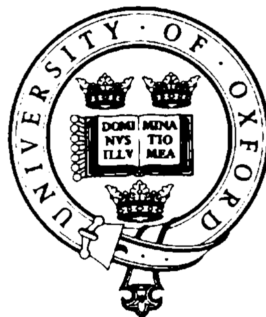
Oxford University
Department of Engineering Science
Robotics Research Group

A Self Adaptive Architecture for Image Understanding



Paul Robertson

A Thesis submitted for the Degree of Doctor of Philosophy of the University of Oxford

July, 2001

- 7 FEB 2002

This thesis is submitted to the Department of Engineering Science, University of Oxford, in partial fulfillment of the requirements for the degree of Doctor of Philosophy. The thesis is entirely my own work, and except where otherwise stated, describes my own research.

Paul Robertson, Wolfson College, Oxford

Copyright © 2001, Paul Robertson
All Rights Reserved

Paul Robertson
Wolfson College

Doctor of Philosophy
Trinity Term 2001

A Self-Adaptive Architecture for Image Understanding

Abstract

This thesis develops a self-adaptive architecture for image understanding that addresses certain kinds of lack of robustness common in image understanding programs. The architecture provides support for making image understanding programs that can manipulate their own semantics and thereby adjust their structure in response to changes in the environment that might cause static image understanding systems to fail.

The general approach taken has been to explore the ideas of self-adaptive software and implement an architectural framework that addresses a class of problems that we term "interpretation problems" common in image understanding. Self-adaptive software is a relatively new idea and this thesis represents one of the first implementations of the general idea. The general idea is that to make programs robust to changing environmental conditions that they should be "aware" of their relationship with the environment and be able to restructure themselves at runtime in order to "track" changes in the environment.

The implementation takes the form of a multi-layered reflective interpreter that manipulates and runs simple agents. The interpreter framework utilizes Monte-Carlo sampling as a mechanism for estimating most likely solutions, uses Minimum Descriptin Length (MDL) as a central coordinating device, and includes a theorem prover based compiler to restructure the program when necessary.

To test the architectural ideas developed in the thesis a test domain of interpreting aerial images was chosen. Much of the research described in the thesis addresses issues in that problem domain. The task of the program is to segment, label, and parse aerial images so as to produce an image description similar to descriptions produced by a human expert. An image corpus is developed that is used as the source of domain knowledge.

The first processing stage of the program segments the aerial images into segments similar to those found in the annotated corpus. To accomplish this a new segmentation algorithm that we call semantic segmentation was developed that not only used MDL as a principle to drive the low-level segmentation but also allows higher level semantics to influence the segmentation. In our usage of the algorithm those semantics take the form of labeling and parsing the resulting segmentation.

The second stage labels the regions and parses the regions into a parse tree. To do this we develop a 2D statistical parser. Rules of grammar are induced from the corpus and an MDL parser finds approximations to the most probable parse of the regions of the segmented image.

Contents

Abstract ii

Table of Contents 1

List of Figures 5

Acknowledgements 8

1 Introduction 9

1.1 Introduction . . . 9

1.1.1 Motivation . . . 9

1.2 The Problem Domain . . . 11

1.3 Logical Components of the System . . . 14

1.3.1 Image Segmentation and Labeling Program . . . 14

1.3.2 Communication Model . . . 16

1.3.3 MDL Agent Architecture . . . 17

1.3.4 Semantic Segmentation . . . 17

1.3.5 Patchwork Parser . . . 19

1.3.6 Statistical/MDL Theorem Prover . . . 19

1.3.7 MDL clustering for model induction . . . 20

1.3.8 Self Adaptive Architecture . . . 21

1.4 Overview of the Thesis . . . 22

2 Architecture 26

2.1 Introduction . . . 26

2.2 Blackboards and Forward Chaining Rule Systems . . . 28

2.3 Subsumption . . . 30

2.4 Agents . . . 31

2.5 Schema . . . 32

2.6 Self Adaptive Software . . . 33

2.7 Overview of the Current State of Self Adaptive Software . . . 39

2.7.1 Systems that interpret their environment through sensors . . . 40

2.8 Conclusion . . . 41

3 Architecture for Interpretation Problems 42

3.1 Introduction . . . 42

3.2 Contributing Ideas . . . 43

3.2.1 Communication . . . 43

3.2.2 Minimum Description Length . . . 46

3.2.3 Monte-Carlo methods . . . 47

3.2.4 Agent Selection Paradigms 50
3.3 MDL Agent Architecture 51
3.3.1 Interpretation Problems 51
3.3.2 Objects in the GRAVA Architecture 53
3.4 An Illustrative Example of MDL Agents 61
3.4.1 Recognizing a Hand Written Phrase 63
3.4.2 Results 67
3.5 Conclusion 69

4 Semantic Segmentation **72**
4.1 Introduction 72
4.2 Prior Work 76
4.2.1 Constructing stable descriptions 77
4.2.2 Region Competition 79
4.2.3 Shortcomings of Low-Level Methods 80
4.3 Semantic Segmentation 81
4.3.1 The Semantic Segmentation Base Algorithm 82
4.3.2 Base Level Segmentation 85
4.3.3 Changes in Region Topology 89
4.3.4 Color and Multi-plane Segmentation 90
4.3.5 MDL Agent Implementation 91
4.3.6 Adding Semantics 94
4.4 Conclusion 100

5 Patchwork Parsing **102**
5.1 Introduction 102
5.2 Prior Work 104
5.2.1 Speech Recognition 106
5.2.2 Natural Language 107
5.3 Statistical Patchwork Parsing 112
5.3.1 Defining the Image Labeling Problem 112
5.3.2 Optical Model 113
5.3.3 Picture Language Model 114
5.3.4 Parsing an image using rules extracted from an image bank 126
5.4 Algorithms for Patchwork Parsing 127
5.5 Conclusion 136

6 Interpreting & Clustering the Corpora **139**
6.1 Introduction 139
6.1.1 The Idea of Contexts 140
6.2 Prior Work 144
6.3 Principal Component Decomposition 147
6.4 Interpreting an Image Corpus as Optical Models 158
6.5 Learning Contexts from Images 161
6.5.1 Summary of the Description of the Corpus 167
6.6 Conclusion 168

7 Reflection and Self-Adaptation **170**
7.1 Introduction . . . 170
7.1.1 Interpretation Problems . . . 171
7.1.2 Overview of the Chapter . . . 173
7.2 Prior Work . . . 173
7.2.1 Reflection . . . 173
7.3 Reflective Interpreter for Self-Adaptation . . . 180
7.4 Program Synthesis . . . 185
7.4.1 Compilation as Proof . . . 185
7.5 Uncertain Information and MDL . . . 189
7.5.1 Protocol for Interpreters . . . 190
7.5.2 Protocol for Agents . . . 191
7.6 Fidelity and Quality . . . 192
7.6.1 Quality . . . 193
7.7 Agent Training and Fidelity . . . 195
7.7.1 MDL statistical theorem prover for code generation . . . 200
7.7.2 Disjunction and Teams . . . 202
7.8 Conclusion . . . 202

8 Self-Adaptive Aerial Image Interpretation **204**
8.1 Introduction . . . 204
8.1.1 Overview of the Chapter . . . 205
8.2 Aerial Interpretation Program Design . . . 205
8.2.1 Reflective Layers . . . 205
8.2.2 Optical and Labeling Contexts . . . 207
8.2.3 Models of World State . . . 207
8.2.4 Generating Code from Contexts . . . 208
8.3 Results . . . 210
8.3.1 Test Interpreted Image . . . 210
8.4 Conclusion . . . 213

9 Conclusions and Further Work **215**
9.1 Introduction . . . 215
9.1.1 Overview of the Chapter . . . 216
9.2 What We Have Learned . . . 217
9.2.1 Usefulness of Interpretation Problems . . . 217
9.2.2 Benefits of the GRAVA architecture . . . 217
9.2.3 Reuse of Ideas from Computational Linguistics . . . 218
9.2.4 Benefits of MDL . . . 218
9.3 Further Work . . . 218
9.3.1 Resource Contention Real-Time Constraints, and Utility . . . 218
9.3.2 Unsupervised Learning . . . 219
9.3.3 Picture Language for 3D . . . 220
9.3.4 Other Ways of Estimating Description Length . . . 221
9.3.5 Image Reconstruction . . . 221
9.3.6 Stability and Predictable Behavior . . . 221
9.4 Looking Forward . . . 223
9.4.1 A New Kind of Computation . . . 223
9.4.2 An Architecture for Intelligence . . . 224

A Corpora **226**
A.1 SPOT Satellite Images . . . 226
A.2 MASS GIS Images . . . 227

B The GRAVA software architecture **231**
B.1 Introduction . . . 231
B.2 Yolambda . . . 232
B.3 Image Processing Kernel . . . 232
B.4 Matlab interface . . . 232
B.4.1 Matlab Demo in Yolambda . . . 234
B.4.2 MATLAB Engine and MX functions . . . 235

C Annotation Facility **238**
C.1 Introduction . . . 238
C.2 Using the annotation facility . . . 239
C.3 Annotating Regions . . . 239
C.3.1 Rectangular Regions . . . 239
C.3.2 Polygonal Regions . . . 240
C.3.3 Live Wire Regions . . . 240
C.4 Region editing . . . 240
C.5 Menu Commands . . . 241
C.5.1 File . . . 241
C.5.2 Edit . . . 241
C.5.3 Annotate . . . 242
C.5.4 View . . . 243
C.5.5 Zoom . . . 243
C.5.6 Option . . . 244
C.5.7 Keyboard Commands . . . 245

D Line Vectorization **246**
D.1 Introduction . . . 246

E Yolambda and Lisp notation **252**
E.1 Introduction . . . 252
E.2 Expressions . . . 252
E.3 Procedure Calls . . . 253
E.4 Lambda Expressions . . . 254
E.5 Literal Expressions . . . 255
E.6 Conditionals . . . 255
E.7 Assignment . . . 257
E.8 Definitions . . . 257
E.9 Binding Constructs . . . 258
E.10 Combining Form . . . 260
E.11 Quasiquotation . . . 260

List of Figures

1.1 Segmented Aerial Image . . . 12
1.2 Marr's two leaves . . . 13
1.3 Logical Components . . . 15
1.4 Communication Model . . . 16
1.5 Segmented Image . . . 18
1.6 Segmenting Kanizza's Triangle . . . 18
1.7 Description for River Using Red and Blue . . . 21
1.8 Hand Written Word Recognizer . . . 24

2.1 Hearsay II architecture . . . 29
2.2 Path of a Self Adaptive Program . . . 34
2.3 The Self Adaptive Loop . . . 35

3.1 Communication of a message through a channel . . . 44
3.2 Ambiguous Visual Interpretations . . . 51
3.3 Objects in the GRAVA Architecture . . . 54
3.4 Interactive activation model for word perception . . . 62
3.5 Nursery Rhyme Test Data . . . 63
3.6 Hand Written Phrase Recognizer Program . . . 65
3.7 Convergence . . . 68

4.1 Marr's two leaves . . . 75
4.2 High level schematic of the semantic segmentation algorithm . . . 84
4.3 Base Description Language for Images . . . 85
4.4 Representation of Regions . . . 87
4.5 Disk for Local Piecewise Smooth Region . . . 88
4.6 Region Splitting . . . 89
4.7 Region Merging . . . 90
4.8 Color Segmented . . . 91
4.9 Models for the Base Segmenter . . . 92
4.10 Segmenting an Image with a Lake . . . 95
4.11 Segmentation of Subjective Contours . . . 100

5.1 Patches without context are hard to identify . . . 103
5.2 Example PCFG grammar . . . 110
5.3 Region with internal and external boundaries . . . 115
5.4 A structure region . . . 117
5.5 Example of a Structured Non-Terminal . . . 117
5.6 Partially Obscured Region . . . 118
5.7 A segmented image . . . 120
5.8 Grammar Induction from an Annotated Image . . . 121

5.9 Smoothing Lattice for (field road swamp river) . . . 125
5.10 Expansion for nodes with wildcards . . . 126
5.11 An example parse . . . 127
5.12 Interpret Segmented Image as a Parse Tree . . . 133
5.13 Monte-Carlo Parse Convergence . . . 135

6.1 Communication Model . . . 140
6.2 Path of a Self Adaptive Program . . . 141
6.3 Image Contexts . . . 143
6.4 Maja Mataric's Imitative Model . . . 146
6.5 The Need for Decomposition . . . 148
6.6 Representation of a Point in a Collection . . . 151
6.7 Dividing the Data Points to Reduce Description Length . . . 152
6.8 Accidentally Severed Points . . . 155
6.9 Merging Collections . . . 156
6.10 Example Result of Principal Component Decomposition . . . 156
6.11 Intertwined Non-Convex Shapes . . . 157
6.12 Curved example data . . . 158
6.13 Final Decomposition of Example Non-Convex Data . . . 159
6.14 Description for River Using Red and Blue . . . 160
6.15 PDF for Label Pixels Contexts . . . 164
6.16 The 4 Images of Label Pixels Context 7 . . . 166
6.17 PDF for Optical Contexts . . . 167

7.1 Example of the relationship between levels of interpretation . . . 172
7.2 Reflective Tower of Interpreters . . . 175
7.3 Runtime MOP Structures . . . 177
7.4 Meta-Knowledge and Compilation . . . 182
7.5 Agent and Model for Assignment . . . 187
7.6 Agent and Model for Addition . . . 187
7.7 Agent and Model for Dereference . . . 188
7.8 Example Proof Tree for Compilation Example . . . 188
7.9 Protocol for Interpreter Meta-Information . . . 190
7.10 Protocol for Agent Meta-Information . . . 191
7.11 Fidelity . . . 192
7.12 Division of a Kind into Separate Beliefs . . . 195
7.13 Data Structures for Computing Conditional Probabilities . . . 196
7.14 Fidelity Results for Version 1, 2 and 3 Tools . . . 199
7.15 Fidelity Convergence . . . 199

8.1 Conceptual Schematic of the Self-Adaptive Loop . . . 204
8.2 Reflective Layers of the Domain Example . . . 206
8.3 Test Run of Image Interpretation . . . 211
8.4 Test Run 2 of Image Interpretation . . . 212

A.1 Portion of image showing detail of a region annotated as "suburban" . . . 228
A.2 Area covered by MassGIS ortho-photos . . . 229
A.3 Portion of a MassGIS ortho-photo image with annotations . . . 230

B.1 Schematic view of the Software Architecture . . . 231

C.1 The annotation facility 240

D.1 Converting pixel lists to vectors 246
D.2 The line filter's epsilon parameter 247
D.3 Estimating local curvature 248

Acknowledgements

I would like to thank my supervisor, Professor Brady for giving me the opportunity to do this work and for his continuous encouragement, support, and advice. My time at Oxford has been one of the most enjoyable periods in my life. This is almost entirely the result of Professor Brady's commitment to his students. Mike always has time to discuss ideas, read drafts and perhaps most important of all to provide encouragement during those inevitable "dry" periods.

I have benefited from many discussions with the people in the Robotics Research Group and the Medical Vision Lab at Oxford and at the MIT AI lab.

