synopsis:

```
Usage: readout --run=INTEGER [OPTION]...
```

Options:

- `--version` Output version information and exit.
- `--help` Display this help and exit.
- `--run=INTEGER` Mandatory. Specify an experiment run number
- `--title=STRING` Experiment title string

Examples:

- `readout --run=45 --title=''Experiment number 45''`
  Produce records using run number 45 and title "Experiment number 45."
- `readout --run=123 --title=''A cool experiment'' | recorddump`
  Produce records using run number 123 and title "A cool experiment" and pipe them into operator **recorddump**.

B.1.17 Record

Read DaSH records from **stdin** and write them to storage. As discussed in this dissertation, this operator is abstract. See **segmenter** for a concrete example of a **record** operator.

B.1.18 Record2binary

Convert DaSH records directly into binary data. Binary data is data stream consisting of records with a leading length field, written as an integer in host byte order, followed by data bytes. Control records, such as BEGINRUN, ENDRUN, BADEND and CONTINUE, are consumed and not passed to **stdout** as part of the raw data stream.

Synopsis:

`Usage: record2binary [OPTION]...`

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--typed` Output typed binary data. That is, add a host-byte-order integer field, following the length field, that indicates the record type.

Examples:

- `daqcat --run=45 /evt/data/output | record2binary`
  Produce a binary data stream from the event data for run 45 stored in directory /evt/data/output. Conversion continues until `record2binary` receives an EOF on `stdin`.

- `daqcat --run=45 /evt/data/output | sieve --packet=DATA | record2binary`
  Produce a binary data stream from the event data for run 45 stored in directory /evt/data/output. **Sieve** restricts the type of records fed to **record2binary** to be of type DATA. Conversion continues until **record2binary** receives an EOF on **stdin**.

B.1.19 Record2raw

Convert DaSH records directory into raw data. Raw data is simply a stream of data bytes about which no structure is assumed. Control records, such as BEGINRUN, ENDRUN, BADEND and CONTINUE, are consumed and not passed to **stdout** as part of the raw data stream.

Synopsis:

`Usage: record2raw [OPTION]...`

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit

Examples:

- `daqcat --run=45 /evt/data/output | record2raw`
  Produce a raw data stream from the event data for run 45 stored in directory /evt/data/output. Conversion continues until **record2raw** receives an EOF on **stdin**.
- `daqcat --run=45 /evt/data/output | sieve --packet=DATA | record2raw`
  Produce a raw data stream from the event data for run 45 stored in directory /evt/data/output. **Sieve** restricts the type of records fed to **record2raw** to be of type DATA. Conversion continues until **record2raw** receives an EOF on **stdin**.

B.1.20 Record2vect

Read records comprising host format float arrays from stdin and write out vectors (one vector per line) on stdout.

Synopsis:

`Usage: record2vect [OPTION]...`

Options:

- `--version` Output version information and exit

- `--help` Display this help and exit
- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--merge=INTEGER` Merge multiple records into one vector. INTEGER indicates the number of records to merge. Default is 1 (each record forms one vector).

Examples:

- `cat audio.wav | wav2rec --samples=768 | record2vect > audio.data` Use `record2vect` to convert audio data into vectors suitable for processing by MESO.

B.1.21 Recorddump

Reads a DaSH record stream from `stdin` and outputs the record headers in human readable format.

Synopsis:

```
Usage: recorddump [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--dump=NBYTES` Dump up to NBYTES of record data in hex format
- `--print` Also dump data as printable characters if possible.
- `daqtail --run=123 /evt/data/output | recorddump` Use daqcat to output the records from run 123 found in directory /evt/data/output and pipe them into recorddump.
- `cat /evt/data/output/run0123_0001.evt | recorddump` Cat the first data file segment from run 123 into recorddump.

B.1.22 Reslice

Read records that contain an array of float values and reformat each record so that every two original records is separated by a record containing half of the first original record and half of the second original record.

Synopsis:

```
Usage: reslice [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.

Examples:

- `cat audio.wav | wav2rec --samples=768 | reslice > resliced.recs`
  Pipe records of audio WAV data through reslicef to create a file where each original record is separated by a record containing half of the preceding record and half of the following record.

B.1.23 Sample

Selectively sample records from an experiment that have the packet types specified using the --packet flag. That is, `sample` considers each record read from `stdin`. If the record's packet type matches one of the specified packet types, then the record will only be written to `stdout` if writing to `stdout` will not block. All other record types are unconditionally written to `stdout` even if the write required to do so would block.

Packet types can be specified as either a single positive integer, a range of positive integers or by using one of the known symbolic names for some well known packet types. Currently, the list of symbolic well known packet types includes: Physics (a.k.a, Data), Scaler, SnapSc, StateVar, RunVar, PktDoc, ParamDescript and Trigger.

Synopsis:

`Usage: sample --packet=PACKETS [--packet=PACKETS]... [OPTION]...`

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--packet=PACKETS` Mandatory. If more than one occurrence of this flag appears on the command line, all packets and ranges specified will be considered. PACKETS can be either a single positive integer, a range of positive integers (e.g., 101-118 or 101,118) or a symbolic name (e.g., Physics). All ranges are inclusive.

Examples:

- `readout --run=123 | sample --packet=101`
  Output those records that have packet type 101 only if writing to `stdout` will not block. All other records are written regardless of blocking.

- `readout --run=890 | sample --packet=105-116 | recorddump`
  Pipe those records that have have a packet type falling in the range from 105 to 116 (inclusive) into the operator `recorddump` only if writing to `stdout` will not block. All other records are written regardless of blocking.

- `readout --run=567 | sample --packet=DATA --packet=150 | recorddump`
  Pipe those records that have have a packet type corresponding to the symbolic type of DATA and those records with packet type 150 into the operator `recorddump` only if writing to `stdout` will not block. All other records are written regardless of blocking.

B.1.24 Saxanomaly

Read records that contain an array of time series, floating point values and compute the symbolic aggregate approximation (SAX) anomaly score.

Synopsis:

`Usage: saxanomally [OPTION]...`

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--pass` Pass all input records through to output without modification. The option only applies to records that are processed by this segment as indicated by `--packet`. Records that are not processed are automatically passed through to output without modification.
- `--emit=PACKET` Records that are created by this process are emitted with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--average=INTEGER` Output anomaly score as a moving average over the specified window size.
- `--samples=INTEGER` The number of samples to encode per output record. Default is 500.
- `--subseqsize=INTEGER` The length of subsequences to use for constructing the SAX bitmaps. Default is 2.

- `--window=INTEGER` Set the sliding window size for selecting a maximum trigger value. Default is 128.
- `--alphabet=INTEGER` Set the SAX alphabet size. Default is 8.
- `--segsize=INTEGER` Set the number of values to use for creating a piecewise aggregate approximation (PAA) of the time series. This representation will then be converted to SAX format. Default is 10.
- `--dist=FLOAT1,FLOAT2` Specify a precomputed mean and standard deviation to use when Z-normalizing rather than compute the mean and standard deviation for each vector. FLOAT1 and FLOAT2 are the specified mean and standard deviation, respectively.

Examples:

- `cat audio.wav | wav2rec --samples=768 | saxbitmap > saxbitmap.recs`
  Use `saxanomaly` to create a file containing records comprising SAX anomaly scores.

B.1.25 Saxbitmap

Read records that contain an array of time series, floating point values and compute the symbolic aggregate approximation (SAX) bitmap representation.

Synopsis:

`Usage: saxbitmap [OPTION]...`

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--pass` Pass all input records through to output without modification. The option only applies to records that are processed by this operator as indicated by `--packet`. Records that are not processed are automatically passed through to output without modification.
- `--emit=PACKET` Records that are created by this process are emitted with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.

- `--subseqsize=INTEGER` The length of subsequences to use for constructing the SAX bitmaps. Default is 2.
- `--alphabet=INTEGER` Set the SAX alphabet size. Default is 8.
- `--segsize=INTEGER` Set the number of values to use for creating a piecewise aggregate approximation (PAA) of the time series. This representation will then be converted to SAX format. Default is 10.
- `--rate` Instead of outputting a bitmap containing counts of subsequences, output the rate at which subsequences occur. That is, divide the count in each cell by the total count of subsequences.
- `--dist=FLOAT1,FLOAT2` Specify a precomputed mean and standard deviation to use when Z-normalizing rather than compute the mean and standard deviation for each vector. FLOAT1 and FLOAT2 are the specified mean and standard deviation respectively.

Examples:

- `cat audio.wav | wav2rec --samples=768 | saxbitmap > saxbitmap.recs`
  Use `saxbitmap` to create a file containing records comprising a SAX bitmap representation of the original records.

B.1.26 Segmenter

Divide a DaSH record stream into segments for storage on disk. This operator does the following:

- Monitors `stdin` for records.
- Upon receipt of a BEGINRUN record, `segmenter` extracts the run number and creates a file named $run\{run_number\}\_0000.evt$.
- The BEGINRUN record is then written to this file.
- Upon receipt of an ENDRUN record, `segmenter` writes the ENDRUN record to file, closes the file and exits.
- In response to a broken pipe, `segmenter` creates a BADEND record, signifying that the run has ended, and writes this record to file and exits.
- All other records are written to file without interpretation.
- If any write would create a file larger than the current segsize, `segmenter` closes the currently open file, increments the segment number, creates a new file named $run\{run_number\}\_\{sequence\}.evt$, and continues writing records.

Synopsis:

`Usage: segmenter [OPTION]... DIRECTORY`

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--segsize=MEGABYTES` Specify run file segment size. Default is 2GB.

Examples:

- `readout --run=56 | segmenter /evt/data/output`
  Use `readout` to produce records for run number 58 and pipe them through `segmenter`. `Segmenter` will write run file segments in directory `/evt/data/output`.
- `readout --run=123 | segmenter --segsize=5 /evt/data/output`
  Use `readout` to produce records for run number 128 and pipe them through `segmenter`. `Segmenter` will write 5MB run file segments in directory `/evt/data/output`.

B.1.27 Sieve

Output only those records from an experiment that have the specified packet types. That is, only those records that have packet types corresponding with those specified using the --packet flag will be written to `stdout`. Records with type BEGINRUN, ENDRUN, BADEND or CONTINUE are also written to `stdout` to enable proper data stream parsing by consumers. All other records are dropped.

Packet types can be specified as either a single positive integer, a range of positive integers or by using one of the well known symbolic names for some well known packet types. Currently, the list of symbolic well known packet types includes: Physics (a.k.a, Data), Scaler, SnapSc, StateVar, RunVar, PktDoc, ParamDescript and Trigger.

Synopsis:

`Usage: sieve --packet=PACKETS [--packet=PACKETS]... [OPTION]...`

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--packet=PACKETS` Mandatory. If more than one occurrence of this flag appears on the command line, all packets and ranges specified will be considered. PACKETS can be either a single positive integer, a range of positive integers (e.g., 101-118 or 101,118) or a symbolic name (e.g., Physics). All ranges are inclusive.

- `--invert` Invert the sense of record type matching. That is, rather than including only those record types specified, only exclude those record types specified.

Examples:

- `readout --run=123 | sieve --packet=101`
  Output only those records that have packet type 101 to `stdout`.
- `readout --run=890 | sieve --packet=105-116 | recorddump`
  Pipe those records that have have a packet type falling in the range from 105 to 116 (inclusive) into the operator `recorddump`.
- `readout --run=567 | sieve --packet=DATA --packet=150 | recorddump`
  Pipe those records that have have a packet type corresponding to the symbolic type of DATA and those records with packet type 150 into the operator `recorddump`.

B.1.28 Stepcutter

Read records that contain an array of time series float values and a trigger signal. Cut the time series into pieces based on when the trigger signal changes to a different integer value.

Synopsis:

```
Usage:  cutter [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--pass` Pass all input records through to output without modification. The option only applies to records that are processed by this operator as indicated by `--packet` or `--signal`. Records that are not processed are automatically passed through to output without modification.
- `--emit=PACKET` Records that are created by this process are emitted with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--signal=PACKET` Records with the type of PACKET are used as the signal for cutting data records (`--packet`). PACKET may be an integer or the symbolic name for a well known packet type such as TRIGGER. Default type is TRIGGER.

Examples:

- `cat audio.wav | wav2rec --samples=768 | steptrigger --pass --emit=trigger | stepcutter --signal=trigger 5 > stepcutter.recs` Pipe audio.wav through **steptrigger** to add trigger records to the data stream that can be used to cut data records into pieces. Then use **stepcutter** to cut the data stream based on the trigger signal.

B.1.29 Steptrigger

Read records that contain an array of time series float values and output a trigger signal that corresponds to the maximum value found within a sliding window. This is a threshold trigger that outputs a positive integer corresponding to a stepwise change in the input values. The size of the step is specified using the **--threshold** parameter.

Synopsis:

`Usage: steptrigger [OPTION]...`

Options:

- **--version** Output version information and exit
- **--help** Display this help and exit
- **--pass** Pass all input records through to output without modification. The option only applies to record that are processed by this operator as indicated by **--packet**. Records that are not processed are automatically passed through to output without modification.
- **--emit=PACKET** Records that are created by this process are emitted with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- **--packet=PACKET** Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- **--samples=INTEGER** The number of samples to encode per output record. Default is 500.
- **--window=INTEGER** Set the sliding window size for selecting a maximum trigger value. Default is 128.
- **--threshold=REAL** Set the threshold value for emitting a high signal. Default is 0.1.

- `--adapt=INTEGER` Indicate that the threshold should be adapted based on the mean and standard deviation of the the data. The INTEGER value indicates the number of standard deviations from the mean that will cause a change in the trigger signal value. Default is not to adapt the threshold.

Examples:

- `cat audio.wav | wav2rec --samples=768 | steptrigger > steptrigger.recs`
  Use `tigger` to create a file containing records of trigger signal data.

B.1.30 Trigger

Read records that contain an array of time series float values and output a trigger signal that corresponds to the maximum value found within a sliding window.

Synopsis:

```
Usage:  trigger [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--pass` Pass all input records through to output without modification. The option only applies to record that are processed by this operator as indicated by `--packet`. Records that are not processed are automatically passed through to output without modification.
- `--emit=PACKET` Records that are created by this process are emitted with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--samples=INTEGER` The number of samples to encode per output record. Default is 500.
- `--window=INTEGER` Set the sliding window size for selecting a maximum trigger value. Default is 128.
- `--threshold=REAL` Set the threshold value for emitting a high signal. Default is 0.5.

- `--adapt=INTEGER` Indicate that the threshold should be adapted based on the mean and standard deviation of the the data. The INTEGER value indicates the number of standard deviations above the mean that will cause a high signal to be emitted. Default is not to adapt the threshold.

Examples:

- `cat audio.wav | wav2rec --samples=768 | trigger > trigger.recs`
  Use `tigger` to create a file containing records of trigger signal data.

B.1.31 Wav2rec

Read a audio WAV file from stdin and output host format float arrays as records.

Synopsis:

```
Usage: wav2rec [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--run=INTEGER` Specify an experiment run number
- `--title=STRING` Experiment title string
- `--packet=INTEGER` Audio data records will have a record type of INTEGER. INTEGER may be an integer or the symbolic name for a well known packet type such as Data. Default type is Data.
- `--samples=INTEGER` Number of samples that comprise each output record.

Examples:

- `cat audio.wav | wav2rec > audio.recs`
  Use `wav2rec` to convert a WAV file into a file of records.

B.1.32 Welchwindow

Read records that contain an array of float values and filter the data using a welch window.

Synopsis:

```
Usage: welchwindow [OPTION]...
```

Options:

- `--version` Output version information and exit

- `--help` Display this help and exit
- `--packet=PACKET` Process records with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as DATA. Default type is DATA.
- `--pass` Pass all input records through to output without modification. This option only applies to records that are processed by this operator as indicated by `--packet`. Records that are not processed are automatically passed through to output without modification.
- `--emit=PACKET` Records that are created by this operator are emitted with the type of PACKET. PACKET may be an integer or the symbolic name for a well known packet type such as DATA. Default type is DATA.

Examples:

- `cat audio.wav | wav2rec --samples=768 | welchwindow > welch.recs` Filter audio WAV data using `welchwindow` to create a file of filtered records.

B.2 Network Pipeline Operators

B.2.1 Streamin

Reads a pipeline record stream from a single network connection and writes it on `stdout`.

Pipeline semantics specify that pipeline termination begins upstream (at the source) and causes the subsequent termination of pipeline operators in response to the closure of of read-side connections. This is intended to reduce the occurrence of data loss by enabling records to be flushed as far downstream as possible as the pipeline is shutdown. As such, loss of the read-side network connection will cause termination when not operating in persistent mode. When in server mode, loss of the read-side network connection will not cause termination only if `--persist` is specified. When in client mode, and `--persist` is specified, closure of the read-side network connection will cause `streamin` to wait for commands on the command pipe (`--fifo`). If no command pipe exists, then `streamin` will terminate.

A SIGHUP will cause `streamin` to exit once scoping has reached a depth of zero. That is, records will continue to be read and written until all open data scopes are closed.

String commands, such as "stop" or "connect localhost 9095" can be passed to `streamin` over the command pipe. Following is a list of available commands:

stop Terminate once the outer most data scope has been reached.

close Close the current network connection.

connect Connect to the specified host. When this command is called with a hostname and port number as arguments, a connection is established. When called with no arguments, the current connection status is returned on the output FIFO.

Synopsis:

```
Usage: streamin [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--server` Run in server mode rather than client mode.
- `--persist` When in server mode, do not terminate when the read-side network connection is closed, but instead attempt to accept a new connection.
- `--port=INTEGER` When running in server mode, this parameter specifies the network port that will be listened on for new connections. When running in client mode, the port number is used in conjunction with a host (specified with `--host`) to establish a connection.
- `--host=STRING` When running in client mode, this parameter specifies the network host to which to establish a connection.
- `--fifo=STRING` Specify the path prefix of a command pipe. Two FIFOs will be created by **streamin**. The first has the suffix ".in" and is used to pass commands to **streamin**. The second has a suffix of ".out" and is used to return results to the user. The directory path must already exist prior to invoking **streamin**.
- `--switch[=INTEGER]` When in client mode, and the connection is closed, switch to server mode. By default, use the port number specified with `--port`. Optionally, a different network port can be specified as an option. This option implies `--persist`.

Examples:

- `streamin --server --port=9090 | recorddump`
  Wait for a connection on port 9090. Once a connection has been established, read records from the connection and write records to **stdout**. **Recorddump** will then print user readable record header information.
- `streamin --port=9090 --host=localhost | recorddump`
  Make a connection to localhost on port 9090. Once a connection has been established, read records from the connection and write records to **stdout**. **Recorddump** will then print user readable record header information.

B.2.2 Streamout

Pipeline semantics specify that pipeline termination begins upstream (at the source) and cause the subsequent termination of pipeline operators in response to the closure of of read-side connections. This is intended to reduce the occurrence of data loss by enabling records to be flushed as far downstream as possible as the pipeline is shutdown. As such, this operator does not terminate when the write-side network connection is lost.

Moreover, to prevent back pressure that inhibits the data source from transmitting or other operators from receiving data in a timely fashion, **streamout** degrades to consumption mode when not connected. That is, **streamout** will continue to read records from **stdin** and simply consume them unless there exists a connection on which to emit records. When in server mode, new connections can be made by a client. When in connection mode, **streamout** degrades permanently to consumption mode until termination. If blocking, instead of consumption, is desired the **--block** option can be used. However, all pipeline traffic will halt if **streamout** is blocking.

Synopsis:

`Usage: streamout [OPTION]...`

Options:

- **--version** Output version information and exit
- **--help** Display this help and exit
- **--block** Block instead of consume when not connected.
- **--server** Run in server mode rather than client mode.
- **--port=INTEGER** When running in server mode, this parameter specifies the network port that will be listened on for new connections. When running in client mode, the port number is used in conjunction with a host (specified with **--host**) to establish a connection.
- **--host=STRING** When running in client mode, this parameter specifies the network host to which to establish a connection.
- **--switch[=INTEGER]** When in client mode, and the connection is closed, switch to server mode. By default, use the port number specified with **--port**. Optionally, a different network port can be specified as an option.

Examples:

- **daqtail --run=123 /evt/data/output | streamout --port=9090 --server**
  Use daqtail to output the records from run 123 found in directory **/evt/data/output** and pipe them into **streamout**. **Streamout** waits for a connection on port 9090 and then reads records from **stdin** and writes them to the network connection.

- `cat /evt/data/output/run0123_0001.evt | streamout --port=9091 --host=localhost`
  Cat the first data file segment from run 123 into `streamout`. `Streamout` will attempt to make a connection to localhost on port 9091.

B.3 Support Programs

B.3.1 Ctrlcmd

Command line utility for sending a command to a remote Dynamic River daemon (`dynriverd`). Commands include:

kill Kill the executing pipeline segment.

running Check if a pipeline segment is running on a specific host.

shutdown Shutdown a `dynriverd` daemon.

signal Set a signal (e.g., SIGHUP) to a pipeline segment.

start Start a new pipeline segment.

status Query the status of a `dynriverd` daemon.