I would like to thank Timor Kadir for his help with some seriously hard Latex issues.

I would especially like to thank all those who helped to make Oxford a fun place to be including but not limited to: Steven Reece, Timor Kadir, Vicky Mortimer, Veit Schenk, Sebastien Gilles and all other members of the Terrapin Hut during my time at Oxford.

Professor Andrew Blake had a significant and beneficial impact on the direction of this research for which I am deeply grateful.

I would like to thank Robert Laddaga for his support as a friend and as a program manager in his role at DARPA. Bob's help has been invaluable. It was Bob who introduced me to the concept of "self-adaptive software" which went on to become the topic of my thesis. Bob has helped me through many crises for which I will be eternally grateful but I am most thankful for his friendship.

Carole J. Lee spent thousands of hours annotating the aerial images used in this work. I am grateful for her care and dedication in the exeution of such a laborious and mind numbing task.

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-98-0056. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

Chapter 1

Introduction

1.1 Introduction

Robust performance of image interpretation programs has been an elusive goal. Under constrained conditions, we are able to make image analysis programs work well but the variability of the unconstrained world remains an obstacle to robust vision programs. The real world is a complex place and building vision programs that have hardwired logic for dealing with every possibility seems naive. Dealing with the complexity of the natural visual world is as important a part of vision research as the focused research efforts into constrained image analysis problems. We have learned a great deal about algorithms for image analysis. Compared to what we have learned about image analysis, however, we know relatively little about how to architect programs that can deal effectively with the complexity and variability of the natural world.

This thesis explores a self-adaptive approach towards achieving reliable performance in complex and changing environments. Self-adaptive software is concerned with the problem of automatically adapting program *structure* in response environmental changes in order to provide robust performance under uncertain and changing conditions. This is in contrast to 'neural' and similar systems that adapt simply by modifying the values of a small number of numerical parameters; but which otherwise leave unchanged the structure of the program conceived by the designers.

We explore the self-adaptation of a program that segments and labels aerial images. The thesis develops algorithms for segmentation and image parsing for aerial image understanding. An architecture and algorithms for self adaptation are also developed. For self adaptation, we concentrate on the tasks of program synthesis, knowledge induction from an image corpus and the interaction between different levels of knowledge. This thesis is therefore a contribution to both Artificial Intelligence and to Computer Vision.

1.1.1 Motivation

The project began with the observation that a significant source of problems with interpreting visual scenes comes from our inability to precisely predict the nature of the environment in which the image understanding programs are expected to operate.

Even if we could know all the different states that the environment could be in, we wouldn't know *a priori* what state the environment would be in at any particular time. Consequently, we believe that in order to achieve robust performance image understanding, programs should determine the state of the environment at runtime and adapt to the environment that is found. In practice it is probably the case that the set of possible environment states cannot be explicitly enumerated *a priori*.

In the spring of 1998, the US agency DARPA introduced the term "Self adaptive software" to describe a software methodology that aspires to solve exactly the kind of problem described above. The DPhil was later funded under the DARPA "Automatic Software Composition" (ASC) program.

Self adaptation is a model-based approach to building robust systems. The environment, the program's goal, and the program's computational structure must be modeled. In principle, the idea is simple. The environment model and the program goal model support both continuous evaluation of the performance of the program. When program performance deteriorates, the program goal model and the computation model together support modification of the program structure. In this way, the program structure evolves as the environment changes, even radically, so that the components of the program are always well suited to the environment in which they are running. The conjecture is that robust performance will result from having all components operating within their effective range.

This thesis addresses for the first time the question: How can the self-adaptive approach described above be realized for computer vision in the form of a software architecture?

From the outset, it seemed likely that a reflective architecture would be effective. This is because a reflective architecture provides the mechanisms necessary to support two of the core problems of self adaptive software—a mechanism for reasoning about the state of the computational system; and a mechanism for making changes to it. Reflective architectures, as we discuss in Chapter 3, had recently been introduced into AI and some of their theoretical properties were attractive. However, few techniques for building reflective architectures were known at the outset of this research, and they were applied mostly for the purpose of building open (modifiable) systems rather than building systems that reasoned about their own behavior.

Most work on reflection concentrates on how to provide these two mechanisms in the domain of interest so that a programmer may make use of them. The problem of self-adaptive software goes a step further. Not only must we have mechanisms that support introspection and change of the systems semantics, we must also have an implementation of the *user* of these mechanisms. That is to say that we must implement the programmer!

Automatic programming has been a goal of AI for decades. In the near term, however,

there continues to be little likelihood of solving the problem in any kind of general way. In constrained forms, however, it is possible to address the problem. Systems that generate code from specifications or architecture diagrams are examples of such solutions to restricted problems.

We decided to explore the conjecture that the class of programs that *interpret* their environment, such as vision and speech understanding, were constraining in ways that would permit a viable self adaptive architecture to be developed. An interpretation program is a program that attempts to build a description that is an interpretation of its input.

Interpretation problems often exist at meta-levels too, so, for example, in order to *interpret* the world as a visual scene, it is first necessary to *interpret* the program goal as a program. The interpretation of the program goal as a program is not a separate problem but is intimately intertwined with the problem of interpreting the visual world as a description. When a problem occurs with interpreting the visual world as a description it occurs while executing part of the program. That program was itself an interpretation of the program goal. The tower of such interpretation problems forms the basis for the reflective architecture presented here.

Generating image descriptions from visual scenes and generating programs from program goal descriptions are similar in that both involve generating structural descriptions and both are usually ambiguous. There are usually many different possible interpretations of a visual scene—some are more likely than others. Likewise, there are usually many ways of configuring a program to satisfy the program goal description—some are more likely than others to be effective in the actual visual environment that the program runs in.

1.2 The Problem Domain

We chose the problem of producing robust interpretations of aerial images taken from satellite and from high altitude planes as the test domain for our investigation into self-adaptive software.

Figure 1.1 shows an aerial image that has been segmented into regions. Each region has a content designation, such as; *lake*, *residential*, and so on. The description that is formed by the collection of labeled regions comprises an interpretation of the image.

Typically, computer vision researchers working on segmentation present their segmentation results, and measure the success of their program, by looking at the image and seeing how well their algorithm performed. We consider segmentation to be a function not only of the input image but also of the use that the program wants to make of the segmentation. A segmentation is a step in the larger problem of interpreting the

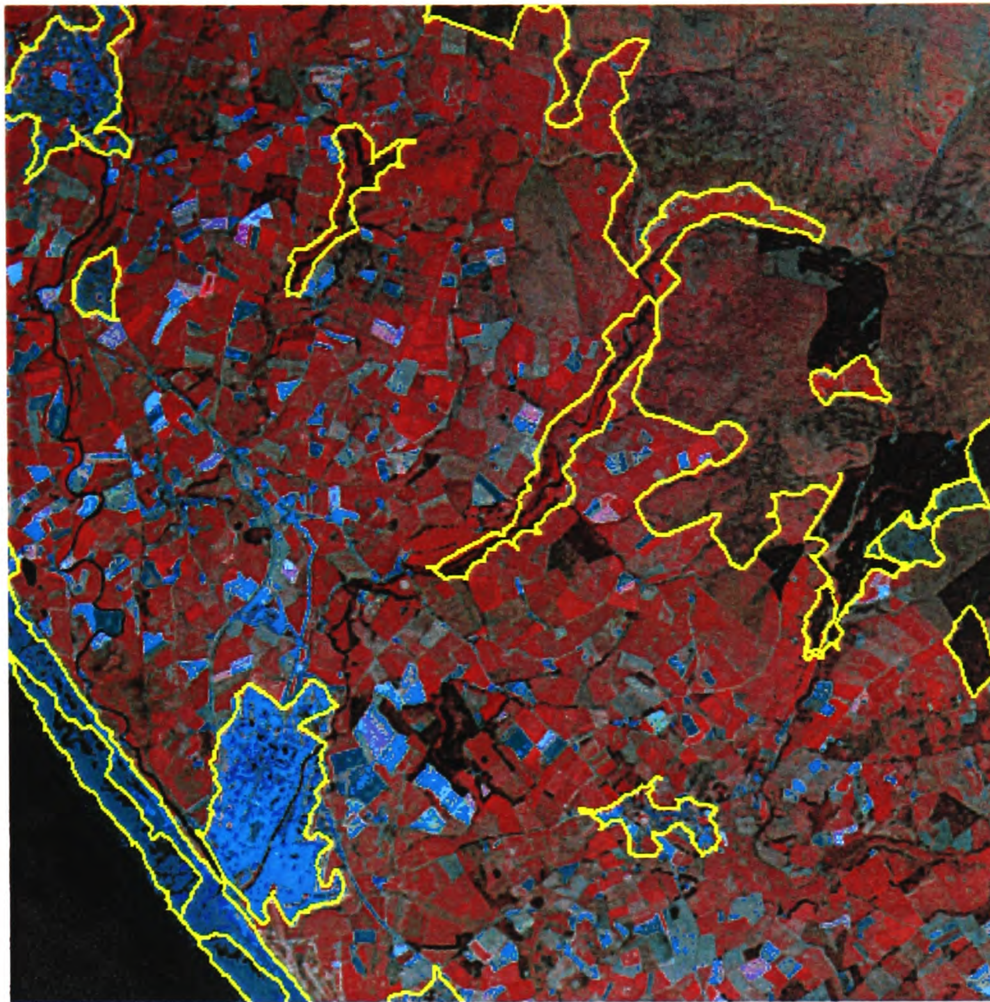


Figure 1.1: Segmented Aerial Image

image. A good segmentation is one that best suits the consumer of the segmentation in achieving the interpretation function that it is performing. The image should not necessarily always be segmented the same way. If the interpretation goal changes, the nature of the segmentation may need to change also. We take the view that segmentation should be influenced by semantics and that interpretation of the image and segmentation of the image should be cooperative processes. Instead of trying to build a good segmentation engine, we seek to find an interaction loop involving the image environment, the segmentation algorithm, and the image interpretation function. This interaction loop with the environment is an approach that has been successful in the AI intelligent agent community (Maes 1990b) as a way of achieving robust behavior.

Our approach to segmentation is profoundly different to that of most research on the topic. Most work in image segmentation takes the view that it is a low-level vision concept and attempts to implement it as such. On the other hand Marr (see Figure 1.2) gives a graphic example of why segmentation is so hard by showing two overlapping leaves with virtually no discernable edge or texture shift between the two leaves.

Identifying the leaves allows us to see the subtle leaf edge that would defeat any low level segmentation. It would seem to be problematic to use segmentation to aid in object recognition if object recognition is required in order to do the segmentation! Marr suggests that what we call segmentation may be an illusion that is based on introspection

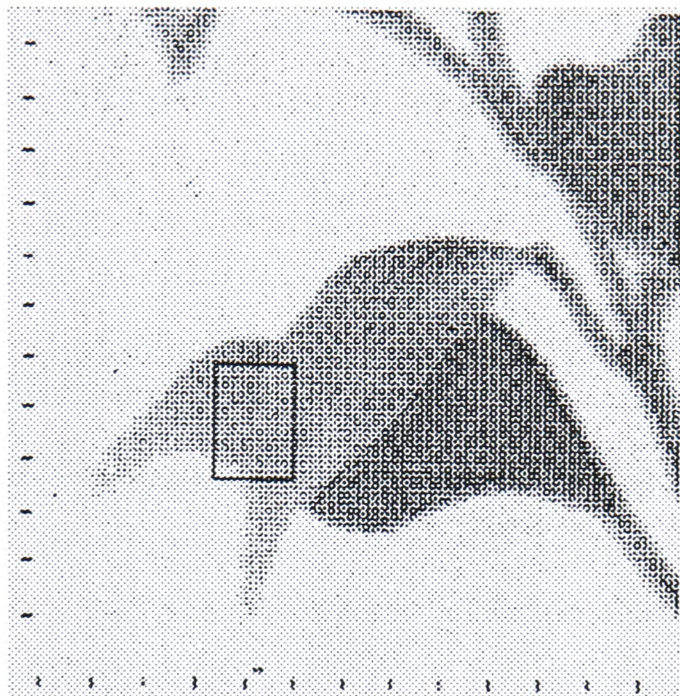


Figure 1.2: Marr's two leaves

of our own capabilities. He suggests that it is probably not a low-level phenomenon at all and so it is not discussed in his book on early vision.

In the system that we have built we have taken the position that segmentation and interpretation of the segmented image are cooperative processes. There are two ways in which the segmenter uses semantics in order to achieve good segmentations. We can think of these as inner loop and outer loop cooperation. The outer loop is the self adaptive loop.

Initially, little is known about the contents of the image or about the conditions under which the image was captured. Parameterizing the segmentation based on experience allows a rough cut segmentation to be generated. An attempt to interpret the segmented image enables something to be learned about the contents of the image and the conditions under which it was captured. If there are problems in interpreting the image based on the initial segmentation, the segmentation algorithm can be modified based on what has been learned about the image in the first iteration. The segmentation that results from such an improved segmenter should yield a better interpretation and more knowledge about the image contents and thus support synthesis of an even better segmenter. The cycle can continue until re-synthesis of the segmenter fails to yield a better interpretation.

The inner loop defines cooperation along the lines of a shared commitment to produce a compact image description. Again, the idea is straightforward. The image semantics provides a vocabulary for describing the segmentation. If the image semantics includes models of leaves, then the segmenter can use those models to describe the segmentation and thereby produce a more compact representation of the segmentation.

In the case of Marr's overlapping leaves, for example, the outer loop would deal with basic contextual issues including image quality, scene type and so on while the inner loop

would provide a database of shape models—in this case of leaves—to the segmentation shape vocabulary. Armed with such a vocabulary, it is easier to represent the image segments as overlapping leaves than as arbitrary regions. In this way, Marr's leaves could be successfully segmented. Of course, hidden in this overly simplified account is the notion of how one description of a segmentation is "easier", or "preferable". We discuss how this is done, within the minimum description length (MDL) framework, in chapter 4.

Many practical systems often achieve acceptable segmentations by hard-wiring domain knowledge. For example, when segmenting an ultrasonic heart image, a parameterized heart outline model may be fitted to the image in order to produce the segmentation. In certain situations we are able to exert sufficient control of the environment for this approach to work; but when the environment cannot be held constant, and this occurs surprisingly often, we need to be able to change the semantics that are used to *help* the segmentation process. The cooperative approach developed in this thesis allows semantics to change throughout the self adaptive process.

In order for the system to produce segmentations, we must provide a model of a *good segmentation.* In our test case, we chose to implement the *program goal* model as a learned model. An expert annotates images in a representative image corpus and the model of a correct segmentation is learned from the annotated corpus.

1.3 Logical Components of the System

We present a system called GRAVA (for Grounded Reflective Adaptive Vision Architecture) that segments and labels aerial images in a way that attempts to mimic the competence of a human expert.

Figure 1.3 shows the logical components of the system along with the supporting relationships between the parts. We now sketch the roles of these components.

1.3.1 Image Segmentation and Labeling Program

To produce an image interpretation, a variety of tools need to be brought into play. First, the image is processed by various tools in order to extract texture or feature information. The selection of the right tools determines ultimately how good the resulting interpretation will be. Next, a segmentation algorithm is employed in order to produce regions with outlines whose contents are homogeneous with respect to content as determined by the chosen texture and feature tools. The segmentation algorithm also depends upon tools that select seed points that initialize the segmentation. The choice of tools to initiate the segmentation determines what kind of segmentation will be produced.