Synopsis:

```
Usage:  ctrlcmd --port=INTEGER [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--unix` Use a Unix socket instead of TCP socket to communicate with `dynriverd`.
- `--port=INTEGER` Mandatory. Network port to use when connecting to `dynriverd`.
- `--host=STRING` Host name where a remote `dynriverd` executes. Default is localhost.

Examples:

- `ctrlcmd --port=1234 start mysegment 9092`
  Instruct the `dynriverd` daemon, running on port 1234, to start the pipeline segment `mysegment` with parameter "9092."
- `ctrlcmd --port=1234 kill`
  Instruct the `dynriverd` daemon, running on port 1234, to kill the current child pipeline segment.

B.3.2 Dynriverd

The Dynamic River daemon. `Dynriverd` accepts commands for starting, stopping and querying the status of pipeline segments.

Although this daemon should **not** be considered security hardened, a number of precautions have been taken to avoid the unauthorized execution of commands. First, all programs and scripts that can be invoked by this daemon must be in the directory specified by `--execpath`. Moreover, all programs and scripts must be owned and executable by the same user that invoked the daemon.

Synopsis:

```
Usage:  dynriverd --port=INTEGER [OPTION]...
```

Options:

- `--version` Output version information and exit
- `--help` Display this help and exit
- `--verbose` Output what the daemon is doing to `stdout`.
- `--daemonize` Run the daemon in background and close `stdin`, `stdout` and `stderr`. Daemonizing implies turning verbosity off.
- `--port=INTEGER` Mandatory. Network port to bind and use for receiving control messages from `ctrlcmd`.
- `--host=STRING` Specify the single host from which commands will be accepted. Default is to accept commands from all hosts.
- `--execpath` Specify the only directory from which commands (pipeline segments) will be executed. The default is the subdirectory `pipes/` under the current working directory.
- `--fifo=STRING` Specify the path prefix of a command pipe. `Dynriverd` sets the environment value `DYNRIVER_FIFO` that can be used by other programs, such as `streamin`, for creating two FIFOs. The first FIFO has suffix ".in" and is used to pass commands to the other program (e.g., `streamin`). The second FIFO has a suffix of ".out" and is used to return results. The directory path must already exist prior to invoking `dynriverd`.

Examples:

- `dynriverd --verbose --port=9100`
  Start `dynriverd` with verbosity turned on and bind to port 9100 to receive commands.

Appendix C

Tabular Data Used for Forecasting Packet Loss

This appendix comprises that tabular data used for producing the plots discussed in Chapter 8. Listed are both the mean accuracies and standard deviations as percentages. Tabulated statistics were generated using the data sets and methodology described in Section 8.4.

Table C.1: MESO forecasting accuracy with loss rate margin. Plotted in Figure 8.9.

Data set	Seconds	Margin 0%	Margin 1%	Margin 2%
Roam	1 sec.	35.8%±4.2%	42.9%±4.7%	49.9%±4.7%
	2 sec.	37.4%±3.6%	45.0%±3.6%	52.4%±4.0%
	3 sec.	40.1%±2.8%	47.7%±3.0%	55.3%±3.0%
	4 sec.	40.6%±2.8%	48.5%±2.8%	57.0%±3.0%
	5 sec.	41.5%±2.1%	49.6%±2.1%	55.7%±2.0%
	6 sec.	44.6%±1.7%	53.9%±1.9%	60.6%±2.0%
	7 sec.	46.1%±2.0%	54.4%±2.0%	61.4%±1.9%
	8 sec.	46.4%±1.4%	55.2%±1.5%	63.3%±1.6%
	9 sec.	46.4%±1.2%	57.5%±1.5%	66.3%±1.6%
	10 sec.	47.8%±1.5%	59.9%±1.6%	67.9%±1.6%
Gsim	1 sec.	14.8%±1.0%	41.6%±1.5%	63.5%±1.4%
	2 sec.	15.7%±1.2%	42.5%±1.6%	64.6%±1.4%
	3 sec.	16.4%±1.1%	46.7%±1.5%	68.5%±1.2%
	4 sec.	16.0%±1.1%	46.9%±1.4%	68.4%±1.3%
	5 sec.	16.8%±1.1%	47.4%±1.4%	71.1%±1.2%
	6 sec.	19.0%±1.2%	52.6%±1.4%	74.7%±1.1%
	7 sec.	18.6%±1.3%	53.4%±1.4%	76.4%±1.1%
	8 sec.	19.9%±1.1%	55.4%±1.5%	78.9%±1.2%
	9 sec.	20.8%±1.2%	57.2%±1.4%	81.3%±1.0%
	10 sec.	22.7%±1.1%	60.8%±1.3%	81.8%±1.0%
Wlsim	1 sec.	55.8%±1.6%	61.1%±1.4%	65.7%±1.3%
	2 sec.	59.6%±1.1%	67.2%±1.1%	72.3%±1.1%
	3 sec.	59.6%±1.2%	67.9%±1.0%	74.4%±1.0%
	4 sec.	62.8%±1.0%	71.2%±1.0%	77.9%±0.9%
	5 sec.	61.5%±0.9%	70.5%±1.1%	77.2%±1.0%
	6 sec.	62.4%±0.9%	71.7%±0.9%	79.4%±0.8%
	7 sec.	63.1%±1.0%	72.9%±0.8%	79.4%±0.7%
	8 sec.	64.2%±1.0%	73.4%±0.9%	80.9%±0.8%
	9 sec.	64.8%±1.0%	76.2%±0.9%	83.5%±0.8%
	10 sec.	63.4%±0.9%	75.4%±0.9%	82.8%±0.8%
Ploss	1 sec.	6.0%±0.8%	13.4%±1.0%	22.2%±1.2%
	2 sec.	9.6%±1.0%	21.8%±1.2%	34.8%±1.4%
	3 sec.	11.2%±0.8%	25.5%±1.2%	39.1%±1.4%
	4 sec.	13.2%±0.9%	29.2%±1.3%	43.1%±1.4%
	5 sec.	14.4%±1.0%	32.1%±1.4%	49.1%±1.3%
	6 sec.	14.5%±0.8%	35.6%±1.3%	52.8%±1.5%
	7 sec.	17.1%±1.0%	41.3%±1.2%	58.1%±1.2%
	8 sec.	17.9%±0.9%	43.5%±1.2%	62.0%±1.3%
	9 sec.	19.8%±1.0%	45.2%±1.3%	64.7%±1.2%
	10 sec.	20.3%±0.9%	48.2%±1.4%	67.7%±1.4%

Table C.1: (cont'd).

Data set	Seconds	Margin 3%	4%	5%
Roam	1 sec.	56.3%±4.7%	60.3%±4.2%	63.1%±4.0%
	2 sec.	56.9%±4.0%	61.2%±3.6%	64.1%±3.6%
	3 sec.	61.3%±2.8%	66.4%±2.8%	69.6%±2.8%
	4 sec.	62.1%±2.8%	66.2%±2.7%	68.0%±2.4%
	5 sec.	62.6%±2.0%	66.6%±2.0%	70.3%±2.0%
	6 sec.	67.9%±2.0%	72.5%±1.5%	76.8%±1.4%
	7 sec.	69.2%±1.4%	74.1%±1.5%	77.3%±1.4%
	8 sec.	69.7%±1.4%	73.9%±1.4%	76.8%±1.4%
	9 sec.	71.0%±1.5%	75.7%±1.4%	78.4%±1.3%
	10 sec.	72.6%±1.4%	77.0%±1.4%	80.5%±1.3%
Gsim	1 sec.	78.1%±1.2%	87.4%±1.0%	93.2%±0.8%
	2 sec.	79.8%±1.1%	88.7%±0.9%	94.1%±0.6%
	3 sec.	82.4%±0.9%	91.0%±0.7%	95.5%±0.5%
	4 sec.	83.6%±1.0%	92.6%±0.8%	96.1%±0.4%
	5 sec.	85.8%±1.0%	93.6%±0.6%	96.8%±0.4%
	6 sec.	87.9%±0.8%	94.6%±0.6%	97.8%±0.3%
	7 sec.	89.0%±0.8%	95.3%±0.6%	98.2%±0.3%
	8 sec.	90.3%±0.9%	95.8%±0.6%	98.4%±0.3%
	9 sec.	91.8%±0.8%	96.4%±0.5%	99.2%±0.3%
	10 sec.	91.8%±0.7%	96.6%±0.4%	99.0%±0.2%
Wlsim	1 sec.	68.2%±1.3%	71.2%±1.2%	72.9%±1.2%
	2 sec.	75.3%±1.1%	77.4%±1.0%	79.3%±1.0%
	3 sec.	78.5%±1.0%	81.3%±0.9%	83.3%±0.9%
	4 sec.	80.8%±0.9%	83.2%±0.8%	85.1%±0.8%
	5 sec.	80.3%±0.9%	83.2%±0.8%	84.6%±0.8%
	6 sec.	83.6%±0.9%	86.4%±0.8%	87.6%±0.7%
	7 sec.	84.0%±0.7%	86.8%±0.7%	89.0%±0.7%
	8 sec.	84.8%±0.8%	87.8%±0.6%	90.3%±0.6%
	9 sec.	87.1%±0.8%	89.7%±0.7%	91.9%±0.7%
	10 sec.	86.3%±0.8%	89.9%±0.6%	91.8%±0.6%
Ploss	1 sec.	30.5%±1.5%	37.7%±1.4%	44.1%±1.5%
	2 sec.	45.5%±1.4%	54.7%±1.4%	63.1%±1.4%
	3 sec.	50.9%±1.4%	60.6%±1.4%	70.3%±1.4%
	4 sec.	56.2%±1.2%	67.3%±1.3%	77.2%±1.3%
	5 sec.	61.5%±1.3%	72.1%±1.2%	80.7%±1.2%
	6 sec.	66.5%±1.3%	76.4%±1.2%	84.8%±0.9%
	7 sec.	71.8%±1.1%	80.4%±1.0%	87.5%±0.9%
	8 sec.	76.7%±1.3%	86.3%±1.0%	92.2%±0.7%
	9 sec.	79.4%±1.0%	87.4%±0.9%	92.5%±0.8%
	10 sec.	81.7%±.10%	88.9%±0.9%	94.0%±0.7%

Table C.2: MESO forecasting accuracy when trained using generated data. Plotted in Figure 8.10.

Data set (Train/Test)	Seconds	Margin 0%	Margin 1%	Margin 2%
Gsim/Roam	1 sec.	4.5%±0.8%	21.4%±4.3%	32.8%±4.4%
	2 sec.	4.5%±1.2%	15.7%±3.6%	34.1%±4.8%
	3 sec.	4.5%±1.1%	14.5%±3.5%	37.1%±6.5%
	4 sec.	5.1%±1.2%	16.1%±4.5%	44.0%±6.7%
	5 sec.	5.1%±1.0%	17.0%±4.4%	42.9%±7.7%
	6 sec.	4.4%±0.7%	17.0%±5.0%	45.6%±8.2%
	7 sec.	4.6%±0.9%	16.5%±4.3%	44.0%±8.2%
	8 sec.	4.7%±1.0%	16.2%±5.1%	44.1%±10.1%
	9 sec.	5.0%±1.0%	15.6%±4.9%	42.4%±9.6%
	10 sec.	5.3%±1.1%	16.9%±5.8%	44.0%±8.9%
Wlsim/Roam	1 sec.	38.8%±3.8%	48.4%±3.2%	53.9%±3.4%
	2 sec.	41.6%±3.7%	47.8%±3.7%	54.0%±3.8%
	3 sec.	37.9%±5.3%	45.2%±5.7%	51.7%±5.5%
	4 sec.	38.7%±6.0%	45.5%±6.4%	52.9%±6.3%
	5 sec.	39.1%±5.5%	46.0%±6.0%	53.3%±6.1%
	6 sec.	36.0%±6.6%	43.2%±7.3%	50.5%±7.7%
	7 sec.	39.0%±6.3%	46.1%±7.0%	52.9%±7.4%
	8 sec.	34.2%±8.4%	40.5%±9.3%	47.4%±10.0%
	9 sec.	36.2%±8.4%	43.5%±9.2%	49.7%±10.0%
	10 sec.	35.8%±7.8%	43.7%±8.1%	49.3%±8.3%
Ploss/Roam	1 sec.	21.3%±4.7%	30.7%±5.8%	48.0%±5.6%
	2 sec.	34.1%±4.6%	48.9%±4.7%	58.1%±3.8%
	3 sec.	33.8%±5.2%	49.7%±5.2%	58.0%±4.4%
	4 sec.	38.6%±5.0%	53.3%±4.2%	60.4%±3.2%
	5 sec.	39.4%±5.1%	54.8%±3.9%	61.2%±2.9%
	6 sec.	40.8%±6.9%	55.1%±4.7%	62.6%±3.6%
	7 sec.	42.9%±6.9%	56.7%±4.3%	63.8%±3.6%
	8 sec.	44.6%±5.9%	57.0%±4.1%	63.9%±3.3%
	9 sec.	45.2%±6.3%	56.3%±4.5%	63.7%±3.8%
	10 sec.	45.7%±5.8%	56.7%±3.3%	64.1%±2.6%

Table C.2: (cont'd).

Data set (Train/Test)	Seconds	Margin 3%	4%	5%
Gsim/Roam	1 sec.	44.4%±4.5%	53.3%±4.5%	63.4%±4.7%
	2 sec.	51.7%±5.1%	61.4%±4.7%	72.5%±4.1%
	3 sec.	56.8%±6.4%	64.8%±5.6%	74.7%±5.1%
	4 sec.	60.2%±5.7%	66.6%±5.3%	76.7%±3.2%
	5 sec.	59.7%±6.7%	67.1%±6.0%	77.1%±4.0%
	6 sec.	60.0%±6.4%	68.0%±5.7%	77.2%±3.8%
	7 sec.	60.2%±7.1%	69.3%±5.0%	78.3%±3.1%
	8 sec.	56.6%±9.7%	66.9%±7.8%	77.9%±5.2%
	9 sec.	57.0%±9.4%	67.1%±7.4%	77.6%±5.2%
	10 sec.	58.0%±8.6%	68.5%±6.4%	78.4%±4.6%
Wlsim/Roam	1 sec.	58.8%±3.4%	61.7%±2.9%	64.1%±2.9%
	2 sec.	60.0%±3.7%	63.3%±3.5%	65.4%±3.5%
	3 sec.	58.7%±5.3%	61.9%±5.2%	64.4%±4.9%
	4 sec.	58.1%±6.4%	60.5%±6.5%	63.2%±6.4%
	5 sec.	57.7%±6.4%	60.7%±6.5%	63.4%±6.5%
	6 sec.	55.1%±7.7%	57.6%±7.7%	61.3%±7.7%
	7 sec.	56.8%±7.5%	60.3%±7.5%	63.6%±7.8%
	8 sec.	51.1%±10.1%	54.9%±10.1%	57.7%±10.2%
	9 sec.	53.6%±9.7%	57.4%±9.8%	60.2%±9.8%
	10 sec.	53.4%±8.6%	57.1%±8.5%	59.8%±8.7%
Ploss/Roam	1 sec.	52.2%±5.4%	57.4%±3.9%	60.3%±3.6%
	2 sec.	62.3%±3.7%	65.8%±3.4%	67.4%±3.1%
	3 sec.	62.7%±4.2%	66.3%±3.8%	69.4%±3.0%
	4 sec.	65.8%±3.0%	69.0%±2.8%	71.7%±2.6%
	5 sec.	67.2%±2.4%	71.0%±2.5%	74.9%±2.2%
	6 sec.	68.1%±2.7%	71.6%±2.6%	75.8%±2.2%
	7 sec.	70.0%±1.4%	73.5%±1.6%	77.3%±1.6%
	8 sec.	69.8%±2.0%	73.0%±2.0%	76.5%±1.8%
	9 sec.	70.6%±1.0%	74.3%±1.2%	77.2%±1.2%
	10 sec.	70.4%±1.5%	74.2%±1.6%	77.5%±1.5%

Table C.3: MESO forecasting accuracy with FEC code labels. Plotted in Figure 8.11(a).

	Data set			
Seconds	Roam	Gsim	Wlsim	Ploss
1 second	49.5%±3.5%	94.3%±0.9%	59.3%±1.6%	41.7%±1.5%
2 second	52.6%±3.1%	93.0%±0.7%	65.1%±1.3%	54.9%±1.4%
3 second	57.1%±2.4%	94.3%±0.6%	67.3%±1.3%	62.2%±1.3%
4 second	57.5%±2.8%	93.5%±0.6%	70.2%±1.0%	67.6%±1.1%
5 second	61.3%±2.2%	94.4%±0.6%	69.1%±0.9%	70.9%±1.2%
6 second	64.9%±1.9%	94.7%±0.4%	72.3%±1.0%	72.8%±1.2%
7 second	65.1%±2.2%	95.3%±0.4%	72.6%±1.1%	77.0%±0.9%
8 second	65.3%±1.6%	95.0%±0.4%	73.1%±1.0%	79.5%±0.9%
9 second	68.9%±1.6%	94.9%±0.4%	74.7%±1.2%	81.0%±1.1%
10 second	71.8%±1.5%	95.6%±0.4%	73.8%±1.1%	83.5%±0.9%

Table C.4: MESO accuracy when trained using generated data with FEC code labels. Plotted in Figure 8.11(b)

	Data set		
Seconds	Gsim/Roam	Wlsim/Roam	Ploss/Roam
1 second	33.3%±0.4%	45.4%±3.3%	35.7%±3.5%
2 second	34.8%±0.6%	49.6%±3.2%	45.0%±3.6%
3 second	33.4%±0.5%	46.1%±4.9%	43.6%±4.8%
4 second	33.9%±0.7%	46.9%±5.7%	48.7%±4.4%
5 second	33.5%±0.4%	49.4%±5.7%	51.1%±4.5%
6 second	33.6%±0.4%	46.3%±6.7%	53.0%±6.0%
7 second	34.1%±0.5%	49.4%±6.4%	54.9%±6.2%
8 second	33.7%±0.6%	45.2%±8.7%	55.2%±5.9%
9 second	33.9%±0.5%	48.0%±8.3%	57.1%±6.1%
10 second	33.8%±0.6%	48.1%±8.0%	58.3%±6.1%

Table C.5: MESO forecasting accuracy with Xnaut FEC code labels. Plotted in Figure 8.12(a)

Seconds	Data set			
	Roam	Gsim	Wlsim	Ploss