Labeling the regions depends upon two processes. The first tries to determine possible

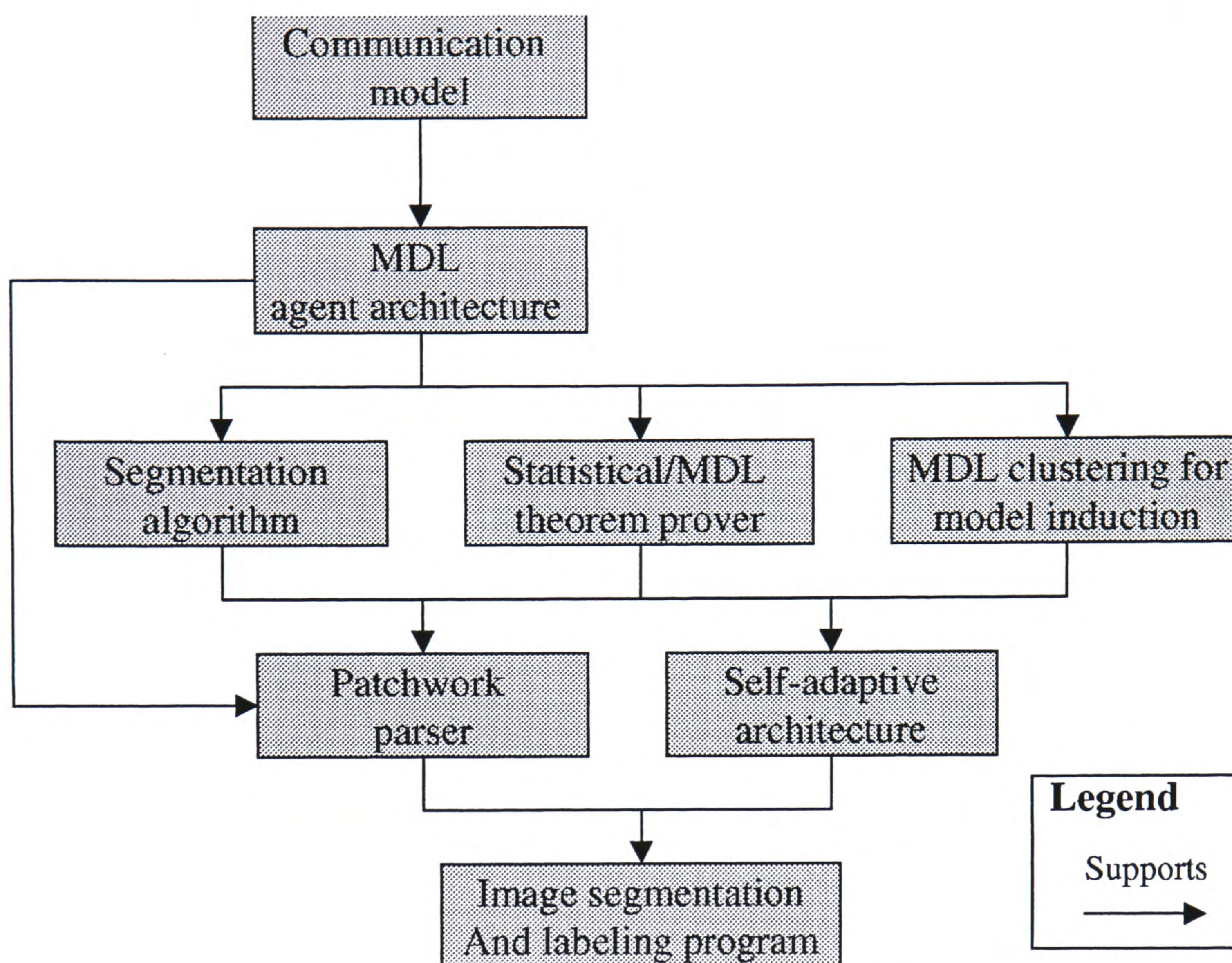


Figure 1.3: Logical Components

designations for the regions by analyzing the pixels within the regions. The second is a statistical parser that attempts to parse the image using a 2D grammar. Our application currently doesn't make use of the parse; but as we discuss in chapter 9 it could be used as the basis for further image interpretation. An important side effect for our application is that contextual information is mobilized by the parse process that enables good labels to be chosen for regions when there may be several ambiguous possibilities if only the pixels within the region are looked at.

At any point, a bad choice of tool—for initial feature extraction, seed point identification, region identification, or for contextual constraints—can lead to a poor image interpretation. The earlier the error occurs, the worse the resulting interpretation is likely to be. For example, a poor choice of tools for extracting textures from the image will result in a poor segmentation. The poor segmentation will result in poor region content analysis and so the resulting interpretation can be very bad indeed. The test for self-adaptive software is to determine how the program that consists of the collection of tools described above can be organized so that when a poor interpretation is produced, the program self adapts into a program that does a better job.

1.3.2 Communication Model

It is customary to think of speech recognition in communication theoretic terms. The speaker encodes the message as words. The message is transmitted across a noisy channel and the listener has the task of decoding the message. A similar view holds for vision, except that the encoding is performed by the reflectance of light from the scene. We are able to characterize vision in terms of communications systems and use this as the basis for the architecture.

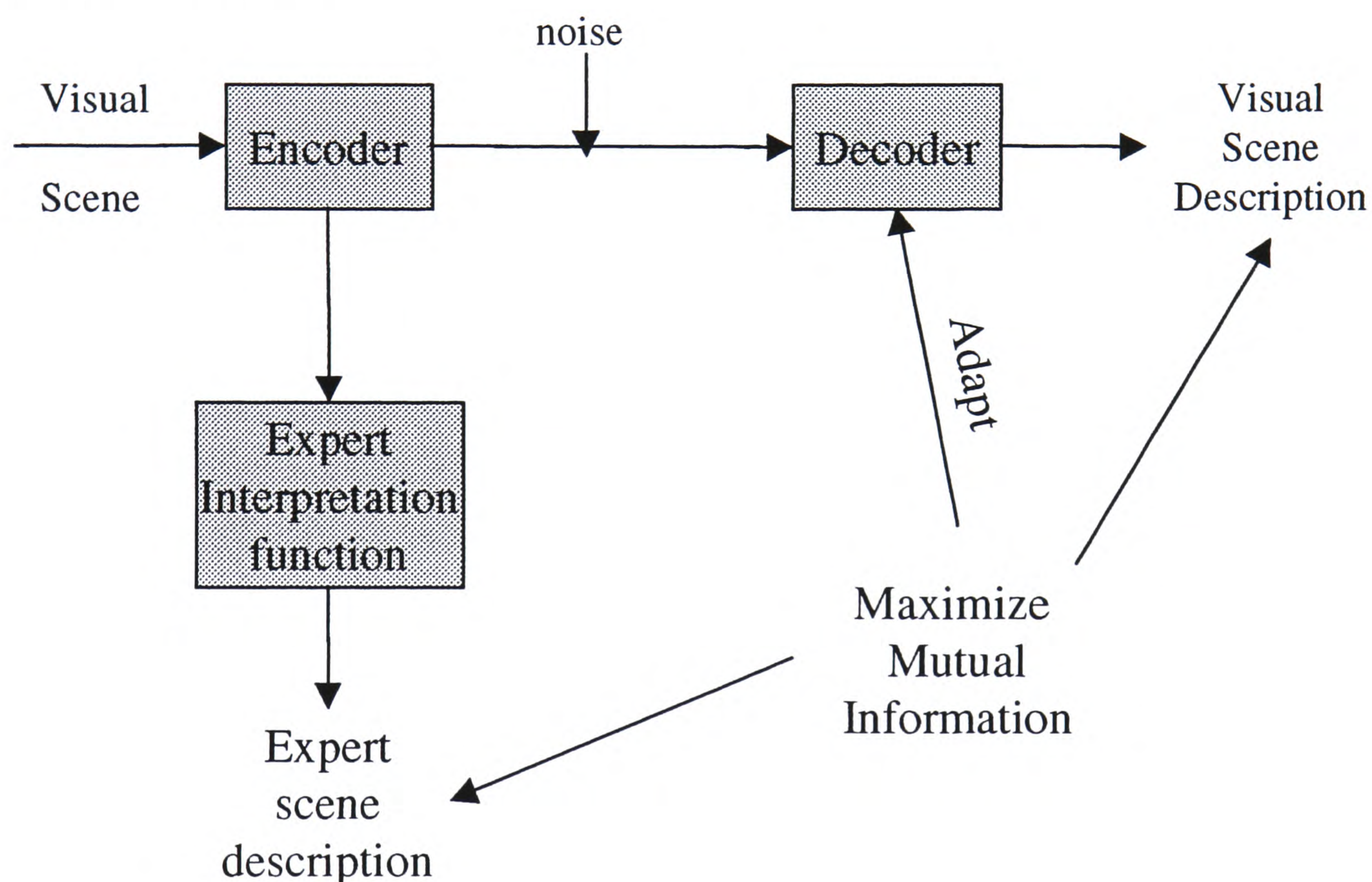


Figure 1.4: Communication Model

Figure 1.4 shows the communication view of the GRAVA architecture. There are two parallel activities involved in producing a good visual scene description. First of all the decoder attempts to produce the most likely description of the visual scene. We define the "most likely" description as the one with the minimum description length representation of the visual scene. The second activity adapts the decoder so as to bring the decoder's scene description as close as possible to those produced by the domain expert.

These two activities highlight the dual focus of the thesis. The first involves an architecture that uses minimum description length as a means of coordinating agents to approximate a global minimum description length description. The second is a self-adaptive reflective architecture to facilitate the adaptation of the decoder.

1.3.3 MDL Agent Architecture

The problem of interpreting the real world is inherently ambiguous. A speech or vision program must select the most likely interpretation from the ambiguous candidates. Selecting the most likely interpretation is equivalent to selecting the interpretation with the minimum description length (MDL). We develop apparently for the first time an agent architecture based on the MDL principle that has a number of desirable properties:

1. It supports a conjecture of Leclerc (Leclerc 1989) that MDL can apply to higher level semantics.

2. Cooperation between agents is an emergent property.

The MDL agent architecture is used as a low level agent language upon which most of the algorithms in this thesis are built. The decision to build the architecture as an agent architecture was for convenience. The choice of agents as a building block is arbitrary.

1.3.4 Semantic Segmentation

The region competition algorithm of Zhu & Yuille (Zhu & Yuille 1996) is probably the leading approach currently to segmentation, it produces reasonably good segmentations based on a purely low level approach; but since there is no provision for combining evidence from higher level semantics, the approach performs poorly when there is poor low level discrimination between regions. By developing an MDL algorithm similar to the region competition algorithm, using the MDL agent architecture, interaction between agents at differing semantic levels allows evidence from higher level semantics to influence the segmentation.

The use of MDL in a segmentation algorithm is not unique (Zhu & Yuille 1996; Leclerc 1989); but the use of MDL as a coordination device for bringing high level semantics to bear on the segmentation *is* new.

The algorithm produces good results. Figure 1.5 shows an image that has been segmented without any high level semantics. Seed points were selected and the base segmenter was allowed to proceed without any image semantics. The results are as good as those achievable using the Zhu & Yuille (Zhu & Yuille 1996) segmenter.

When semantics are introduced, difficult segmentations in which the boundaries are not evident at all such as the Marr example of overlapping leaves shown in Figure 1.2 are possible. Although the leaves can easily be segmented by human sight, analysis of the pixels along the overlapped leaf region shows that there are no intensity changes from which the edge could be found using low level techniques. The edge that we perceive is a kind of subjective contour. We can illustrate this point by demonstrating a traditional subjective contour example.

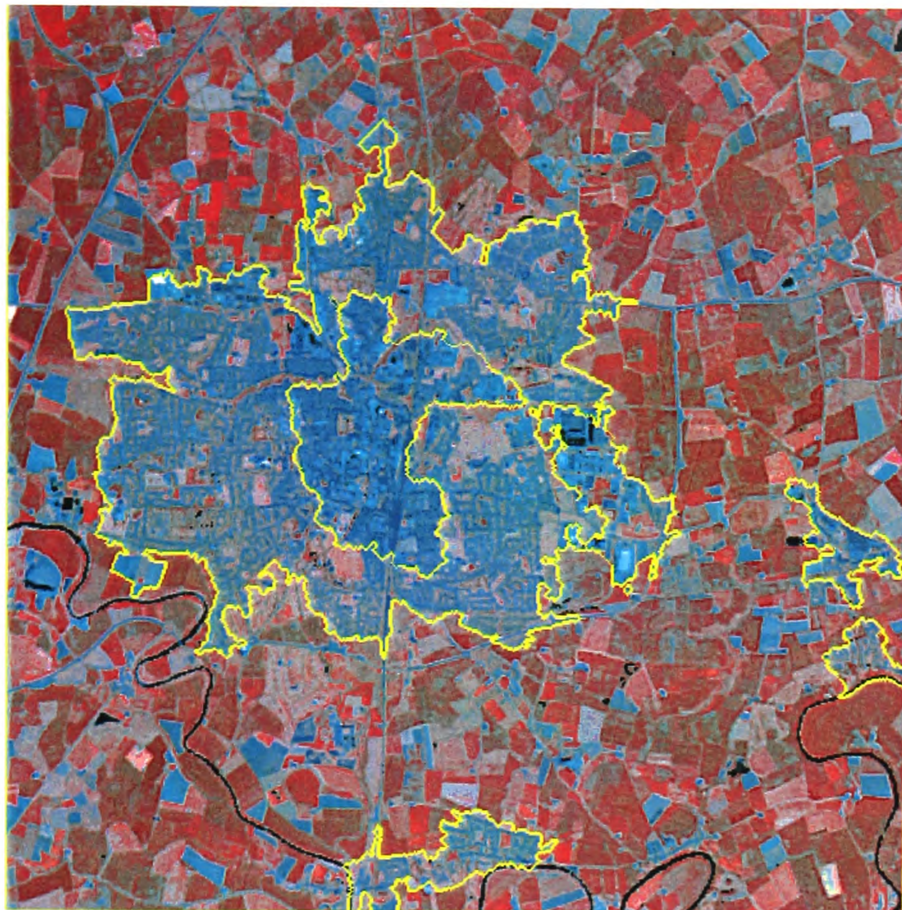


Figure 1.5: Segmented Image

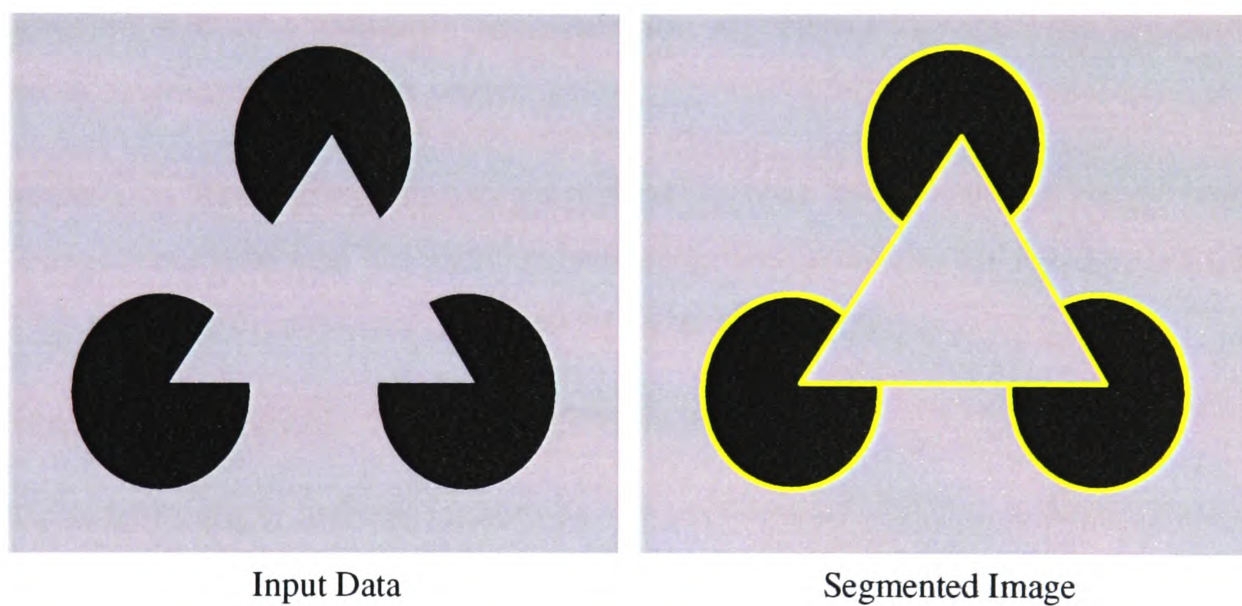


Figure 1.6: Segmenting Kanizza's Triangle

Figure 1.6 shows Kanizza's triangle a traditional subjective contour. The description length is shorter with the triangle than without it so the segmentation produces the occluding triangle. It may be that human perception of subjective contours is produced by an entirely different mechanism but the fact that our segmenter can produce segmentations of this kind is very encouraging.

1.3.5 Patchwork Parser

In our test case, what we have been calling semantics in the foregoing is a meaningful *parse* of the image. The patchwork parser is a statistical 2D parser. The parser takes as input a segmented image with content descriptors assigned to the segments. The parser builds a structural description of the image. Difficulties in producing a good parse can cause self-adaptation, which results in the segmentation program producing different segmentations that may parse better.

Two corpora were developed using aerial images imaged from: a satellite, and from a plane. Annotations were made to the images of the corpora by an expert (our expert was a summer student hired to annotate all of the images using a special image annotation tool described in appendix C). The image grammar is induced from the corpus. The image parser attempts to make structural sense out of the segmentation produced by the semantic segmentation algorithm. This results in:

1. The propagation of contextual information.
2. Assignment of labels to the segments indicating region contents.
3. Interaction with the semantic segmentation algorithm to refine the segmentation based on evidence from the parse.

The parser is a structure-generating application that interprets the image contents. Both the parse structure and the synthesized program structure for the application are generated by a common structure generating theorem prover.

1.3.6 Statistical/MDL Theorem Prover

If image understanding is defined in part as the problem of building a description of the scene depicted in the image, there is a need for some apparatus that can assemble such descriptions. While the component parts of the description are expected to be generated by special purpose agents the task of building a structurally complex description out of these component parts is a problem that does not belong to any of the individual components.

Similarly, the problem of synthesizing the program within the self adaptive framework described above requires some mechanism capable of assembling the individual agents into a form of program.

In both cases, there are many different structures that may be produced. The task of parsing an image is inherently ambiguous. There are often many different allowable parses of the scene just as there are frequently many different parses of English sentences in a natural language parser. Similarly, there are many different program structures that

may succeed in implementing the image interpretation task. In agent planning, there are many possible plans. In each case, the problem is constructing the structure that has the greatest probability of succeeding in the task at hand.

A theorem prover assembles proofs from axioms and rules of inference. The resulting proof can be viewed as a parse tree, a program, or a plan. If the components of the structure have a description length that is determined by a probability that describes the likelihood of succeeding in its particular task the structure most likely to succeed is the one whose description length is shortest.

We have developed a common structure-building component in the form of a theorem prover, and incorporated it into the architecture to address structure building needs. The proof that has the shortest description length is judged to be the best proof. In this kind of proof, the theorem is a valid solution. In the case of a parse it is a syntactically correct structure for the set of lexical items (a lexical item in the case of a patchwork parse is a region). In the case of a program it is a legal program that addresses the details of the program's design; in the case of a plan, the theorem would be a legal plan that addresses the details of the goal structure for which the plan seeks to find a solution. While the parse, program, or plan may be legal none is guaranteed to be correct. The execution of the program may fail at some point. A plan may not succeed. Checkpoints are added to the generated structure whenever a component has a probability of success less than 1. These checkpoints provide a mechanism whereby self adaptation can be initiated.

1.3.7 MDL clustering for model induction

GRAVA is grounded because it applies models that are *grounded* in experience with the real world. We develop a corpus of images annotated by a human photo interpreter as the basis for grounding the models used by the system. We develop an MDL based clustering algorithm that supports the automatic induction of statistical models from the corpus. We call this algorithm principal component decomposition (PCD). It works by searching in decreasing order of eigenvalue for ways of dividing the input space along principal components in order to decompose a complex set of data points that constitute positive examples of a phenomenon for which we wish to induce models. The resulting clusters can be modeled using principal component analysis (PCA) so that the resulting set of models can be described with a minimal description length (MDL). As such the resulting model is the most probable interpretation of the data points.

Figure 1.7 shows data points for river using the red and blue channels. The MDL representation for these data points involves dividing the data into two separate models as indicated by the coloring of the data points.

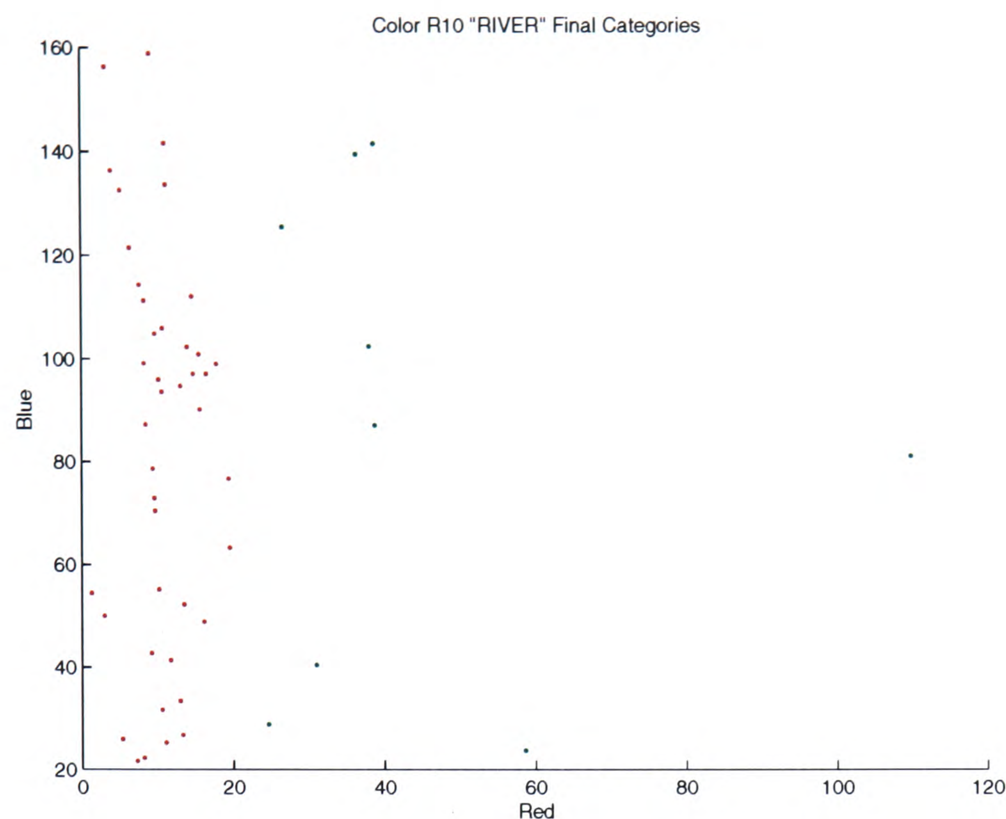


Figure 1.7: Description for River Using Red and Blue

1.3.8 Self Adaptive Architecture

The goal of the architecture is to support self-adaptation. The idea behind self-adaptation is very simple. The program needs to be able to continually access how well it is doing at its task rather than running an algorithm blind as is usually the case in non-adaptive programs. When the self-assessment determines that it is doing poorly the program should seek some way of adjusting its structure so as to do better. The self-adaptive architecture is a collection of supporting capabilities that permits this simple approach to self-adaptation to work. The supporting components are as follows:

1. Self-assessment—the ability of a computational agent to evaluate how well it is doing at its current task. Self-assessment is not so much a component as it is a protocol. The GRAVA architecture provides a protocol for supplying self-assessment functions.

2. Structure building—the mechanism that constructs a program from a collection of computational agents that implements the overall objective of the program. This structure building apparatus is invoked whenever self-adaptation becomes necessary. When self-assessment indicates poor performance the system tries to improve by re-synthesizing its program code. The structure builder in GRAVA is provided by the statistical theorem prover discussed above.

3. Reflection—the support for self-understanding within the system. Reflective systems are systems that contain an embedded semantic account of their computational processes to some level of detail allowing introspection of the program's state

and also semantic modification by mutating the semantic account. The embedded semantic account is not just a static representation but is intimately involved with the operation of the system. By inspecting the state of the semantic account the system can *understand* why it is doing what it is doing. That is it can reason about what the system is doing in terms of a goal that its actions are intended to achieve.

Reflection

To date, reflection has existed as a means whereby a programmer can gain access to the computational state of the program. Sometimes this access is purely introspective. It is, for example, easy to implement a debugger on top of a reflective system. In other cases, the semantics may be extended or modified by the programmer.

The role of reflection in the GRAVA architecture is to allow the system to modify itself. The idea in this case is that if the program can know why it is doing what it is doing and also that it is not doing very well in some aspect of its computation, then the system could in principle adjust itself to do what it is doing in a different way and perhaps be more successful. If the system knows why it is doing the thing that it is not doing very well at, it may be able to find other ways of achieving what it was trying to do. If no other ways can be found, it will fail at a meta-level, causing a meta-level reorganization. The way that these goals are achieved in GRAVA is by having the meta-level goal of the program be described in some form of specification. Agents are provided that interpret that specification and produce a design for a program that would satisfy it. Agents that are built to interpret those parts of the design are then used to interpret the design in the form of a program. The number of levels of *meta* that lay between the meta-goal of the system at the top and the program code at the bottom is arbitrary. When the ultimate program code is run it interprets the image in order to produce its description. This arrangement generates a tower of interpreters. At each point in the decomposition from meta-goals to image interpretation the components at one level are linked to those components at a higher level that played a role in defining the semantics of the low-level component.

1.4 Overview of the Thesis

The components described above are described in a bottom up way in the chapters of the thesis.

The thesis begins (in Chapter 2) with a review of conventional architectural approaches to building robust interpretation systems. The second part of the chapter provides an architectural overview of the self adaptive approach taken up in the thesis.

Reflection is proposed as an important component in building self adaptive systems. A model for self adaptation is proposed that treats the synthesis of a program as an interpretation problem. A reflective architecture is developed that allows feedback to take the form of a reflective act that allows adaptation to occur. An information theoretic framework for computer vision programs is developed that provides a conceptual framework for all algorithms described in this thesis.

Chapter 3 develops a low-level agent architecture that implements most algorithms described in the thesis as well as the adaptive architecture. The low-level agent architecture is demonstrated on a simple test case whose purpose is to interpret simple hand written sentences. This illustrative test case provides a compelling case for the MDL approach. Using only two simple feature extractors—a line endpoint detector (red circles in Figure 1.8) and a line junction detector (blue circles)—the words and characters shown in Figure 1.8 are successfully interpreted even though the features extracted are insufficient to determine the characters uniquely.

Chapter 4 demonstrates how the MDL agent architecture can be applied to the task of segmenting images. A novel algorithm is presented that we call *Semantic Segmentation.* The segmentation algorithm, when seeded appropriately and in its base form, produces segmentations that are as good as those produced by Zhu & Yuille (Zhu & Yuille 1996) or Leclerc (Leclerc 1989). The real power of semantic segmentation however is its ability to draw upon higher level semantics. Unlike traditional segmentation algorithms the MDL agent architecture permits image semantics to participate in the segmentation process.

A system based on semantic segmentation will produce a segmentation that depends not only on the bits directly available in the image but also on the models of the world that are available and the likelihood of their occurrence in the world. We see what we know.

Chapter 5 develops the statistical parser that is used to produce the image description. Three problems are solved:

1. The 2D image meta-grammar is developed;

2. The construction of models of region content descriptors in terms of region content features, how they are learned from the corpus and how they are related to symbolic names such as field is developed.