1 second	67.7%±2.7%	96.5%±0.8%	73.1%±1.5%	67.2%±1.2%
2 second	72.2%±2.2%	96.6%±0.5%	81.3%±1.1%	74.8%±0.9%
3 second	75.6%±2.0%	96.8%±0.5%	80.9%±1.3%	78.9%±0.9%
4 second	77.1%±2.4%	96.3%±0.4%	83.3%±0.9%	82.1%±0.6%
5 second	79.2%±1.8%	96.9%±0.4%	82.3%±0.9%	84.0%±0.9%
6 second	80.8%±1.6%	96.9%±0.4%	83.5%±0.9%	85.4%±0.8%
7 second	82.5%±1.9%	92.3%±0.3%	83.9%±1.0%	87.2%±0.7%
8 second	82.6%±1.4%	97.2%±0.3%	84.1%±1.0%	88.8%±0.7%
9 second	83.9%±1.2%	97.0%±0.4%	85.0%±1.0%	88.7%±0.7%
10 second	85.3%±1.3%	97.4%±0.3%	83.4%±1.0%	90.2%±0.7%

Table C.6: MESO accuracy when trained using generated data with Xnaut FEC code labels. Plotted in Figure 8.12(b)

Seconds	Data set		
	Gsim/Roam	Wlsim/Roam	Ploss/Roam
1 second	50.6%±0.3%	60.0%±2.4%	53.3%±2.8%
2 second	50.9%±0.4%	67.3%±2.8%	60.0%±3.3%
3 second	50.8%±0.3%	62.0%±4.5%	57.5%±4.2%
4 second	50.8%±0.5%	64.4%±4.8%	60.9%±4.2%
5 second	50.5%±0.2%	65.7%±4.5%	62.8%±4.3%
6 second	50.6%±0.3%	63.3%±5.2%	65.0%±5.3%
7 second	50.6%±0.2%	65.9%±4.7%	66.8%±5.6%
8 second	50.5%±0.3%	62.1%±6.9%	67.2%±5.6%
9 second	50.6%±0.4%	63.2%±6.6%	68.9%±5.7%
10 second	50.5%±0.4%	63.8%±6.4%	69.9%±5.9%

Bibliography

Bibliography

[1] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM Computer Surveys*, vol. 31, pp. 264–323, September 1999.

[2] L. Bergmans, "The composition filters object model," tech. rep., Department of Computer Science, University of Twente, 1994.

[3] N. Amano and T. Watanabe, "An approach for constructing dynamically adaptable component-based software systems using LEAD++," in *OOPSLA International Workshop on Object Oriented Reflection and Software Engineering*, (Denver, Colorado, USA), pp. 1–16, November 1999.

[4] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification, Second Edition.* New York, New York, USA: John Wiley and Sons, Incorporated, 2001.

[5] J. Lin, E. Keogh, S. Lonardi, and B. Chiu, "A symbolic representation of time series with implications for streaming algorithms," in *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, (San Diego, California, USA), June 2003.

[6] N. Kumar, N. Lolla, E. Keogh, S. Lonardi, and C. A. Ratanamahatana, "Time-series bitmaps: A practical visualization tool for working with large time series databases," in *Proceedings of SIAM International Conference on Data Mining (SDM'05)*, (Newport Beach, California, USA), pp. 531–535, April 2005.

[7] C. Tang and P. K. McKinley, "Modeling multicast packet losses in wireless LANs," in *Proceedings of the Sixth ACM International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems (MSWiM) (in conjunction with ACM Mobicom 2003)*, (San Diego, California, USA), pp. 130–133, September 2003.

[8] "Proceedings of the ACM workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), held in conjunction with the 16th Annual ACM International Conference on Supercomputing," June 2002.

[9] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng, "Composing adaptive software," *IEEE Computer*, vol. 37, pp. 56–64, July 2004.

[10] *Proceedings of the Distributed Auto-adaptive and Reconfigurable Systems Workshop (DARES), held in conjunction with the 24th IEEE International Conference on Distributed Computing Systems (ICDCS).* Tokyo, Japan, March 2004.

[11] *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC'04).* New York, New York, USA, May 2004.

[12] *Proceedings of the Second IEEE International Conference on Autonomic Computing (ICAC).* Seattle, Washington, USA, June 2005.

[13] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, pp. 41–50, January 2003.

[14] J. Porter, P. Arzberger, H.-W. Braun, P. Bryant, S. Gage, T. Hansen, P. Hanson, C.-C. Lin, F.-P. Lin, T. Kratz, W. Michener, S. Shapiro, and T. Williams, "Wireless sensor networks for ecology," *Bioscience*, vol. 55, pp. 561–572, July 2005.

[15] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An analysis of a large scale habitat monitoring application," in *Proceedings of The Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, (Baltimore, Maryland, USA), November 2004.

[16] T. Abdelzaher, B. Blum, Q. Cao, D. Evans, J. George, S. George, T. He, L. Luo, S. Son, and R. Stoleru, "EnviroTrack: towards an environmental computing paradigm for distributed sensor networks," in *Proceedings SenSys 2003*, (Los Angeles, California, USA), November 2003.

[17] W. Bourgeois, A.-C. Romain, J. Nicolas, and R. M. Stuetz, "The use of sensor arrays for environmental monitoring: Interests and limitations," *Journal of Environmental Monitoring*, vol. 5, pp. 852–860, 2003.

[18] D. Estrin, W. Michener, and G. Bonito, "Environmental cyberinfrastructure needs for distributed sensor networks: A report from a national science foundation sponsored workshop," tech. rep., Scripps Institute of Oceanography, August 2003. 12 May 2005; www.lternet.edu/sensor_report/.

[19] "Environmental sensor networks: A revolution in the earth system science?," *Earth-Science Reviews*, vol. 78, pp. 177–191, 2006.

[20] K. Martinez, J. K. Hart, and R. Ong, "Environmental sensor networks," *IEEE Computer*, vol. 37, pp. 50–56, August 2004.

[21] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin, "Habitat monitoring with sensor networks," *Communications of the ACM*, vol. 47, pp. 34–40, June 2004.

[22] A. Suri, S. Iyengar, and E. Cho, "Ecoinformatics using wireless sensor networks: An overview," *Ecological Informatics*, vol. 1, pp. 213–340, November 2006. 4th International Conference on Ecological Informatics.

[23] P. Arzberger, ed., *Sensors for Environmental Observatories*, (Seattle, Washington, USA), World Technology Evaluation Center (WTEC) Inc., Baltimore, Maryland, December 2004. Report of a NSF sponsored workshop.

[24] "National ecological observatory network (NEON)." `http://www.neoninc.org`, November 2006.

[25] S. A. Isard and S. H. Gage, *Flow of Life in the Atmosphere: An airscape approach to understanding invasive organisms.* East Lansing, Michigan, USA: Michigan State University Press, 2001.

[26] A. O'Donnell, "Invasive species: Closing the door to exotic hitchhikers," tech. rep., National Invasive Species Information Center, November 2006. Land Letter: The Natural Resources Weekly Report.

[27] "National invasive species information center (NISIC)." `http://www.invasivespeciesinfo.gov/`, December 2006.

[28] The 2020 Science Group, "Towards 2020 science." `http://research.microsoft.com/towards2020science/downloads/T2020S_ReportA4.pdf`, June/July 2005. Report from the Towards 2020 Science Workshop.

[29] B. C. Smith, "Reflection and semantics in Lisp," in *Proceedings of 11th ACM Symposium on Principles of Programming Languages*, 1984.

[30] P. Maes, "Concepts and experiments in computational reflection," in *Proceedings of the ACM Conerfence on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 147–155, December 1987.

[31] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," in *Proceedings, Usenix Winter 1995 Technical Conference*, (New Orleans, Lousiana, USA), pp. 213–223, January 1995.

[32] S. Bouchenak and D. Hagimont, "Pickling threads state in the java system," in *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS)*, (St. Malo, USA, France), June 2000.

[33] M. Ma, C. Wang, and F. Lau, "Delta execution: A preemptive Java thread migration mechanism," *Cluster Computing*, vol. 3, no. 2, pp. 83–94, 2000.

[34] S. Zhang, M. Khambatti, and P. Dasgupta, "Process migration through virtualization in a computing community," in *13th IASTED Conference on Parallel and Distributed Computing Systems (PDCS2001)*, (Dallas, Texas, USA), August 2001.

[35] S. Fünfrocken, "Transparent migration of java-based mobile agents," in *Proceedings of Second International Workshop on Mobile Agents 98*, (Stuttgart, Germany), pp. 26–37, September 1998.

[36] T. Watanabe, A. Noriki, and K. Shinbori, "Towards a substrate for reliable mobile agent systems," in *Proceedings of the Workshop on Reflective Middleware*, (Palisades, New York, USA), April 2000.

[37] E. P. Kasten, P. K. McKinley, S. M. Sadjadi, and R. Stirewalt, "Separating introspection and intercession to support metamorphic distributed systems," in *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems ICDCS'02*, (Vienna, Austria), July 2002.

[38] P. K. McKinley, E. P. Kasten, S. M. Sadjadi, and Z. Zhou, "Realizing multidimensional software adaptation," in *Proceedings of the ACM Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), held in conjunction with the 16th Annual ACM International Conference on Supercomputing*, (New York, New York, USA), June 2002.

[39] S. Sadjadi, P. K. McKinley, and E. P. Kasten, "Metasockets: Run-time support for adaptive communication services," in *Addendum to the proceedings of the International Symposium on Distributed Object and Applications (DOA)*, (Irvine, California, USA), pp. 42–45, November 2002.

[40] P. K. McKinley, S. M. Sadjadi, and E. P. Kasten, "An adaptive software approach to intrusion detection and response," in *Proceedings of the 10th International Conference on Telecommunication Systems, Modeling and Analysis (ICTSM10)*, (Monterey, California,USA), October 2002.

[41] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and R. Kalaskar, "Programming languages support for adaptive wearable computing," in *Proceedings of the International Symposium on Wearable Computers (ISWC02)*, (Seattle, Washington, USA), October 2002.

[42] S. M. Sadjadi, P. K. McKinely, and E. P. Kasten, "Architecture and operation of an adaptable communication substrate," in *Proceedings of the 9th International Workshop on Future Trends of Distributed Computing Systems (FTDCS '03)*, (San Juan, Puerto Rico), May 2003.

[43] S. M. Sadjadi, P. K. McKinley, E. P. Kasten, and Z. Zhou, "Metasockets: Design and operation of run-time reconfigurable communication services," *Software: Practice and Experience (SP&E). Special Issue: Experiences with Auto-adaptive and Reconfigurable Systems.*, vol. 36, pp. 1157–1178, 2006.

[44] E. P. Kasten and P. K. McKinley, "Perimorph: Run-time composition and state management for adaptive systems," in *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems ICDCS'04*, (Tokyo, Japan), March 2004.

[45] J. M. Fuster, *Memory in the Cerebral Cortex: An Empirical Approach to Neural Networks in the Human and Nonhuman Primate.* Cambridge, Massachusetts, USA: The MIT Press, 1995.

[46] S. Franklin, "Perceptual memory and learning: Recognizing, categorizing and relating," in *Proceedings of the Developmental Robotics AAAI Spring Symposium*, (Stanford, CA, USA), March 2005.