3. Grammar induction from an annotated corpus is developed; and

4. The MDL parser that searches for the MDL parse is developed;

Chapter 6 addresses the issue of grounding the architecture by providing support for the induction of models from an annotated corpus. An MDL clustering algorithm

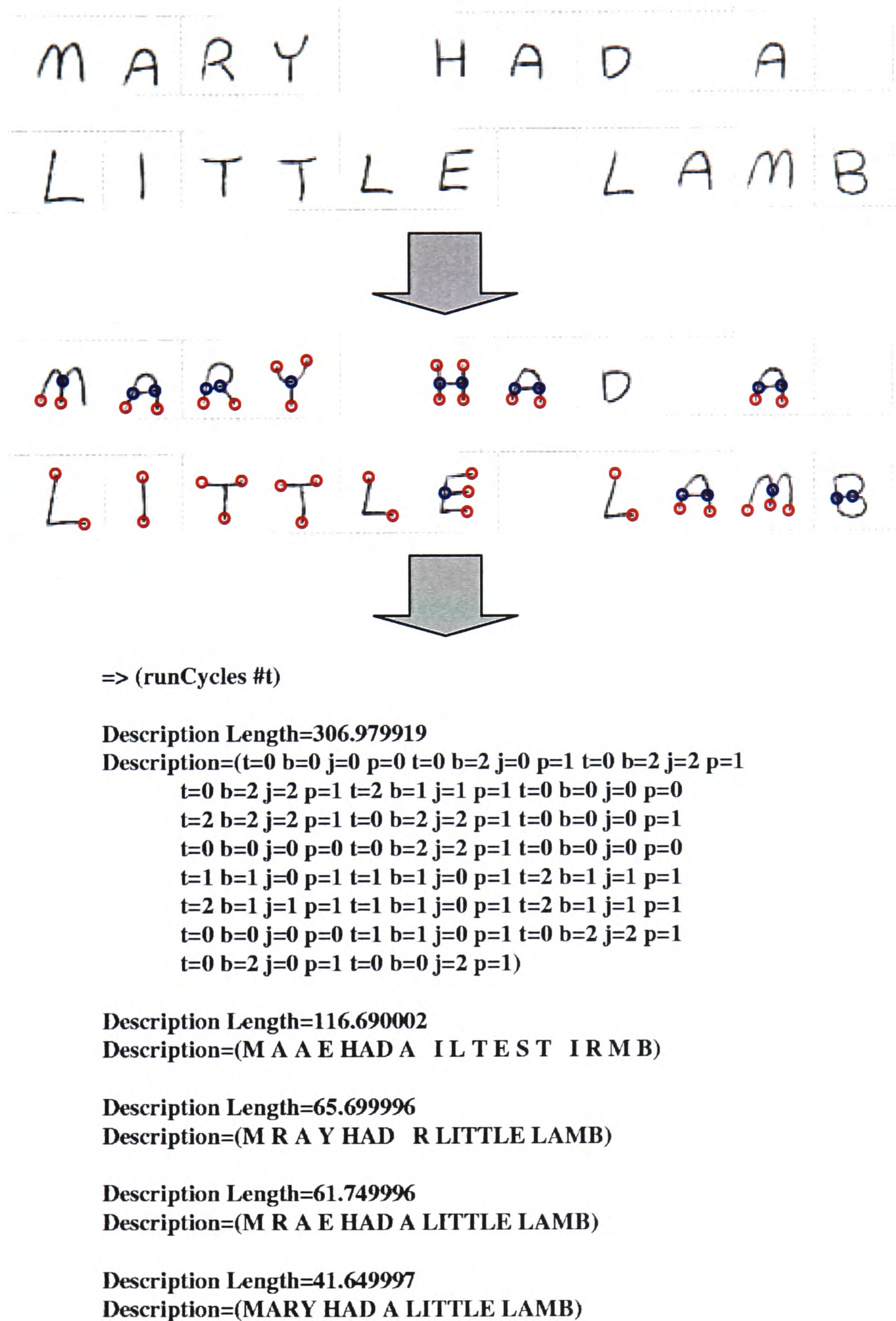


Figure 1.8: Hand Written Word Recognizer

is developed that, without having to specify the number of clusters in advance, can approximate an optimal division of the data points into separate models.

Chapter 7 develops the program synthesis capability and shows how it can be used to implement the self adaptive architecture outlined in chapter 2. The relaxed theorem prover is developed that is used as the structure generating component for program synthesis.

Chapter 8 brings together all of the components of the architecture and the segment-

ing and labeling test case described in this thesis and shows how they work together.

Chapter 9 assesses the success of the research and discusses further work. A discussion follows outlining outstanding problems and issues for self adaptive software research.

Appendix A describes the image corpora used in the project. Appendix B gives an overview of the software architecture that was implemented as a test bed for the research presented in this thesis. Appendix C describes the image annotation facility that was developed for the purpose of annotating the image corpora. The image annotation facility was used by the student image interpreter to annotate all of the images in the corpora.

Chapter 2

Architecture

2.1 Introduction

I am not the first to recognize the need for more appropriate architectures in computer vision or in Artificial Intelligence. Architecture is important and necessary because it addresses the issue of how to build complex systems that work. While much work in computer vision is focused and low-level, success in building complete vision systems requires the results of all of the focused work on low-level vision to be integrated into an application that can effectively apply those routines and integrate their results so as to achieve robust visual interpretation. Building such systems is essentially an architecture problem. The architecture problem is also the Artificial Intelligence problem as it deals with the integration of information, interaction with the real world, manipulation of beliefs and intentions, and focus of attention. Over the years, a number of architectures have been developed that are relevant to computer vision. In this chapter we review the relevant architectures and highlight the advantages and problematic areas of each. The self-adaptive approach and communication theory are introduced. Self adaptive software has only been around as an architectural idea for a few years but already there are a number of ambitious projects underway. We provide an overview of these experiments in self adaptive software so as to situate the work of this thesis within the field of self-adaptive software.

We conclude by sketching a high level overview of the GRAVA architecture. The details of the GRAVA architecture are elaborated throughout the thesis in the context of solving aspects of the problem domain.

The problem that this thesis attempts to address is building image understanding systems that are robust with respect to the environment in which they operate. This is a difficult problem because the environment can be complex and the program complexity required to deal with it would involve highly complex program logic. The thesis began with a conjecture that an effective way to build vision systems that can accurately and dynamically match up vision capabilities with the dynamic visual world would be to have the systems *understand* what they are doing and to *know* how well they are doing at it. We conjectured that *Reflection* (Maes & Nardi 1988) provided an appropriate architectural basis for building such systems. Understanding what the systems is doing

and knowing how well it is doing at it is only half of the story though, because when the program finds itself not doing well at its task, it must have a mechanism for making changes and thereby doing better.

The traditional approach to building systems is to try to think through the whole problem and to consider every possibility. By programming every possibility in to the logic of the program, a complete programmed solution can be constructed. Most traditional systems are built this way. The approach is effective when the problem is of manageable size and when the system/environment does not change too much or too often. The size and complexity of the resulting system reflects the size and complexity of the problem definition.

A number of architectural approaches have been developed over the years to address the goal of having a programmed system of limited complexity produce complex and interesting behavior in the face of a complex environment. Most of these can be characterized as *data driven* or *reactive* approaches as follows:

- Blackboard and Forward chaining systems: Blackboard architectures (Erman *et al.* 1980; Erman, London, & Fickas 1981; Hayes-Roth 1985; Jagannathan, Dodhiawala, & Baum 1989) have been used successfully in a variety of applications including speech understanding (Erman *et al.* 1980) and vision. Forward chaining languages (Forgy & McDermott 1977; Robertson 1985; Laddaga 1996; Rowley, Shrobe, & Cassels 1987) are very similar to blackboard architectures. They have tended to be structured as languages rather than architectures and have largely been employed as the basis for building expert systems; but they are very similar computational engines. They are structured so as to invoke specialized routines when the prerequisites for the routines are satisfied by the current state of the environment that is represented on a blackboard or working memory structure. All of these systems are data driven. The choice of what routines to run is made on the basis of what the environment looks like. Their behavior is hard to predict and control even in a relatively stable environment.

- Subsumption architectures (Brooks 1986; 1987; 1991a; 1991b; Mataric 1991; 1992) are based on layers of augmented finite state machines (AFSMs). Each layer provides pre-wired behaviors and operates asynchronously. Each AFSM supports suppression and inhibition from other AFSMs allowing coherent behavior to result. Subsumption has mostly been applied to building robot controllers but in principle could be applied to other problem domains. Subsumption provides a way of allowing the robot to react to its local environment while performing a requested behavior. In that sense it is both goal directed and reactive.

- Schemas

> Schemas as developed by Riseman (Draper *et al.* 1988) are an extension of the blackboard idea and have been used as an architecture for computer vision (Arkin, Riseman, & Hansen 1988). Schema instances provide distributed control over how knowledge is applied and which knowledge sources are used. This overcomes problems in working with a centralized scheduler by providing explicit control information in the form of schemas.

Other approaches include parallel distributed processes (PDP) (McClelland & Rumelhart 1986a) which we discuss in chapter 3.

Many people have postulated that things would be much better if there were a mathematical foundation. In this thesis we go some way toward that by basing the architecture on MDL. By employing *ad hoc* architectural ideas there is a danger that the architecture itself becomes a source of unmanageable complexity while not providing reliable solutions to the problems they seek to solve. The architectures reviewed here allow for complex and *interesting behavior* by supporting complex interactions between simpler components. They suffer not from limitations in what the architecture will allow—for they all allow a great deal—but from the problem of coordinating the computational components to yield coherent and relevant behavior. The architectures support the development of unmanageable complexity and demand carefully designed control mechanisms and intricately designed interfaces between components. Control in these architectures has to be wired in to the code by the system developers.

The GRAVA architecture is novel in that it seeks to achieve its goals of robust and interesting behavior through self adaptation and reflection but what sets it apart from earlier architectures is the communication theoretic underpinnings that provide the basis for a crisp mathematical foundation and which supports in a reasoned way the kind of *interesting behavior* that is so tantalizing yet ultimately frustrating in the older architectures.

2.2 Blackboards and Forward Chaining Rule Systems

The blackboard architecture is, on the surface, a very simple idea. A problem is posed by placing a description onto a shared data-structure called a blackboard. This starting description could be the raw data from a speech understanding system. A number of routines known as blackboard knowledge sources (BBKS's) watch the description on the blackboard for part of the description that matches their condition part. Each BBKS is defined as a condition part that matches against part of the description on the blackboard and an action part. When a BBKS finds part of the blackboard that matches its condition part it may be invoked. Once invoked the BBKS runs its action part in order to rewrite that part of the description. In principle this allows BBKS's to apply their knowledge

as appropriate in order to transform the original description into a solution. At each step, BBKS's apply themselves to pieces of the description to which they are matched and thus a form of cooperation among BBKS's is achieved and the emergent function of the blackboard is the complex and interesting behavior of transforming the initial description into a higher level understanding.

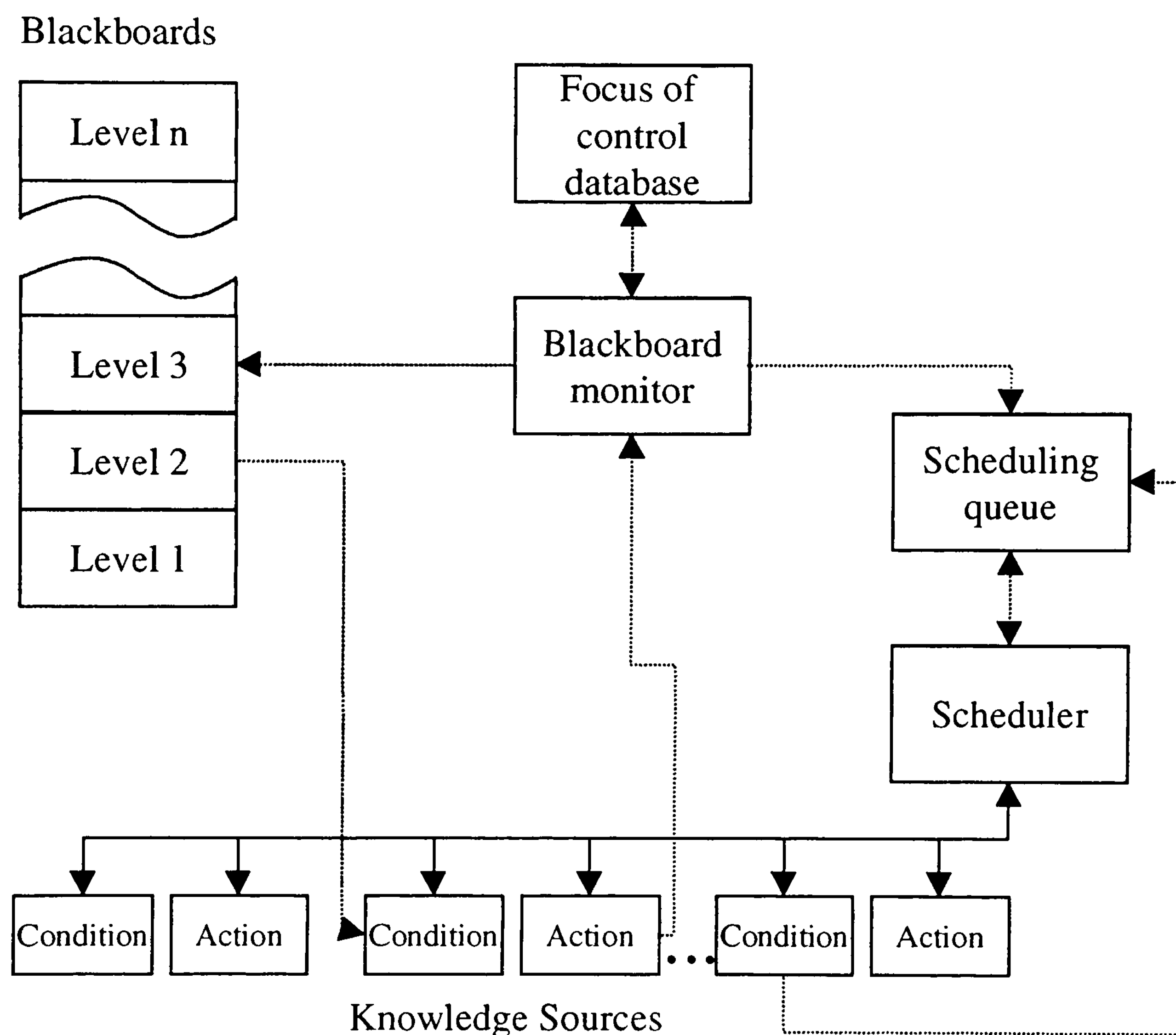


Figure 2.1: Hearsay II architecture

Figure 2.1 shows the schematic for the Hearsay II architecture (Erman *et al.* 1980). The blackboard is divided into multiple levels. This allows the problem to be modeled as moving the original low-level description to a higher level description. A BBKS can read its part of the description at level n and write its interpretation at level n+1. In this way BBKS's gradually transform the low-level description up through the levels of the blackboard until a high-level understanding is achieved. The simple idea breaks down in practice because at any time there can be more than one BBKS that may potentially be invoked for the same piece of raw data on the blackboard. To overcome that embarrassment of riches, the Hearsay II architecture provides pieces of mechanism

to manage the application of BBKS's. Overall control of the interpretation task is managed by a focus of attention database and a blackboard monitor. BBKS's that are applicable at any point in the interpretation advertise themselves to the scheduler. The condition part of the BBKS is compared with the focus of attention in order to select the BBKS that should be invoked next. The naive view of an anarchy of BBKS's cooperating to transform the description on the blackboard in reality degenerates into a highly engineered solution in which the correct sequence of applications of BBKS's is managed via the scheduler and the focus of control database. For a complex problem the management of application of BBKS's becomes a major problem and the separation of knowledge that is represented by the separate BBKS's is more illusion than reality. In reality they must be carefully designed so as to work with the other BBKS's to lead the computation towards a solution.

The ideas of the blackboard architecture have frequently reappeared in other approaches. In Rule based systems (Forgy & McDermott 1977; Robertson 1985; Laddaga 1996; Feigenbaum, Buchanan, & Lederberg 1971; Buchanan & Shortliffe 1984) rules are described by condition–action pairs just as they are in a blackboard and a shared data structure is used to represent the problem. In the case of forward chaining systems like OPS (Forgy & McDermott 1977) a shared working memory takes the place of the blackboard and a collection of simple control rules determine the order of application of the rules. For example, in the case of OPS5, one control strategy applies more specific rules in preference to less specific ones. Elements in the working memory are time stamped and rules that deal with elements with more recent times stamps are preferred. In such a way some kind of focus of attention can be achieved. The developer of such a system ends up carefully designing the rules so as to achieve the desired pattern of execution. This programming style is opaque and difficult to debug as systems become large or as the environment changes in ways that were not originally envisioned. Complex and successful systems have been built—and continue to be built—using blackboards and rule based systems, and a number of commercial implementations exist to support such activities; but mature systems built this way rapidly become maintenance nightmares.