[47] E. P. Kasten and P. K. McKinley, "MESO: Perceptual memory to support online learning in adaptive software," in *Proceedings 3rd International Conference on Development and Learning (ICDL'04)*, (La Jolla, California, USA), October 2004.

[48] E. P. Kasten and P. K. McKinley, "MESO: Supporting online decision making in autonomic computing systems," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 19, pp. 485–499, April 2007. Featured article.

[49] K. Munagala, S. Babu, R. Motwani, and J. Widom, "The pipelined set cover problem," in *Proceedings of the Tenth International Conference on Database Theory (ICDT)*, (Edinburgh, Scotland), January 2005.

[50] W. Aspray, "John von Neumann's contributions to computing and computer science," *Annals of the History of Computing*, vol. 11, pp. 189–195, Fall 1989.

[51] F. Hohl, "Time limited blackbox security: Protecting mobile agents from malicious hosts," in *Lecture Notes in Computer Science* (G. Vigna, ed.), vol. 1419, pp. 92–113, Berlin, Germany: Springer-Verlag, 1998.

[52] N. Venkatasubramanian, "Safe 'composability' of middleware services," *Communications of the ACM*, vol. 45, pp. 49–52, June 2002.

[53] M. A. Hiltunen and R. D. Schlichting, "Adaptive distributed and fault-tolerant systems," *International Journal of Computer Systems Science and Engineering*, vol. 11, pp. 125–133, September 1996.

[54] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, pp. 1053–1058, December 1972.

[55] K. Czarnecki and U. Eisenecker, *Generative Programming.* Boston, Massachusetts, USA: Addison-Wesley, May 2000.

[56] E. Avdičaušević, M. Mernik, M. Lenič, and V. Žumer, "Experimental aspect-oriented language - AspectCOOL," in *Proceedings of 17th ACM Symposium on Applied Computing, SAC 2002*, (Madrid, Spain), pp. 943–947, 2002.

[57] E. Truyen, W. Joosen, and P. Verbaeten, "Run-time support for aspects in distributed system infrastructure," in *Proceedings of the First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'2002)*, (Enschede, Netherlands), 2002.

[58] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with AspectJ," *Communications of the ACM*, vol. 44, pp. 59–65, October 2001.

[59] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (M. Aksit and S. Matsuoka, eds.), Springer-Verlag, June 1997. volume 1241 of Lecture Notes in Computer Science.

[60] C. Szyperski, *Component Software: Beyond Object-Oriented Programming, 2nd ed.* Boston, Massachusetts, USA: Addison-Wesley, 2002.

[61] M. Mezini and K. Lieberherr, "Adaptive plug-and-play components for evolutionary software development," *ACM SIGPLAN Notices, Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, vol. 33, October 1998.

[62] R. Keller and U. Hölzle, "Binary component adaptation," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 1998)* (E. Jul, ed.), (Brussels, Belgium), pp. 307–329, Springer-Verlag, July 1998. volume 1445 of Lecture Notes in Computer Science.

[63] G. A. Cohen, J. S. Chase, and D. L. Kaminsky, "Automatic program transformation with JOIE," in *Proceedings of the USENIX Annual Technical Symposium*, (New Orleans, Louisiana, USA), pp. 167–178, June 1998.

[64] A. Duncan and U. Hölzle, "Load-time adaptation: Efficient and non-intrusive language extension for virtual machines," Tech. Rep. TRCS99-09, Department of Computer Science University of California, Santa Barbara, Santa Barbara, California, USA, April 1999.

[65] D. Alexander, M. Shaw, S. Nettles, and J. Smith, "Active bridging," in *Proceedings ACM SIGCOMM 1997*, (Cannes, France), September 1997.

[66] M. A. Hiltunen and R. D. Schlichting, "The cactus approach to building configurable middleware services," in *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, (Nuremberg, Germany), October 2000.

[67] R. Litiu and A. Prakash, "DACIA: A mobile component framework for building adaptive distributed applications," in *Principles of Distributed Computing (PODC) 2000 Middleware Symposium*, (Portland, Oregon, USA), July 2000. also appeared as Technical Report CSE-TR-416-99, Department of EECS, University of Michigan, Dec 1999.

[68] F. Kon, M. Romàn, P. Liu, J. Mao, T. Yamane, L. C. Magalhaes, and R. Campbell, "Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB," in *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, (Hudson River Valley, New York, USA), pp. 3–7, April 2000.

[69] E. Bruneton and M. Riveill, "Reflective implementation of non-functional properties with the JavaPod component platform," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2000): Workshop On Reflection and Metalevel Architectures*, (Sophia Antipolis and Cannes, France), June 2000.

[70] F. Akkawi, A. Bader, and T. Elrad, "Dynamic weaving for building reconfigurable software systems," in *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, (Tampa Bay, Florida, USA), October 2001.

[71] A. Popovici, T. Gross, and G. Alonso, "Dynamic weaving for aspect-oriented programming," in *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, (Enschede, The Netherlands), pp. 141–147, ACM Press, 2002.

[72] L. Bergmans and M. Aksit, "Composing crosscutting concerns using composition filters," *Communications of the ACM*, vol. 44, pp. 51–57, October 2001.

[73] B. Redmond and V. Cahill, "Supporting unanticipated dynamic adaptation of application behaviour," in *Proceedings of the 16th European Conference on Object-Oriented Programming*, Malaga, Spain: Springer-Verlag, June 2002. volume 2374 of Lecture Notes in Computer Science.

[74] A. Oliva and L. E. Buzato, "The design and implementation of Guaraná," in *Proceedings 5th USENIX Conference on Object-Oriented Technologies and Systems*, (San Diego, California, USA), May 1999.

[75] F. Costa, H. Duran, N. Parlavantzas, K. Saikoski, G. Blair, and G. Coulson, "The role of reflective middleware in supporting the engineering of dynamic applications"," *Reflection and Software Engineering*, pp. 79–98, 2000.

[76] J. des Riviéres and B. C. Smith, "The implementation of procedurally reflective languages," in *Conference Record of the 1984 ACM Symposium on LISP and functional programming*, (Austin, Texas, USA), pp. 331–347, 1984.

[77] L. Capra, W. Emmerich, and C. Mascolo, "Reflective middleware solutions for context-aware applications," in *Proceedings of Reflection 2001, Lecture Notes in Computer Science*, (Kyoto, Japan), Springer Verlag., 2001.

[78] B. Foote and R. E. Johnson, "Reflective facilities in Smalltalk-80," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages,*

and Applications (OOPSLA), (New Orleans, Louisiana, USA), pp. 327–335, October 1989.

[79] M. Golm and J. Kleinöder, "Jumping to the meta level: Behavioral reflection can be fast and flexible," in *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection (Reflection'99)*, (Saint-Malo, France), pp. 22–39, July 1999.

[80] M. Aksit, L. Bergmans, and S. Vural, "An object-oriented language-database integration model: The composition-filters approach," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, (Utrecht, Netherlands), pp. 372–395, June 1992.

[81] G. Kiczales, J. des Rivières, and D. Bobrow, *The Art of the Metaobject Protocol*. Cambridge, Massachusetts, USA: MIT Press, 1991.

[82] G. Blair and M. Papathomas, "The case for reflective middleware," in *Proceedings of the 3rd Cabernet Plenary Workshop*, (Rennes, France), April 1997.

[83] J. Itoh, R. Lea, and Y. Yokote, "Adaptive operating system design using reflection," in *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, May 1995.

[84] E. P. Kasten and P. K. McKinley, "A taxonomy for computational adaptation," Tech. Rep. MSU-CSE-04-4, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, USA, January 2004.

[85] B. Tekinerdogan and M. Aksit, "Adaptability in object-oriented software development workshop report," in *Proceedings of the 10th Annual European Conference on Object-Oriented Programming (ECOOP)*, (Linz, Austria), July 1996.

[86] A. K. Dey and G. D. Abowd, "The Context Toolkit: Aiding the development of context-aware applications," in *Proceedings of the 22nd International Conference on Software Engineering (ICSE): Workshop on Software Engineering for Wearable and Pervasive Computing*, (Limerick, Ireland), June 2000.

[87] Information Sciences Institute University of Southern California, "RFC 793: Transmission control protocol." `http://www.faqs.org/rfcs/rfc793.html`, September 1981.

[88] V. Jacobson, "Congestion avoidance and control," in *Proceedings of the SIGCOMM '88 Symposium*, pp. 314–332, August 1988.

[89] J. Flinn, E. de Lara, M. Satyanarayanan, D. S. Wallach, and W. Zwaenepoel, "Reducing the energy usage of office applications," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, (Heidelberg, Germany), November 2001.

[90] V. Adve, V. V. Lam, and B. Ensink, "Language and compiler support for adaptive distributed applications," in *ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001)*, (Snowbird, Utah, USA), June 2001. Held in conjunction with PLDI2001.

[91] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting, "Constructing adaptive software in distributed systems," in *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21)*, (Mesa, Arizona, USA), pp. 635–643, April 2001.

[92] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr, "Building adaptive systems using Ensemble," Tech. Rep. TR97-1638, Department of Computer Science, Cornell University, Ithaca, New York, USA, July 1997.

[93] P. K. McKinley and U. I. Padmanabhan, "Design of composable proxy filters for heterogeneous mobile computing," in *Proceedings of the Second International Workshop on Wireless Networks and Mobile Computing*, 2001.

[94] P. Costanza, "Dynamic object replacement and implementation-only classes," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001): 6th International Workshop on Component-Oriented Programming (WCOP 2001)*, (Budapest, Hungary), June 2001.

[95] "Distributed operating systems group: Computing communities project." `http://calypso.eas.asu.edu`, March 2004.

[96] R. Koster, *A Middleware Platform for Information Flows.* PhD thesis, University of Kaiserslautern, Hombrechtikon, Switzerland, July 2002.

[97] V. Kumar, B. F. Cooper, and K. Schwan, "Distributed stream management using utility-driven self-adaptive middleware," in *2nd IEEE International Conference on Autonomic Computing (ICAC)*, (Seattle, Washington, USA), June 2005.