2.3 Subsumption

A variation on the focus of control ideas of blackboards and rule based systems is provided by the Subsumption architecture. The intriguing idea of subsumption is that instead of having a complex model of the world that mirrors the real world one can get by with a very simple idealized model of the world. In getting a robot from point A to point B, instead of having a model of the world and a plan for getting from A to B, subsumption would ignore the possibility of any complexities of the world that might frustrate a

straightforward solution to getting from A to B.

Subsumption has largely been used to implement robot controllers. The controller has a high level goal such as "get from A to B". The rules that implement the goal (implemented in subsumption by finite state machines) don't attempt to anticipate difficulties caused by the terrain. For example the strategy might involve turning towards B and executing move forward commands until B is reached. Along the way certain obstacles may be encountered. When they are encountered control is subsumed by an obstacle avoidance layer. When the obstacle is no longer a problem control returns to the higher goal achievement layer where the simplistic approach of getting from A to B resumes. Complex and interesting behaviors result from the subsumption approach. The emergent function of the subsumption architecture is a product of both the algorithms encoded in the layers of finite state machines and the environment in which the robot operates.

The strength and weakness of the subsumption layered approach is that the layers know nothing about each other. In the example described above the AFSM for avoiding an obstacle knows nothing about the goal of getting from A to B and as a result silly behavior often results. Subsumption is purely reactive and has no notion of a model of the world and there is no notion of a plan.

2.4 Agents

A similar approach to interacting with an arbitrarily complex environment is taken by situated agents (Maes 1990b; Giroux 1995; Krogh 1995). Agents are modules of competence. On one level, the agent view is nothing more than a simple organizing principle like object oriented programming. From the standpoint of artificial intelligence the idea behind agents is that intelligence may be decomposable into small components of capability—agents. Agents that organize themselves are known as autonomous agents (Maes 1990a). By having a collection of agents each of which has some self contained expertise, and by having some mechanism whereby these agents can cooperate to solve complex problems, the behavior that emerges from that interaction may appear similar to behavior that we call intelligent when it is performed by humans.

The idea of emergent behavior (Maes 1990a) is important. The way that agent systems are designed is by finding some form of interaction loop between the collection of agents and the environment that converges on a solution to the problem. Systems built this way can exhibit surprising robustness and interesting behavior. Agent implementations are extremely varied. Various mechanisms exist for controlling how agents organize themselves and communicate with each other in order to have the emergent behavior of a system that solves the desired problem. A key idea of agents is that (unlike blackboards)

there should be no central control mechanism. There are, however, often other kinds of central mechanism such as a directory of agents that can be used by an agent seeking to find other agents to help to solve a problem.

Often there is more than one agent capable of solving a particular problem. This is similar to the problem with blackboards and with forward chaining expert systems when there are multiple rules that can potentially be invoked. It follows that with agents the fundamental problem is the same as with the other architectures—finding ways of controlling which agents will be selected to run when there are multiple choices. Various mechanisms have been suggested for how to solve the agent selection problem. The benevolent agent assumption says that all agents are eager to cooperate on a problem and the problem lies in selecting which agent is most suitable. Another approach is to use a market based system. Market based systems (Clearwater 1995) implement fees for the services of agents and agents can bid on an a problem solving activity. Each agent is assumed to be greedy in wanting to spend as little on tasks that it has to delegate and collect as much as possible from the agents that it performs work for. Self organizing agents must communicate not only to exchange problem specific information but also information suitable for determining if the agents are capable of solving the problem that they may be called upon to perform and perhaps to come up with a mutually agreeable price. Negotiation (Rosenschein & Zlotkin 1998) is one way to achieve this.

Of particular interest is the BDI (Rao & Georgeff 1995) agent. BDI agents have a simple interpreter that depends upon three pieces of information maintained by each agent: belief, desire, and intention.

Belief is the agent's beliefs about the world. *Desire* is the agent's goal—the problem definition; and *Intention* is a representation of the agent's approach to solving the problems—the plan.

BDI agents are reflective in the sense that they carry enough information to reflect on their computational state. Whereas BBKS's and rules are more or less blind condition action pairs BDI agents know what they are expected to do [desire], their current plan for accomplishing it [intention], and their beliefs about the world [belief]. In principle the belief and the intention can change at runtime and thereby cause the semantics of the agent to change.

2.5 Schema

Riseman's Schema is an attempt to marry top down and bottom up control in a vision architecture so as to manage the problems described above with the reactive systems. A schema is a semantic entity that can be supported by lower level knowledge. A schema system breaks down a complex task—such as understanding an image containing houses

and trees into sub schemas that are responsible for recognizing the parts that make up such a scene—the houses and the trees. The elaboration of a hierarchy of schemas bottoms out in a collection of low-level processes that can contribute to understanding various parts of the image. Having chosen a set of processes in this way, the interpretation of the image proceeds in a reactive way with the low level processes finding things in the image and then having their contributions used as evidence for trees and houses in the appropriate schemas. The top down decomposition of the problem into a hierarchy of schemas which each contribute low-level processes helps to control which low-level processes are involved in the computation and thereby constrains the set of processes that may contribute to understanding an image.

2.6 Self Adaptive Software

One problem with data driven approaches is that it is hard to steer the computation in the right direction. The result is that focus of attention is implemented in a way that is inflexible and hard to maintain. In forward chaining systems, for example, there may be a number of rules that could manipulate the data in the working memory. However, which of those rules leads in the right direction? To force the system to select *the* rule that does what we want we have to cunningly place data elements in the working memory that indicate what we want to happen and the data driven rules have to be constructed so that they attend to what is wanted. The subsumption approach has a top level program that implements the intended behavior in an idealized world. When the real world deviates from an ideal one, such as by including an unexpected obstacle, a lower level subprogram subsumes the current computation and *hides* the parts of the real world that are not *ideal* such as by moving around the obstacle. Unfortunately, there is no way for the lower level sub program to know what the higher level was doing and so it is easy to generate nonsensical behaviors. For example, if the robot responds to an obstacle by moving away from it but whenever the higher level program regains control it moves back towards the obstacle the robot will endlessly back up and move forward.

All of the above approaches are reactive. In each case, a lack of understanding about the goal to be achieved by the system is the major weakness of the system. The self-adaptive approach is different in that the system solves the problem of how to make a programmed solution react to failure and to unexpected events in the environment. At any point, there is a programmed solution that has a model of the world that is trying to interpret the world. When failure occurs, for whatever reason, the programmed solution is mutated so as to deal with what is understood to be the true nature of the world.

The self-adaptive approach is *model driven* not data driven. The program's goal is

modeled. A program exists or is generated that satisfies the program's goal. At runtime, the success of the program is checked by its components and poor performance results in adapting the program so that it still satisfies the goal model.

A good way of thinking about self-adaptive software is that instead of having one program that deals with the full diversity that the program *may* be exposed to, a collection of smaller simpler programs cover patches of the space of diversity and at runtime the running program is changed to suit the particular features of the environment.

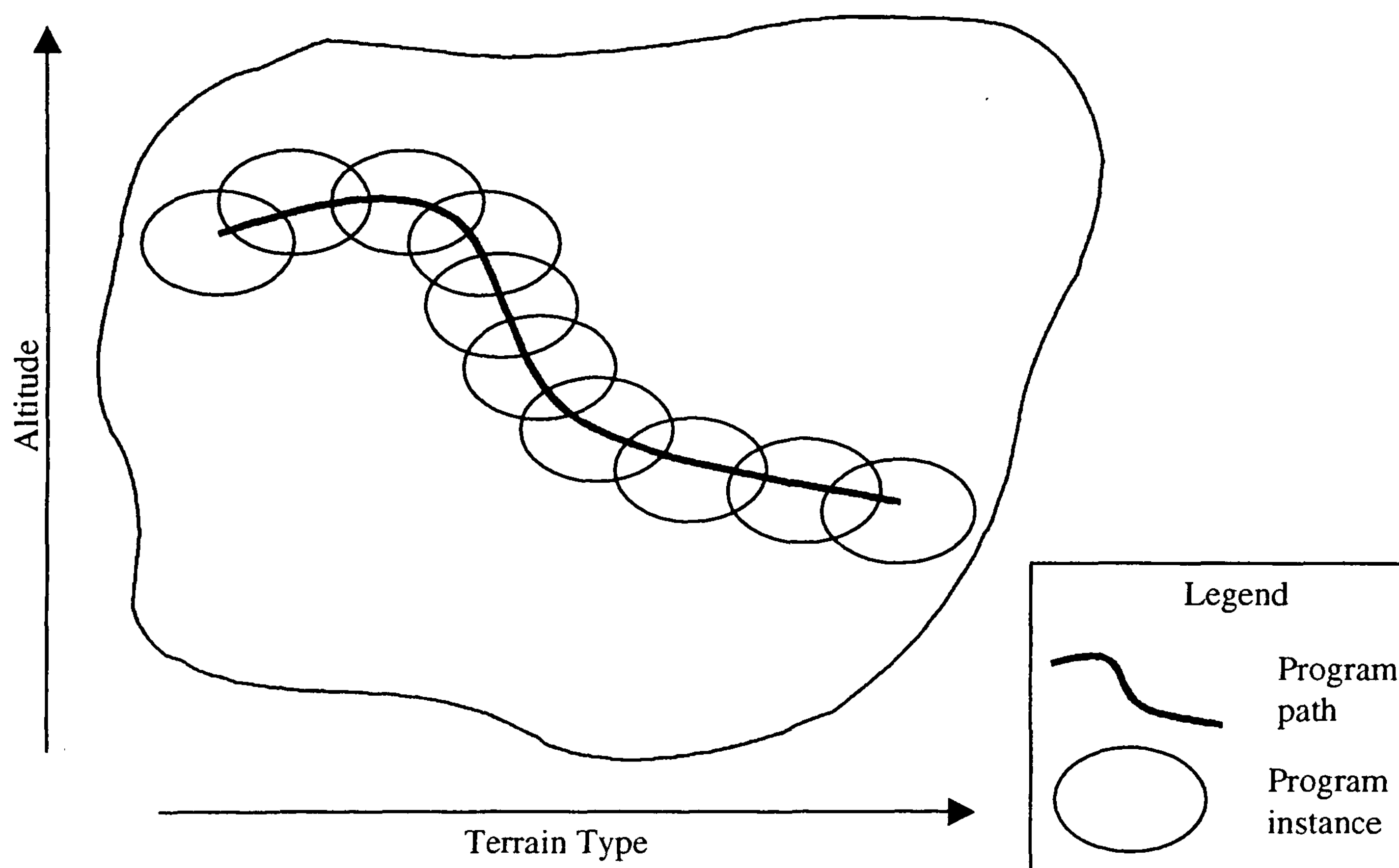


Figure 2.2: Path of a Self Adaptive Program

Figure 2.2[1] illustrates the case showing two dimensions of diversity: altitude and terrain type. If we imagine a camera attached to a plane that is flying over varying terrain and changing altitude as it goes indicated by the solid path we can imagine a collection of programs each suited to understanding images at a particular altitude and terrain type indicated by the ellipses. As the environment changes, the self-adaptive program must track where it is in the space of environment features and select (or generate) the program that best deals with what the environment is currently presenting. The result should be that the program code that is executing at any time is best able to deal with the current circumstances.

[1]This illustration is borrowed from Musliner (Musliner *et al.* 1999).

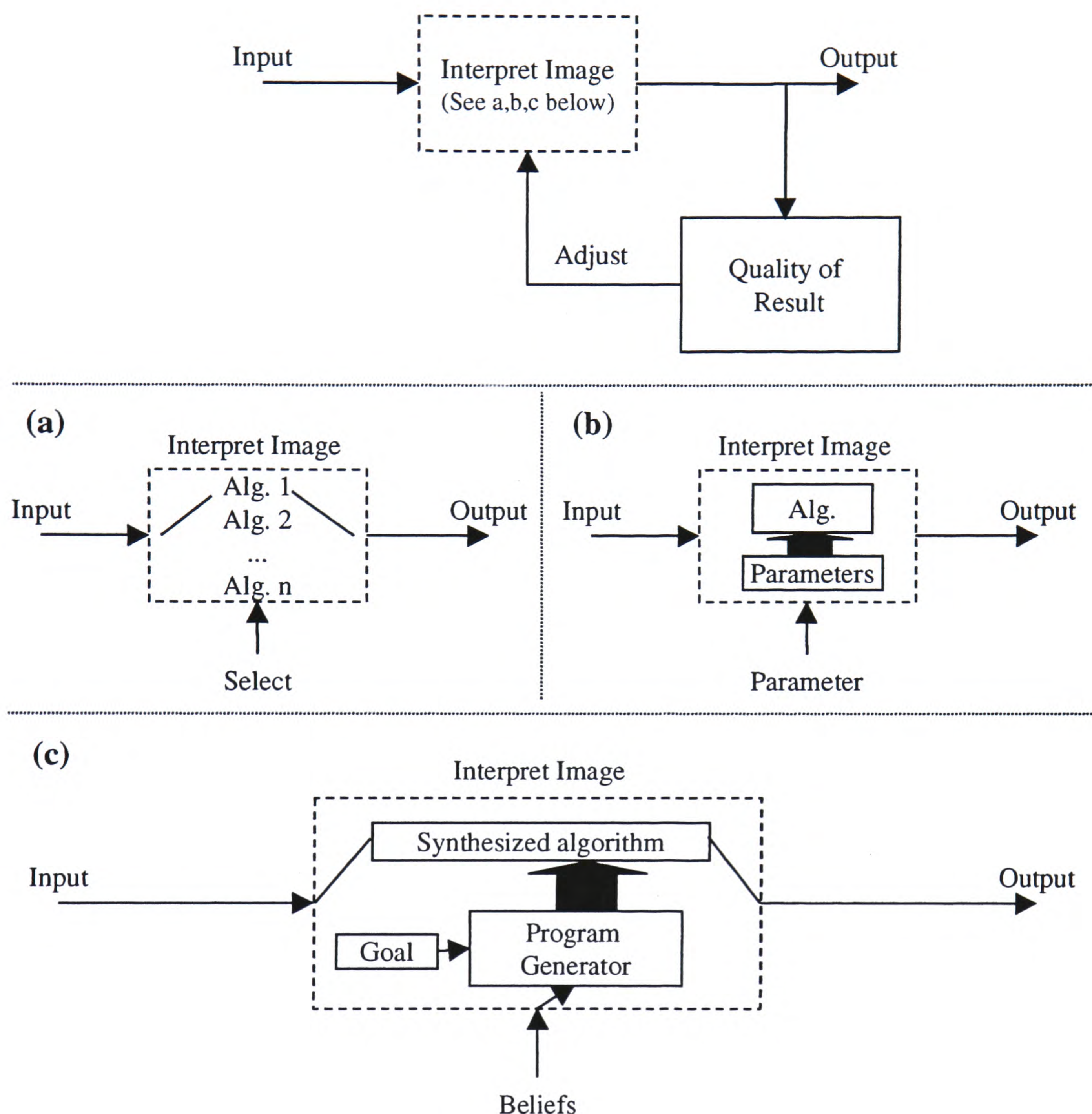


Figure 2.3: The Self Adaptive Loop

The self-adaptive program tracks changes in the environment by continually monitoring the quality of the results that it produces and making adjustments as necessary. Figure 2.3 shows the feedback loop implicit in this approach. The loop is simple but begs the question "how is the quality of result computed?", and "how does *adjust* modify the *interpret image* process?". We will deal with these questions in reverse order. Several possibilities suggest themselves for how the interpret image function could be changed by adjustment. The simplest idea is simply to have multiple algorithms for interpreting the image (Figure 2.3a). When it becomes necessary to adapt, adjustment takes the form of changing the selected algorithm. This makes adaptation into a dynamic algorithm selection problem. This approach is natural if you have several agents that achieve the same end result. This approach requires only that the input to "adjust" be able to

choose which algorithm is best from a set of alternatives. An even simpler approach is to leave the algorithm unchanged, but instead adjust its parameterization. (Figure 2.3b). Parameter adjustment is a generalization of a big switch since the algorithm could be a collection of algorithms wrapped by a big switch that selects an algorithm based on a parameter value. The most general approach, however, that includes both of the above involves resynthesizing the program (Figure 2.3c). The resynthesis approach requires the runtime presence of a program generator. This is the approach that we develop in this thesis. The synthesized algorithm is generated from two pieces of information:

1. The program goal.

2. The program belief state.

Both of these inputs to the code generator can change. The program goal is a function of the consumer of the program's output. The belief state is something that changes as the program runs and learns about the environment that it is in. As the environment changes so does the program's belief state. The belief state can be updated explicitly by the running agents or it can be updated when there are causes to adapt. The program goal and the belief state together contain the information necessary to synthesize the algorithm. The result of the *quality of result* component is that the belief state gets updated and this causes the synthesis engine to produce a different program.

A key idea behind the self-adaptive approach to achieving robust performance is that it is easier and often more efficient to postpone certain decisions until their need is evident from the circumstances. It is often easier to make a program that monitors its performance and recovers from errors than it is to make a program that goes to great lengths to avoid making any errors. This notion is already familiar to us in our use of errors and exceptions in languages like Common Lisp (Steele 1984) and Java (Gosling, Joy, & Steele 1996).

Exceptions allow simple code to be written that expects the environment to be well behaved. The code must also include checks that everything is as expected. If things are not as expected an exception is raised. Elsewhere an exception handler contains the code that decides how the exception should be handled. For example if the code was trying to read a number and a non numeric character was detected, a "non numeric character in number" exception may be raised. The handler could then clear the input buffer, caution the user against using non numeric characters and retry the read. In this example the program has a 'model' of what it expects (a sequence of numeric digits). The program checks for differences between what it is getting as input and the model and when such differences occur the program tries to deal with them robustly--in this case by issuing a warning and trying again.

While the exception mechanism is invaluable for a certain class of problems and is suggestive of a general approach, it does not easily extend to the full range of problems that can occur in complex systems. When a robot encounters difficulty in understanding a visual scene, it could be for a variety of reasons: a sensor may be functioning incorrectly or have failed completely; the lighting may have changed so as to require a different set of algorithms; the contents of the scene may have changed requiring the robot to update its model of the environment; and so on. In general the environment cannot be given a warning and asked to try again! The program must deal with what it gets as best it can. In these situations when a sensor fails or an algorithm doesn't work or some other assumption about the environment turns out to be untrue, the solution will often require reconstituting the program to deal with the actual environment rather than having enough complexity to deal with all possible environments hidden within exception handlers.

Self adaptive software therefore involves having suitable models of the environment and of the running program, the ability to detect differences between the models and the actual runtime environment, the ability to diagnose the nature of the disparity, and the ability to re-synthesize or modify the running program (or its models) so that the program can operate robustly.

The notion of programs that can manipulate their own representation is not new and has been a key characteristic of Lisp (Steele 1984; Keene 1989) since its inception, although the full power of that capability has not often been used. The idea of programs that reason about their own behavior and modify their own semantics is well developed. Programming languages that support self reasoning and self modification are called 'reflective' a term coined by Brian Smith in his original development of the idea of reflective programs (Smith 1982; 1984). Reflection has been around long enough for efficient implementation methodologies to be developed (des Rivieres & Smith 1984; Bawden 1988; Malenfant, Cointe, & Dony 1991; Kiczales, des Rivieres, & Daniel 1993) and some level of reflective capability is now available in CLOS (Keene 1989) and Java. Reflection makes it possible (and convenient) to write programs that can reason about the state of computation and make changes to the semantics of the running program. Self-adaptive software can clearly benefit from such a capability but there is more to self adaptive software than reflection. Self-adaptive software must have a model of what the computation is trying to achieve in order to have something against which to compare the current computation. Having determined some difference between the intended program state and the actual one a self adaptive program must be able to automatically adjust the semantics of the program so that the difference is reduced. Reflection provides the tools for writing such programs but it doesn't provide any guidance in how it should be done.

Managing complexity is a key goal of self-adaptive software. If a program must match the complexity of the environment in its own structure it will be very complex indeed! Somehow we need to be able to write software that is less complex than the environment in which it is operating yet operate robustly. By keeping the program code separate from the diagnosis and synthesis code the core code can remain simple and the synthesis can be performed by a purpose built synthesis engine. Such an architecture makes writing the core code much less of a tangled web. In essence, the total runtime program may be a sequence of different programs each one individually simple. The complexity of the program as a whole is moved into the architectural mechanism that maintains models, checks models against performance, and re-synthesizes new programs.

There are many advantages to this approach beyond making it possible to write programs that operate in complex environments. Correctness of programs may be easier to establish whether by testing, proof, or other means because at any point the operating program should be simpler than a program that attempted to consider every possibility in advance. To some extent, correctness may be guaranteed by the synthesis engine or may be checked automatically because there is a model of the program against which it may be tested. Greater efficiency may also be achieved through the self adaptive software approach. Greater efficiency occurs when a crude but fast program that may not always work is substituted for a complex but slow program that tries to guarantee that it always works. By making decisions at run time when much more is known about the actual environment than could have been known when the program was originally designed it is possible to select more suitable procedures for achieving the intended goal. This can result in a more direct solution with less failure and less backtracking. Having better information about the environment allows better decisions to be made which can result in more efficient and successful execution.

There are also concerns and difficulties evident in the approach. A program that changes itself at runtime may in some sense be a less well understood entity than a program that remains unchanging during its execution. The notion that it may be more robust than its fixed counterpart should make us feel more comfortable about its behavior but the notion that the program changes at runtime is unsettling to people responsible for building complex missile delivery systems (for example). How can we trust a smart missile that may reprogram itself in mid flight? There are (at least) two answers to this. The first is that if the program fails in mid flight because it was unable to deal with its complex environment it may do a very bad thing in its failure mode so the self adaptive version that is less brittle should be less of a concern. The second is that we must develop a better understanding of self adaptive software and new approaches to software testing, debugging, and semantics that allow us to be confident about the systems that we deploy either through a proof theoretic framework or through a testing regime.

Today, self-adaptive software draws upon a great many technologies that may be used as building blocks. Reflection and exception handling mechanisms have already been mentioned; but there are a great many other candidate contributing technologies including model based computing, theorem provers, models for reasoning about uncertainty, and agent based systems to name a few.

Self adaptive software is a relative new idea and the approach involves taking a significant departure from established software development methodologies and potentially requires rethinking some basic concepts in computer science. Self-adaptive software will be achieved by taking small steps from what we currently know. Some people are attempting to build self adaptive software solutions to solve existing problems, some are experimenting with existing technologies to see how they can be used to achieve self adaptive software, some are working with systems that interpret the world through sensors, and some are directly studying aspects of self adaptive software ideas in order to extend our understanding of their semantics.

2.7 Overview of the Current State of Self Adaptive Software

One of the more ambitious self-adaptive software projects that brings together a wide range of technologies is being conducted at the University of Massachusetts at Amherst (Osterweil & Clarke 2000; Grupen 2000). Osterweil et al. describe a process of perpetual testing as a means of achieving self improving software. The ideas are being applied to a complex robot vision system.

The environment in natural systems as well as artificial ones is not only complex but in many instances it can be hostile. Information survivability attempts to deal with systems that come under attack from outside agents. Shrobe and Doyle (Shrobe & Doyle 2000) consider how self adaptive software can provide software services in an environment that is actively hostile. Rather than considering issues of information survivability as a binary affair in which the system is either compromised (and useless) or not compromised (and useable) they consider how survivability can be achieved by self-testing to detect that the system has been compromised in various ways and then adapting so as to be able to provide the best level of service possible given the compromised state of the system.

There is a bone fide need for formal methods in building self adaptive software. Formal methods may be involved in the automatic synthesis of the program when adaptation is required. Formal methods may also be useful in understanding the range of behaviors that a self adaptive program may exhibit. Pavlovic (Pavlovic 2000) provides a first attempt at building a theoretical basis for self adaptive software.

The straightforward characterization of self-adaptive software as continual runtime comparison of performance against a preset goal and subsequent adjustment of the program so that performance is brought back towards the goal is very similar in structure to a control system. The task of building self-adaptive systems may be much more like building a control system than it is to conventional software development. Since certain kinds of control systems, and their design, are well understood, it is hoped that some ideas can be borrowed from control system design and adapted to self adaptive software. Kokar (Kokar *et al.* 2000) takes a radar-based target tracking problem and shows how it can be mapped onto a control system architecture.

When a system is reconfigured at runtime such as by the replacement of one set of sensors with another set undesirable transient effects may occur at the point of switching. Gyula et al. (Simon, Kovacshazy, & Peceli 2000) have investigated the management of such effects.

A key notion of self-adaptive software is that the software contains some embedded account (or model) of its goal and of its environment. Bakey et al. (Ledeczi, Bakay, & Maroti 2000) describe a model based generative technology they call 'Model-Integrated Computing' and show how the technology is applicable to self adaptive software.

While much attention has been given to robotic, sensor based, and embedded applications for self adaptive software there is a role to be played in the network computing environment. Network computing provides all of the attributes of a naturally complex environment within a man made network. Ben-Shaul (Ben-Shaul *et al.* 2000) describes two implemented architectures that provide support for self adaptive software in Java.

Since the need for self adaptive software derives from running applications in an unpredictable environment many of the applications of self adaptive software involve real-time issues. Musliner (Musliner 2000) has developed a real-time self adaptive controller synthesis system that can adapt by dynamically re-synthesizing new controllers. Musliner shows how the synthesis process can be managed so that reconfiguration can meet real-time deadlines.

2.7.1 Systems that interpret their environment through sensors

Grupen et al. (Grupen 2000) have developed a self adaptive motion tracking system that has redundant sensors and the ability to adjust which sensors are best suited to tracking.

Khosla et al. (Dixon, Pham, & Khosla 2000) have developed an architecture that supports self adaptive software through an agent architecture that they employ in a multi-robot mapping system.

Reece (Reece 2000) has investigated the management of redundant sensors for image interpretation. Sensor performance is learned from test sets and qualitative reasoning (QR) models of the imaged environment's physics which allows a theorem prover to

synthesize interpretation code.

2.8 Conclusion

This chapter recapitulates some of the salient features of architectures in AI as I understood them already at the outset of the DPhil work. For the reasons given in sections 2.6 and 2.7, the self-adaptive approach was selected.

Unfortunately very little had actually been built and so my conjecture that self-adaptation was an appropriate basis to attack the changing environment of large systems problem was at best a conjecture—perhaps an article of faith.

Chapter 3

Architecture for Interpretation Problems

3.1 Introduction

The approach described in this thesis is particularly applicable to a class of problems that we call interpretation problems. An interpretation problem is one in which an input, for example an image or signal, must be processed in such a way as to produce a symbolic representation that amounts to a description of the interpretation of that data (signal/image). Such problems are usually ambiguous (there are multiple plausible interpretations) and the task involves finding the most likely interpretation.

We have observed that interpretation problems are frequently best analyzed as being layered. Layering makes it possible to build a reflective interpreter that supports multi-layer interpretation problems. Two ideas are exploited throughout this thesis. The first is the use of reflection as a driver for self-adaptation. The second is the idea of minimum description length as a means of finding the globally most likely interpretation.

Emphasis is given to models that can be learned from a corpus. One of the easiest ways of estimating probabilities—and hence description lengths—is to collect frequencies from representative data. The system described in the thesis uses an image corpus for this purpose. Other mechanisms suggest themselves for estimating description lengths including using Qualitative Reasoning (QR) models and models of uncertainty which we discuss in Chapter 9.

This chapter focuses on how MDL can be used in a novel way as a mechanism by which agents can compete; how such an approach can provide a basis for communication between different semantic levels; and to how this facilitates finding the best interpretation of the data.

The chapter has three main sections. First, we describe the essential contributing ideas: MDL, agents, and communication theory. In the second section we give an overview of how these ideas are realized in the GRAVA architecture by introducing the objects that make up the architecture as well as an overview of their protocols. In the final section, we develop an illustrative example of how agents compete using MDL. Chapters 4 and 5 develop agents within the problem domain that build upon the ideas described

in this chapter. Finally, Chapter 7 completes the architecture development by describing how GRAVA implements self-adaptation through reflection and code-synthesis.

The architecture described in this chapter, and elaborated in the following chapters, was implemented in a reflective object-oriented Scheme-like (IEEE 1991) dialect of Lisp (Steele 1984) called Yolambda (Laddaga & Robertson 1996). Yolambda was designed and implemented by the author as a research platform for exploring ideas in reflective languages (Robertson 1992). Lisp has been a popular language for AI research for many years and its ability to facilitate the implementation of domain specific languages makes it ideal for research such as that described in this thesis. A brief overview of the Yolambda language and its notation is provided in Appendix E.

3.2 Contributing Ideas

3.2.1 Communication

It has long been recognized in speech recognition and natural language processing (NLP) that the problem of interpretation can be modeled as the communication of a message. The communication of a message includes: encoding of the original message; transmission of the message through a channel; and decoding of the message by the receiver. The communication model for interpretation is not limited to speech recognition and NLP: it can profitably be applied to all interpretation problems because all interpretation problems share some basic attributes:

1. Message encoding is important because for there to be something to interpret there must be a mechanism that encoded it. In the case of speech recognition the encoder is the speaker. In the case of magnetic resonance imaging (MRI) the encoder is a physical process. The encoding process may not retain all of the original message – it may be 'lossy'. In echo cardiograms (EGC), the encoding yields fuzzy signals that lacks much of the high frequency components of the imaged physical structure.

2. Transmission of the message is important because in many interpretation problems the message cannot be guaranteed to arrive at the receiver without being partially lost in transmission or corrupted by noise. In the case of speech, the receiver may not be able to hear the word exactly as it was spoken, while in vision the image may contain noise from the passage and scatter of photons through the atmosphere before reaching the sensor and from the sensor itself.

3. Decoding the message is an attempt to reproduce as much of the original message as possible. Since information may have been lost during encoding and during transmission of the message, there is generally not enough information in the message to completely recover the original. We are therefore faced with having to pick

an interpretation from a collection of competing almost equally plausible alternatives. The problem therefore is to pick the most probable interpretation. In order to know which interpretation is the most probable, we must have models of the interpreted world.