[98] D. Batory and B. J. Geraci, "Composition validation and subjectivity in genvoca generators," in *IEEE Transactions on Software Engineering*, pp. 67–82, feb 1997.

[99] V. Singhal and D. Batory, "P++: A language for large-scale reusable software components," in *6th Annual Workshop on Software Reuse*, (Owego, New York, USA), November 1993.

[100] D. M. Hoffman and D. M. Weiss, eds., *Software Fundamentals: Collected Papers by David L. Parnas*, ch. 10. Boston, Massachusetts, USA: Addison-Wesley, 2001. On the Design and Development of Program Families.

[101] R. Koster and C. Pu, "Infopipes for composing distributed information flows," in *The ACM Multimedia Workshop on Multimedia Middleware*, (Ottawa, Canada), October 2001.

[102] A. P. Black, J. Huang, R. Koster, J. Walpole, and C. Pu, "Infopipes: An abstraction for multimedia streaming," *Multimedia Systems (special issue on Multimedia Middleware)*, vol. 8, no. 5, pp. 406–419, 2002.

[103] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu, "Thread transparency in information flow middleware," *Software: Practice and Experience (SP&E)*, vol. 33, pp. 321–349, April 2003.

[104] "The SwitchWare project." `http://www.cis.upenn.edu/~switchware/`, April 2006.

[105] P. Bridges, W.-K. C. Matti, A. Hiltunen, and R. D. Schlichting, "Supporting coordinated adaptation in networked systems." `ftp://ftp.cs.arizona.edu/ftol/papers/hotos.pdf`, May 2001. A one page summary of this paper was accepted as a position paper in the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII).

[106] D. Lafferty and V. Cahill, "Real world evaluation of aspect-oriented programming with Iguana," in *Proceedings of the International Workshop on Aspects and Dimensional Programming at the 14th European Conference on Object-Oriented Programming (ECOOP)*, (Sophia Antipolis and Cannes, France), June 2000.

[107] A. Gavrilovska, S. Kumar, S. Sundaragopalan, and K. Schwan, "Platform overlays: Enabling in-network stream processing in large-scale distributed applications," in *Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, (Skamania, Washington, USA), June 2005.

[108] G. Eisenhauer, F. Bustamente, and K. Schwan, "Event services for high performance computing," in *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC-9)*, (Pittsburgh, Pennsylvania, USA), August 2000.

[109] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti, "CANS: Composable, adaptive network services infrastructure," in *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, (San Francisco, California, USA), pp. 135–146, March 2001.

[110] X. Fu and V. Karamcheti, "Automatic creation and reconfiguration of network-aware service access paths," *Computer Communications*, vol. 28, pp. 591–608, April 2005.

[111] B. Li, W. Jeon, W. Kalter, K. Nahrstedt, and J. Seo, "Adaptive middleware architecture for a distributed omni-directional visual tracking system," in *Proceedings of SPIE Multimedia Computing and Networking 2000 (MMCN'00)*, January 2000.

[112] B. Li and K. Nahrstedt, "A control-based middleware framework for quality of service adaptations," *IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms*, vol. 17, September 1999.

[113] *IBM Systems Journal, Special issue on Autonomic Computing*, vol. 42, no. 1, 2003.

[114] M. Wang and T. Suda, "The bio-networking architecture: A biologically inspired approach to the design of scalable, adaptive, and survivable/available network applications," Tech. Rep. 00-03, Department of Information and Computer Science, Unversity of California, Irvine, California, USA, February 2000.

[115] N. Arshad, D. Heimbigner, and A. L. Wolf, "Deployment and dynamic reconfiguration planning for distributed software systems," in *Proceedings of the 15th International Conference on Tools with Artificial Intelligence (ICTAI'03)*, (Sacramento, California, USA), November 2003.

[116] N. H. Gardiol and L. P. Kaelbling, "Computing action equivalences for planning under time constraints," Tech. Rep. MIT-CSAIL-TR-2006-022, Massachusetts Institute of Technology CS & AI Laboratory, Cambridge, Massachusetts, USA, 2006.

[117] D. O. Keck and P. J. Kuehn, "The feature and service interaction problem in telecommunications systems: A survey," *IEEE Transactions on Software Engineering*, vol. 24, pp. 779–796, October 1998.

[118] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec, "Feature interaction: A critical review and considered forecast," *Computer Networks*, vol. 41, no. 1, pp. 115–141, 2002.

[119] P. Robertson and R. Laddaga, "A self-adaptive architecture and its application to robust face identification," in *Proceedings of the 7th Pacific Rim Conference on Artificial Intelligence*, Springer-Verlag, 2002. volume 2417 of Lecture Notes in Computer Science.

[120] D. Isla, R. Burke, M. Downie, and B. Blumberg, "A layered brain architecture for synthetic creatures," in *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, (Seattle, Washington, USA), August 2001.

[121] T. Jebara and A. Pentland, "Statistical imitative learning from perceptual data," in *Proceedings of the Second International Conference on Development and Learning*, (Boston, Massachusetts, USA), pp. 191–196, June 2002.

[122] M. Weiser, "The computer for the twenty-first century," *Scientific American*, pp. 94–110, 1991.

[123] R. Laddaga, M. L. Swinson, and P. Robertson, "Seeing clearly and moving forward," *IEEE Intelligent Systems*, vol. 15, pp. 46–50, November/December 2000.

[124] K. Gödel, "Über formal unentscheidbare sätze der principia mathematica und verwandter systeme I," *Monatsh. Math. Phys.*, vol. 38, pp. 235–242, 1931. Gödel's 1931 writings on undecidability in axiomatic systems.

[125] K. Gödel, *On Formally Undecidable Propositions of Principia Mathematic and Related Systems.* Dover Publications, Incorporated, 1992. translation of Gödel's 1931 writings.

[126] C. Strachey, "Correspondence: An impossible program," *The Computer Journal*, p. 313, January 1965.

[127] G. Kiczales, "Towards a new model of abstraction in the engineering of software," in *International Workshop on Reflection and Meta-Level Architecture*, (Tama-City, Tokyo, Japan), November 1992.

[128] M. Shaw and W. Wulf, "Towards relaxing assumptions in languages and their implementations," *ACM SIGPLAN Notices*, vol. 15, pp. 45–61, March 1980.

[129] H. Duran-Limon and G. Blair, "A resource management framework for adaptive middleware," in *3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2K)*, (Newport Beach, California), March 2000.

[130] D. Batory and S. O'Malley, "The design and implementation of hierarchical software systems with reusable components," in *ACM TOSEM*, October 1992.

[131] Usability Center, Georgia Institute of Technology, *CUP User's Manual*, July 1999.

[132] S. M. Sadjadi, P. K. McKinley, B. H. Cheng, and R. K. Stirewalt, "TRAP/J: Transparent generation of adaptable java programs," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, (Agia Napa, Cyprus), October 2004.

[133] S. Fleming, B. H. Cheng, R. K. Stirewalt, and P. K. McKinley, "An approach to implementing dynamic adaptation in c++," in *Proceedings of the ICSE Workshop on Design and Evolution of Autonomic Application Software (DEAS)*, (St. Louis, Missouri, USA), May 2005.

[134] P. K. McKinley and A. P. Mani, "An experimental study of adaptive forward error correction for wireless collaborative computing," in *Proceedings of the IEEE 2001 Symposium on Applications and the Internet (SAINT-01)*, (San Diego-Mission Valley, California, USA), January 2001.

[135] A. J. McAuley, "Reliable broadband communications using burst erasure correcting code," in *Proceedings of ACM SIGCOMM*, pp. 287–306, September 1990.

[136] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *ACM Computer Communication Review*, vol. 27, pp. 24–36, April 1997.

[137] E. P. Kasten, *DAQ Superhighway (DaSH) Reference Manual.* National Superconducting Cyclotron Laboratory, Michigan State University, East Lansing, Michigan, USA, 2006.

[138] P. D. Welch, "The use of the fast Fourier transform for the estimation of power spectra: A method based on time-averaging over short, modified periodograms," *IEEE Transactions on Audio and Electroacoustics*, vol. AU-15, pp. 70–73, June 1967.

[139] "The TAILOR project." `http://javalab.iai.uni-bonn.de/research/tailor/`, April 2006.

[140] L. Girod, K. Jamieson, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden, "The case for a signal-oriented data stream management system," in *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR)*, (Pacific Grove, California, USA), January 2007.

[141] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams: A new class of data management applications," in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, (Hong Kong, China), August 2002.

[142] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *VLDB Journal*, vol. 12, pp. 120–139, August 2003.

[143] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the Borealis stream processing engine," in *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*, (Pacific Grove, California, USA), January 2005.

[144] S. Zdonik, U. Çetintemel, M. Stonebraker, M. Balazinska, M. Cherniack, and H. Balakrishnan, "The Aurora and Medusa projects," *IEEE Data Engineering Bulletin*, vol. 26, March 2003.

[145] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Indianapolis, Indiana, USA: Addison-Wesley, 1995.

[146] "Rocky mountain mapping center: Elevation program." `http://rmmcweb.cr.usgs.gov/elevation/`, April 2006.

[147] G. Candea, A. B. Brown, A. Fox, and D. Patterson, "Recovery-oriented computing: Building multitier dependability," *IEEE Computer*, vol. 37, pp. 60–67, November 2004. Cover feature article.

[148] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. Singh, "Mobile agent programming in Ajanta," in *Proceedings of the 19th International Conference on Distributed Computing Systems*, (Austin, Texas, USA), pp. 314–322, 1999.

[149] G. Karjoth, D. B. Lange, and M. Oshima, "A security model for Aglets," *IEEE Internet Computing*, pp. 68–77, July–August 1997.

[150] E. Kendall, P. M. Krishna, C. Pathak, and C. Suresh, "Patterns of intelligent and mobile agents," in *Proceedings of teh 2nd International Confernece on Autonomous Agents (AGENTS-98)*, (New York, New York, USA), pp. 92–99, May 1998.

[151] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot – A techinique for cheap recovery," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, (San Francisco, California, USA), December 2004.

[152] A. Fox and D. Patterson, "Self-repairing computers," *Scientific American*, pp. 54–61, 2003.

[153] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox, "Autonomous recovery in componentized internet applications," *Cluster Computing Journal*, 2004.

[154] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining.* Boston, Massachusetts, USA: Pearson Education, Incorporated, 2006.

[155] W.-S. Hwang and J. Weng, "Hierarchical discriminant regression," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, November 2000.

[156] J. A. Hartigan, *Clustering Algorithms.* New York, New York, USA: John Wiley and Sons, Inc., 1975.

[157] Y. A. Ivanov and B. M. Blumberg, "Developmental learning of memory-based perceptual models," in *Proceedings of the Second International Conference on Development and Learning*, (Boston, Massachusetts, USA), pp. 165–171, June 2002.

[158] W. Buntine, "Tree classification software," in *Proceedings of the Third National Technology Transfer Conference and Exposition*, (Baltimore, Maryland, USA), December 1992.

[159] C. L. Blake and C. J. Merz, "UCI repository of machine learning databases." http://www.ics.uci.edu/~mlearn/MLRepository.html, 1998.

[160] S. Hettich and S. D. Bay, "UCI KDD archive." http://kdd.ics.uci.edu, 1999.

[161] F. Samaria and A. Harter, "Parameterisation of a stochastic model for human face identification," in *Proceedings of Second IEEE Workshop on Applications of Computer Vision*, (Sarasota, Florida, USA), December 1994.

[162] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, November 1998.

[163] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, pp. 179–188, 1936.

[164] M. van Breukelen, R. P. W. Duin, D. M. J. Tax, and J. E. den Hartog, "Handwritten digit recognition by combined classifiers," *Kybernetika*, vol. 34, no. 4, pp. 381–386, 1998.

[165] A. K. Jain, R. P. W. Duin, and J. Mao, "Statistical pattern recognition: A review," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, pp. 4–37, January 2000.

[166] P. W. Frey and D. J. Slate, "Letter recognition using holland-style adaptive classifiers," *Machine Learning*, vol. 6, March 1991.

[167] J. S. Schlimmer, *Concept Acquisition Through Representational Adjustment.* PhD thesis, Department of Information and Computer Science, University of California, Irine, Irvine, California, USA, 1987.

[168] M. Kudo, J. Toyama, and M. Shimbo, "Multidimensional curve classification using passing-through regions," *Pattern Recognition Letters*, vol. 20, pp. 1103–1111, 1999.

[169] J. A. Blackard and D. J. Dean, "Comparative accuracies of neural networks and discriminant analysis in predicting forest cover types from cartographic variables," in *Proceedings of the Second Southern Forestry GIS Conference*, (Athens, Georgia, USA), pp. 189–199, 1998.

[170] S. Murthy, S. Kasif, and S. Salzberg, "A system for induction of oblique decision trees," *Journal of Artificial Intelligence Research (JAIR)*, vol. 2, pp. 1–32, 1994.

[171] J. Weng and W.-S. Hwang, "An incremental learning algorithm with automatically derived discriminating features," in *Proceedings of the Asian Conference on Computer Vision*, (Taipei, Taiwan), pp. 426–431, January 2000.

[172] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees.* Chapman and Hall, Boca Raton, Florida, USA, 1984.

[173] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[174] W. L. Buntine, "Decision tree induction systems: A Bayesian analysis," in *Proceedings of the Third Conference on Uncertainty in Artificial Intelligence*, (Seattle, Washington, USA), pp. 109–128, July 1987.

[175] S. K. Murthy, "Automatic construction of decision trees from data: A multi-disciplinary survey," *Data Mining and Knowledge Discovery*, vol. 2, no. 4, pp. 345–389, 1998.

[176] C. C. Aggarwal, C. Procopiuc, J. L. Wolf, P. S. Yu, and J. S. Park, "Fast algorithms for projected clustering," in *Proceedings of the ACM SIGMOD Conference on Management of Data*, (Philadelphia, Pennsylvania, USA), pp. 61–72, June 1999.

[177] S. Kumar, J. Ghosh, and M. M. Crawford, "Hierarchical fusion of multiple classifiers for hyperspectral data analysis," *Pattern Analysis and Applications*, vol. 5, pp. 210–220, 2002.

[178] J. Tantrum, A. Murua, and W. Stuetzle, "Assessment and pruning of hierarchical model based clustering," in *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (Washington, D.C., USA), August 2003.

[179] J. Tantrum, A. Murua, and W. Stuetzle, "Hierarchical model-based clustering of large datasets through fractionation and refractionation," in *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (Edmonton, Alberta, Canada), pp. 183–190, July 2002.

[180] H. Yu, J. Yang, and J. Han, "Classifying large data sets using SVMs with hierarchical clusters," in *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (Washington, D.C., USA), pp. 306–315, August 2003.

[181] A. Kalton, P. Langley, K. Wagstaff, and J. Yoo, "Generalized clustering, supervised learning, and data assignment," in *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (San Francisco, California, USA), pp. 299–304, August 2001.

[182] J. Kivinen, A. J. Smola, and R. C. Williamson, "Online learning with kernels," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, (Cambridge, Massachusetts, USA), 2002.

[183] K. Crammer, J. Kandola, and Y. Singer, "Online classification on a budget," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, (Cambridge, Massachusetts, USA), 2003.

[184] C. Gupta and R. Grossman, "GenIc: A single pass generalized incremental algorithm for clustering," in *Proceedings of the SIAM International Conference on Data Mining*, (Lake Buena Vista, Florida, USA), April 2004.

[185] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, (Athens, Greece), pp. 426–435, August 1997.

[186] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An efficient data clustering method for very large databases," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, (Montreal, Quebec, Canada), pp. 103–104, June 1996.

[187] M. M. Breunig, H.-P. Kriegal, P. Kröger, and J. Sander, "Data Bubbles: Quality preserving performance boosting for hierarchical clustering," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, (Santa Barbara, California, USA), May 2001.

[188] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic, internet services," in *Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track)*, (Washington, D.C., USA), 2002.

[189] P. Geurts, I. E. Khayat, and G. Leduc, "A machine learning approach to improve congestion control over wireless computer networks," in *Proceedings of the 4th IEEE Conference on Data Mining (ICDM'04)*, (Brighton, United Kingdom), pp. 383–386, November 2004.

[190] R. Amit and M. Matarić, "Learning movement sequences from demonstration," in *Proceedings of the Second International Conference on Development and Learning*, (Boston, Massachusetts, USA), pp. 165–171, June 2002.

[191] P. Ge and P. K. McKinley, "Leader-driven multicast for video streaming on wireless LANs," in *Proceedings of the IEEE International Conference on Networking*, (Atlanta, Georgia, USA), August 2002.

[192] P. K. McKinley, C. Tang, and A. P. Mani, "A study of adaptive forward error correction for wireless collaborative computing," *IEEE Transactions on Parallel and Distributed Systems*, September 2002.

[193] P. K. McKinley, U. I. Padmanabhan, N. Ancha, and S. M. Sadjadi, "Composable proxy services to support collaboration on the mobile internet," *IEEE Transactions on Computers (Special Issue on Wireless Internet)*, pp. 713–726, June 2003.

[194] Z. Zhou, P. K. McKinley, and S. M. Sadjadi, "On quality-of-service and energy consumption tradeoffs in FEC-enabled audio streaming," in *Proceedings of the*

12th IEEE International Workshop on Quality of Service (IWQoS), (Montreal, Canada), June 2004. Selected as best student paper.

[195] "Computational Ecology and Visualization Laboratory (CEVL)." `http://www.cevl.msu.edu`, November 2006.

[196] "Kellogg Biological Research Station (KBS)." `http://www.kbs.msu.edu`, November 2006.

[197] "North American Breeding Bird Survey." `http://www.pwrc.usgs.gov/bbs/`, December 2006.

[198] "Christmas Bird Count." `http://www.audubon.org/bird/cbc/`, December 2006.

[199] "UK Breeding Bird Survey." `http://www.bto.org/bbs/`, December 2006.

[200] "Crossbow Technology, Inc.." `http://www.xbow.com`, November 2006.

[201] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System architecture directions for networked sensors," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, (Cambridge, Massachusetts, USA), November 2000.

[202] W. H. Thorpe, "The learning of song patterns by birds, with especial reference to the song of the chaffinch, Fingilla coelebs," *Ibis: The international journal of avian science*, vol. 100, pp. 535–570, 1958.

[203] W. H. Thorpe, *Bird Song: The biology of vocal communication and expression in birds.* New York, New York, USA: Cambridge University Press, 1961.

[204] O. Tchernichovski, T. Lints, S. Deregnaucourt, and P. Mitra, "Analysis of the entire song development: Methods and rationale," *Annals of the New York Academy of Science*, vol. 1016, pp. 348–363, 2004. special issue: Neurobiology of Birdsongs.

[205] C. Catchpole and P. Slater, *Bird Song: Biological themes and variations.* New York, New York, USA: Cambridge University Press, 1995.

[206] E. A. Brenowitz, D. Margoliash, and K. W. Nordeen, "An introduction to birdsong and the avian song system," *Journal of Neurobiology*, vol. 33, pp. 495–500, 1997.

[207] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra, "Dimensionality reduction for fast similarity search in large time series databases," *Knowledge and Information Systems*, vol. 3, no. 3, pp. 263–286, 2000.

[208] B.-K. Yi and C. Faloutsos, "Fast time sequence indexing for arbitrary Lp norms," in *Proceedings of the 26th International Conference on Very Large Databases*, (Cairo, Egypt), September 2000.

[209] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, April 1965.

[210] K.-P. Li and J. E. Porter, "Normalizations and selection of speech segments for speaker recognition scoring," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 1, pp. 595–598, April 1988.

[211] E. Keogh, J. Lin, and W. Truppel, "Clustering of time series subsequences is meaningless: Implications for past and future research," in *Proceedings of the 3rd IEEE International Conference on Data Mining*, (Melbourne, Florida, USA), November 2003.

[212] B.-K. Yi, H. Jagadish, and C. Faloutsos, "Efficient retrieval of similar time sequences under time warping," in *Proceedings of the IEEE International Conference on Data Engineering*, (Orlando, Florida, USA), pp. 201–208, February 1998.

[213] B. Chiu, E. Keogh, and S. Lonardi, "Probabilistic discovery of time series motifs," in *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (Washington, D.C., USA), pp. 493–498, August 2003.

[214] J. Lin, E. Keogh, S. Lonardi, and P. Patel, "Finding motifs in time series," in *Proceedings of the 2nd Workshop on Temporal Data Mining, at the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (Edmonton, Alberta, Canada), July 2002.

[215] G. Tandon, P. Chan, and D. Mitra, "MORPHEUS: Motif oriented representations to purge hostile events from unlabeled sequences," in *Proceedings of the Workshop on Visualization and Data Mining for Computer Security (Viz/DMSEC) held in conjunction with the 11th ACM Conference on Computer and Communications Security (CCS)*, (Washington, DC, USA), pp. 16–25, October 2004.

[216] E. Keogh, J. Lin, and A. Fu, "HOT SAX: Finding the most unusual time series subsequence," in *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005)*, (Houston, Texas, USA), pp. 226–233, November 2005.

[217] F. Provost and R. Kohavi, "Glossary of terms," *Machine Learning*, vol. 30, pp. 271–274, February 1998.

[218] R. L. Smith, "Acoustic signatures of birds, bats, bells and bearings," in *Proceedings of the Annual Vibration Institute Meeting*, (Dearborn, Michigan, USA), June 1998.

[219] J. P. Eagn, *Signal Detection Theory and ROC Analysis.* Series in Cognitition and Perception, New York, New York, USA: Academic Press, 1975.

[220] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[221] F. J. Provost and T. Fawcett, "Robust classification for imprecise environments," *Machine Learning*, vol. 42, no. 3, pp. 203–231, 2001.

[222] T. S. Group, "STREAM: The stanford stream data manager," *IEEE Data Engineering Bulletin*, vol. 26, September 2003.

[223] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the 21st ACM Symposium on Principles of Database Systems (PODS)*, (Madison, Wisconsin, USA), June 2002.

[224] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (Dallas, Texas, USA), May 2000.

[225] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous dataflow processing for an uncertain world," in *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, (Asilomar, California, USA), January 2003.

[226] P. Bizarro, S. Babu, D. DeWitt, and J. Widom, "Content-based routing: Different plans for different data," in *Proceedings of the Thirty-First International Conference on Very Large Data Bases*, (Trondheim, Norway), pp. 757–768, September 2005.

[227] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *Proceedings of KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, (Seattle, Washington, USA), pp. 359–370, July 1994.

[228] J. Beringer and E. Hullermeier, "Online clustering of parallel data streams," *Data and Knowledge Engineering*, vol. 58, no. 6, pp. 180–204, 2006.

[229] F. Chu, Y. Wang, and C. Zaniolo, "An adaptive learning approach for noisy data streams," in *Proceedings Forth IEEE Conference on Data Mining (ICDM'04)*, (Brighton, United Kingdom), pp. 351–354, November 2004.

[230] J. Yang and W. Wang, "AGILE: A general approach to detect transitions in evolving data streams," in *Proceedings of the 4th IEEE Conference on Data Mining (ICDM'04)*, (Brighton, United Kingdom), pp. 559–562, November 2004.

[231] D. Ron, Y. Singer, and N. Tishby, "The power of amnesia: Learning probabilistic automata with variable memory length," *Machine Learning*, vol. 25, no. 2–3, pp. 117–149, 2004.

[232] C. Partridge, D. Cousins, A. W. Jackson, rajesh Krishnan, T. Saxena, and W. T. Strayer, "Using signal processing to analyze wireless data traffic," in *Proceedings International Conference on Mobile Computing and Networking, ACM workshop on Wireless security*, (Atalanta, Georgia, USA), pp. 67–76, 2002.

[233] Y. Ke, D. Hoiem, and R. Sukthankar, "Computer vision for music identification," in *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition*, (San Diego, California, US), June 2005.

[234] D. K. Mellinger and C. W. Clark, "Recognizing transient low-frequency whale sounds by spectrogram correlation," *Journal of the Acoustical Society of America*, vol. 107, pp. 3518–3529, June 2000.

[235] S. Fagerlund and A. Härmä, "Parameterization of inharmonic bird sounds for automatic recognition," in *13th European Signal Processing Conference (EUSIPCO)*, (Antalya, Turkey), September 2005.

[236] P. Somervuo and A. Härmä, "Bird song recognition based on syllable pair histograms," in *Proceedings of the IEEE International Conference on Accoustics, Speech and Signal Processing (ICASSP)*, (Montreal, Quebec, Canada), May 2004.

[237] J. A. Kogan and D. Margoliash, "Automated recognition of bird song elements from continuous recordings using dynamic time warping and hidden markov models: A comparative study," *Journal of the Acoustical Society of America*, vol. 103, no. 4, pp. 2185–2196, 1998.

[238] M. Ghiassi, H. Saidane, and D. Zimbra, "A dynamic artificial neural network model for forecasting time series events," *International Journal of Forecasting*, vol. 21, pp. 341–362, April-June 2005.

[239] "Time Series Prediction: Forecasting the Future and Understanding the past: Proceedings of the NATO Advanced Research Workshop on Comparative Time Series Analysis," May 1992.

[240] P. J. Brockwell and R. A. Davis, *Introduction to Time Series and Forecasting*. New York, New York, USA: Springer-Verlag, 1996.

[241] N. A. Gershenfeld and A. S. Weigend, "The future of time series: Learning and understanding," in *Time Series Prediction: Forecasting the Future and Understanding the past: Proceedings of the NATO Advanced Research Workshop on Comparative Time Series Analysis*, (Santo Fe, New Mexico, USA), May 1992.

[242] H. Nyquist, "Certain topics in telegraph transmission theory," *Transactions of the American Institute of Electrical Engineers (AIEE)*, vol. 47, pp. 617–644, April 1928.

[243] H. Nyquist, "Certain topics in telegraph transmission theory," *Proceedings of the IEEE*, vol. 90, pp. 280–305, February 2002. Reprint of Nyquist's 1928 classic paper.

[244] C. E. Shannon, "Communication in the presence of noise," *Proceedings of the Institute of Radio Engineers (IRE)*, vol. 37, pp. 10–21, January 1949.

[245] C. E. Shannon, "Communication in the presence of noise," *Proceedings of the IEEE*, vol. 86, pp. 447–457, February 1998. Reprint of Shannon's 1949 classic paper.

[246] G. T. Nguyen, R. H. Katz, B. Noble, and M. Satyanarayanan, "A trace-based approach for modeling wireless channel behavior," in *Proceedings of the Winter Simulation Conference*, (Coronado, California, USA), pp. 597–604, December 1996.

[247] W. Jiang and H. Schulzrinne, "Modeling of packet loss and delay and their effect on real-time multimedia service quality," in *Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, (Chapel Hill, North Carolina, USA), June 2000.

[248] H. Sanneck and G. Carle, "A framework model for packet loss metrics based on loss runlengths," in *Proceedings of the SPIE/ACM SIGMM Multimedia Computing and Networking Conference (MMCN)*, (San Jose, California, USA), pp. 177–187, January 2000.

[249] E. N. Gilbert, "Capacity of a burst-noise channel," vol. 39, pp. 1253–1265, September 1960.

[250] E. O. Elliot, "Estimates of error rates for code on burst-noise channels," vol. 42, pp. 1977–1997, September 1963.

[251] C. W. J. Granger, "Forecasting in economics," in *Time Series Prediction: Forecasting the Future and Understanding the past: Proceedings of the NATO Advanced Research Workshop on Comparative Time Series Analysis*, (Santo Fe, New Mexico, USA), May 1992.

[252] A. F. Atiya, S. M. El-Shoura, S. I. Shaheen, and M. S. El-Sherif, "A comparison between neural-network forecasting techniques – case study: River flow forecasting," *IEEE Transactions on Neural Networks*, vol. 10, pp. 402–409, March 1999.

[253] W. Cui and M. A. Bassiouni, "Virtual private network bandwidth management with traffic prediction," *Computer Networks*, vol. 42, pp. 765–778, 2003.

[254] J. Ilow, "Forecasting network traffic using FARIMA models with heavy tailed innovations," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 6, pp. 3814–3817, June 2000.

[255] B. P. Bogert, M. J. R. Healy, and J. W. Tukey, "The quefrency alanysis of time series for echoes: Cepstrum, pseudo-autocovariance, cross-cepstrum, and saphe cracking," in *Proceedings of the Symposium on Time Series Analysis (Brown University, June 1962)*, (New York, New York, USA), pp. 209–243, John Wiley and Sons, 1963.

[256] M. Papadopouli, H. Shen, E. Raftopoulos, M. Ploumidis, and F. Hernandez-Campos, "Short-term traffic forecasting in a campus-wide wireless network," in *Proceedings of the 16th International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC)*, (Berlin, Germany), September 2005.

[257] M. C. Mozer, "Neural net architectures for temporal sequence," in *Time Series Prediction: Forecasting the Future and Understanding the past: Proceedings of the NATO Advanced Research Workshop on Comparative Time Series Analysis*, (Santo Fe, New Mexico, USA), May 1992.

[258] A. M. Fraser and A. Dimitriadis, "Forecasting probability densities using hidden markov models with mixed states," in *Time Series Prediction: Forecasting the Future and Understanding the past: Proceedings of the NATO Advanced Research Workshop on Comparative Time Series Analysis*, (Santo Fe, New Mexico, USA), May 1992.

[259] J. Zhang, B. H. C. Cheng, Z. Yang, and P. K. McKinley, "Enabling safe dynamic component-based software adaptation," in *Architecting Dependable Systems III, Springer Lecture Notes for Computer Science* (A. R. Rogerio de Lemos, Cristina Gacek, ed.), Springer-Verlag, 2005.

[260] J. R. Koza, "Hierarchical genetic algorithms operating on populations of computer programs," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence* (N. Sridharan, ed.), (San Mateo, California, USA), pp. 768–774, Morgan Kaufmann, 1989.

[261] R. Friedberg, "A learning machine: Part I," *IBM J. Research and Development*, vol. 2, no. 1, pp. 2–13, 1958.

[262] C. Ofria, C. Adami, and T. C. Collier, "Design of evolvable computer languages," *IEEE Transactions in Evolutionary Computation*, vol. 6, pp. 420–424, August 2002.