Communication theory is the study of how to evaluate the amount of information in a stream of data and how in principle it can be transmitted from a source to a receiver by means of a communication channel.

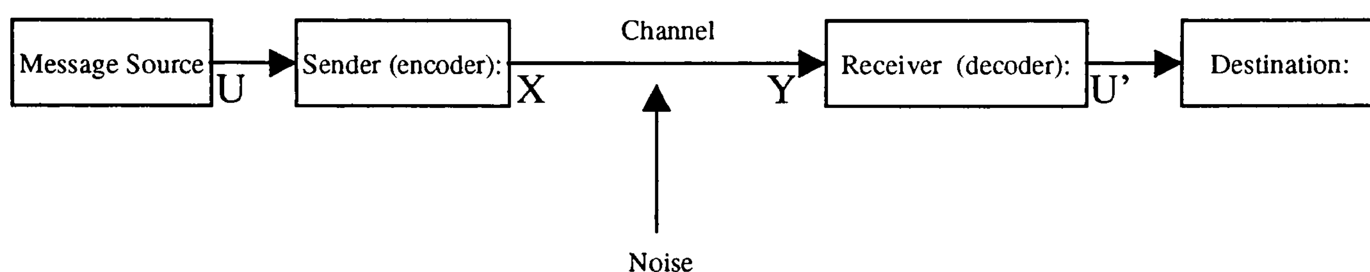


Figure 3.1: Communication of a message through a channel

Figure 3.1 shows the transmission of a message. The original message U is encoded to produce the encoded message X. The message is sent through a noisy channel which results in a corrupted version of the encoded message Y. The decoder attempts to recover the original message from the corrupted version of the encoded message and produces U'.

One approach to developing ideas about the amount of content in a data stream exploits the idea of uncertainty.

Meaning from Uncertainty

In describing the source, we use the term "features" where it is customary to use the word "symbol" in order to emphasize the applicability of the foregoing to the problem of encoding and communicating a representation of the features in some interpreted space—of features.

If we examine the probabilities of features occurring in an input stream, different features will have different probabilities reflecting how common they are (their frequency). In predicting what feature is represented by a particular part of the data stream we can form an estimate of the level of uncertainty of our guess. The uncertainty estimate will be largest if all features occur with equal probability in the data stream. If, on the other hand, the data stream only contains one kind of feature, there is no uncertainty.

If we learn the identity of a particular feature in the data stream we have learned nothing if the data stream was known to contain only that feature. If [on the other hand] the data stream contained many different features with similar likelihoods, we have learned a lot. What we have learned about the data stream is known as information.

If a message M consists of the i features $\tilde{m}_1, \tilde{m}_2, ..., \tilde{m}_i$ which occur with probabilities $p(\tilde{m}_1), p(\tilde{m}_2), ..., p(\tilde{m}_i)$ respectively, $H(M) = H(p_1, p_2, ..., p_i)$ is an information measure.

There have been many plausible functions proposed for $H()$. The most commonly used, but by no means the only, measure of information is Shannon entropy (Shannon 1949). In the Shannon information measure the average information provided by a set of i symbols (or features) whose probabilities are $\{p_1, p_2, ..., p_i\}$ is expressed as:

$$H(p_1, p_2, ..., p_i) = -\sum_{f=1}^{i} p_f log_2 p_f \tag{3.1}$$

Information is measured in bits. The measure incorporates a massive assumption (that limits its usefulness for signals and images) that the probability of a feature in a data stream is independent of its position in the stream—that is, the data stream is an ergodic source. Given the ergodic assumption, the probability of a message M with n features $\tilde{m}_1, \tilde{m}_2, ..., \tilde{m}_n$ with probabilities $\{p_1, p_2, ..., p_n\}$ is given by:

$$p(M) = \prod_{i=1}^{n} p_i \tag{3.2}$$

This satisfies our intuition that while probabilities are multiplicative information is additive and that multiplicative entities can be turned into additive entities by taking their logs.

Given an unbiased 8 sided dice, a message can be formed by n rolls of the dice. There are eight equally likely outcomes from each roll of the dice with probabilities $\{1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8\}$. The information measure for the dice is given by:

$$H(p(1), p(2), p(3), p(4), p(5), p(6), p(7), p(8)) = -\sum_{f=1}^{8} \frac{log_2(\frac{1}{8})}{8} = -log_2(\frac{1}{8}) = 3 \tag{3.3}$$

Each throw of the dice contains three bits of information. A message containing two successive throws of the dice contains six bits. The probability of any such message of two throws is $1/8 * 1/8 = 1/64$ and $-log_2(1/64) = 6$.

Mutual information $I(X \wedge Y)$ measures the amount of information common to two sources. Intuitively this is the amount of information that X tells us about Y (and vice versa).

$$I(X \wedge Y) = H(Y) - H(Y|X) \tag{3.4}$$

The symmetry of the equation allows X and Y to be interchanged yielding $I(X \wedge Y) = H(X) - H(X|Y)$

A more intuitive definition for mutual information is given by subtracting the total amount of information contained in the X and Y messages from the amount of information in each of the messages X and Y.

$$I(X \wedge Y) = H(X) + H(Y) - H(X, Y) \tag{3.5}$$

Mutual information helps us to define imitation learning which we develop in chapter 7.

The problem of efficiently communicating a message over a channel involves choosing codes to represent the features in the message. The codes should be chosen so as to minimize the average code length in such a way that the original message can be reconstructed by the receiver. The minimum code assignments depend upon the nature of the channel through which they are being transmitted. If the channel is noiseless so that the codes can be guaranteed to reach the receiver exactly as they were encoded, the noiseless-coding theorem (Shannon) allows us to calculate the expected code word length for an optimal coding.

Let U be a random variable that describes the emission of features u_i from the source with $P(U = u_i) = p_i$. Suppose that u_i is encoded by $f(u_i) \in G_a^*$ where G_a^* is the set of strings of arbitrary length made from a distinct coding symbols. The length of the code $f(U)$ is denoted by S and the expected code word length $E(S)$ is given by:

$$\frac{H(U)}{log_2 a} \leq E(S) < \frac{H(U)}{log_2 a} + 1 \tag{3.6}$$

The lower bound is attained when $p_i = a^{s_i}$ for integer s_i. So, for example, for the eight faced dice example the expected code word length is given by:

$$\frac{H(U)}{log_2 a} = \frac{3}{log_2 2} = \frac{3}{1} = 3 \tag{3.7}$$

An algorithm for computing $f()$, for mapping source features to code words was provided by Huffman. In this thesis, however, we are concerned with computing the optimal code word lengths but not in actually producing the codes. The theoretical optimal code word length for a given symbol S is given by:

$$DL(S) = -log_2(P(S)) \tag{3.8}$$

3.2.2 Minimum Description Length

The notion of theoretical minimum code length described above provides a convenient way of restating the goal of "most probable interpretation" in communication theoretic

terms.

An interpretation is a description that consists of many parts. Each part has an associated probability.

Consider the case of an image containing two blobs. If the probability of the first blob being a boy given by $P(blob_1 = boy) = P_{boy}$ and the probability of the second blob being a dog is given by $P(blob_2 = dog) = P_{dog}$ the probability that the image is one in which the first blob is a boy and the second blob is a dog is given by $P_{boy} * P_{dog}$ (from equation 3.2). An image containing n blobs such that for any $i < n$ the interpretation of $blob_i$ is given by $I_i(blob_i)$ and the probability of such an interpretation being correct is given by $P(I_i(blob_i))$ the goal is to maximize the probability (assuming conditional independence):

$$arg \max_{I_{1,n}} \prod_{i=1}^{n} P(I_i(blob_i)) \tag{3.9}$$

In order to communicate the description using a theoretically optimal code would require a description length of: $(-log_2(P_{boy})) + (-log_2(P_{dog}))$. For an image containing n blobs as defined above the goal is to choose interpretations $I_1 ... I_n$ so as to minimize the description length:

$$arg \min_{I_{1,n}} \sum_{i=1}^{n} -log_2(P(I_i(blob_i))) \tag{3.10}$$

Finding the most probable interpretation is identical to finding the minimum description length (MDL).

The approach raises two significant issues.

1. How to estimate P.
2. How to find the global MDL.

Neither of these issues is straightforward. In this thesis we depend upon estimating P from frequencies obtained from a corpus. Other methods such as utilizing physical or qualitative reasoning (QR) models are discussed in Chapter 9. For a general mechanism for finding MDL we employ a Monte Carlo sampling scheme.

3.2.3 Monte-Carlo methods

The MDL Agent architecture described in this chapter addresses the need to integrate knowledge at different semantic levels. To understand an image that consists of high level components such as objects and of low level features such as lines and textures we need to integrate different levels of processing. We have already seen in Chapter 2 that Hearsay II achieved this integration of levels using a multileveled blackboard and Schemas achieved

it using a hierarchy of schemas and sub-schemas. However, neither of these approaches offered any insights into how to find the most likely multilevel interpretation.

Single thread of control solutions, such as the blackboard and forward chaining approaches, depend upon taking a path towards a solution and backtracking past failures until a solution is found. The same is true of subsumption. These are essentially depth first search for a solution—not search for the *best* solution. In a robot navigation problem, we may be happy if the robot negotiates the obstacles in the environment and finally ends up at the destination. In interpretation problems, just finding a solution is not good enough. An English sentence can have many plausible parses. Most of them do not make sense. ie: syntactically okay, but semantically garbage.

Markov Chain/Monte Carlo methods (MCMC) have become popular recently in computer vision(Geman & Geman 1984; Hammersley & Handscomb 1964; Lau & Ho 1997) but have been limited to modeling low level phenomenon such as textures (Karssemeijer 1992). In natural language understanding, use of hidden Markov models has been successful as an optimization technique with certain restricted kinds of grammar. Problems that can be described as Hidden Markov Models (HMM) (Baum 1972) can yield efficient algorithms. For example, in natural language understanding, some grammars permit efficient algorithms for finding the most probable parse. Stochastic Context Free Grammars (SCFGs) can be parsed so as to find the most probable parse in cubic time using the Viterbi algorithm (Viterbi 1967). Only the simplest grammars and problems can be solved efficiently in this way, however, and for the more interesting grammars and for more complex problems in general, other techniques must be used. Certainly something as loosely defined as an agent system incorporating semantics from multiple levels would rarely fit into the HMM straitjacket.

Even for natural language processing, finding the best solution can be prohibitively expensive when the Viterbi algorithm can't be employed. In visual processing, with images whose complexity greatly exceeds that of sentences, and which are three dimensional [as opposed to the linear arrangement of words in a sentence], finding the best solution is infeasible. Approximate methods are therefore attractive. Monte-Carlo techniques are very attractive in these situations.

In an ambiguous situation, such as parsing a sentence, in which many [perhaps thousands] of legal parses exist, the problem is to find the parse that is the most probable. If the problem can be defined in such a way that parses are produced at random and the probability of producing a given parse is proportional to the probability that the parse would result from a correct interpretation, the problem of finding the most probable parse can be solved by sampling. If P is a random variable for a parse, the probability distribution function (PDF) for P can be estimated by sampling many parses drawn at random. If sufficiently many samples are taken the most probable parse emerges as the

parse that occurs the most frequently. Monte Carlo techniques use sampling to estimate PDF's. We develop a Monte Carlo parser for a 2D picture grammar in chapter 5. In this chapter we generalize the idea of parsing in order to build an agent system.

Monte Carlo methods are attractive for a number of reasons:

1. They provide an approximate solution to search of search spaces whose combinatorics are prohibitive.
2. By adjusting the number of samples, the solution can be computed to an arbitrary accuracy.
3. Whereas the best solution can not be guaranteed by sampling methods, measuring standard error during the sampling phase allows the number of samples to be adjusted to yield a desired level of accuracy.

It is intriguing to note that there is some support for the idea that it is easy to build Monte Carlo based search with neural systems. Human neurons fire at a frequency proportional to their level of excitement. Neurons that listen to their outputs respond probabilistically in proportion to the firing rate of the input. This is because the firing of an input causes a small cumulative chemical effect which builds until it is noticed. If that point is reached at the same time as other inputs are suitable for excitation of the neuron, it will fire. This shows that it is relatively easy for us to build Monte Carlo systems from more or less standard biological neurons. Biological brains may or may not utilize such techniques to implement search of complex solution spaces. However, the fact that such systems can be constructed, is encouraging.

Of particular interest to us is the observation that search implemented in this way is similar to neural systems that utilize negative feedback—except that no negative feedback is required. Systems involving negative feedback have different attributes from those computed using Monte Carlo sampling. Neural feedback systems are gradient descent methods that may be prone to finding local minima.

Local systems in which the landscape is simple and well suited to gradient descent methods will converge very rapidly with inhibition based systems whereas systems that exhibit complex, discontinuous landscapes with local minima will approximate the global minima using Monte Carlo sampling. In the brain small local structures can involve inhibition whereas more distant functional areas are less likely to have inhibition. This appeals to our intuitions that for highly discontinuous landscapes [like parsing] and fairly high level of functional integration [like agents] that Monte Carlo sampling is appropriate.

In this thesis, we are not interested in comparing models of computation based on neurons; but rather in composing systems of agents in such a way that the most probable global descriptions are produced. This brings us directly to the issue of how to select

among agents that may be applicable at a given place in a computation. We have considered three approaches in the foregoing:

1. The selection is carefully programmed. Hearsay II did this with its focus of control database.

2. Local competition among agents.

3. Monte Carlo sampling to estimate global minima.

3.2.4 Agent Selection Paradigms

Autonomous agents are expected to operate without the intervention of a central control mechanism [such as a focus of control database]. As we mentioned in Chapter 2, one approach to the agent selection problem that has been the focus of considerable attention, is the notion of a market based approach. The idea is that an agent wishes to farm out a subtask to another agent capable of performing the subtask. Agents that are candidates to perform the subtask compete by bidding a price. This often works well, producing efficient solutions. However, two problems arise in such systems:

1. Selecting an appropriate basis for cost computations so that the bidding is fair.

2. Because the bidding is piecewise local, such systems are prone to find local minima and miss the global minima.

Our approach addresses these two problems as follows.

The basis for cost computation is description length. Description length is the correct measurement in an interpretation problem because it captures the notion of likelihood directly: $DL = -log_2(P)$.

Monte Carlo sampling allows us to avoid the problem of finding unwanted local minima.

In the following section, we describe the architectural objects that implement these ideas in the GRAVA architecture. In the third section we highlight the idea that Monte Carlo sampling can solve problems similar to those for which systems of inhibition have been used by demonstrating the agent architecture on a reading problem that was previously solved using a multi-layer inhibition based neural network with downward control by McClelland (McClelland & Rumelhart 1986b). Our approach, however, achieves highly robust results without the use of any downward flow of control.

3.3 MDL Agent Architecture

3.3.1 Interpretation Problems

In vision, as well as in speech, we are interested in building descriptions of images or utterances that represent the most likely interpretations. Usually interpretations are ambiguous. Many optical illusions occur when we are fooled in our selection of ambiguous interpretations and interpret some visual data in a way that does not correspond to reality or are able to see two likely interpretations.

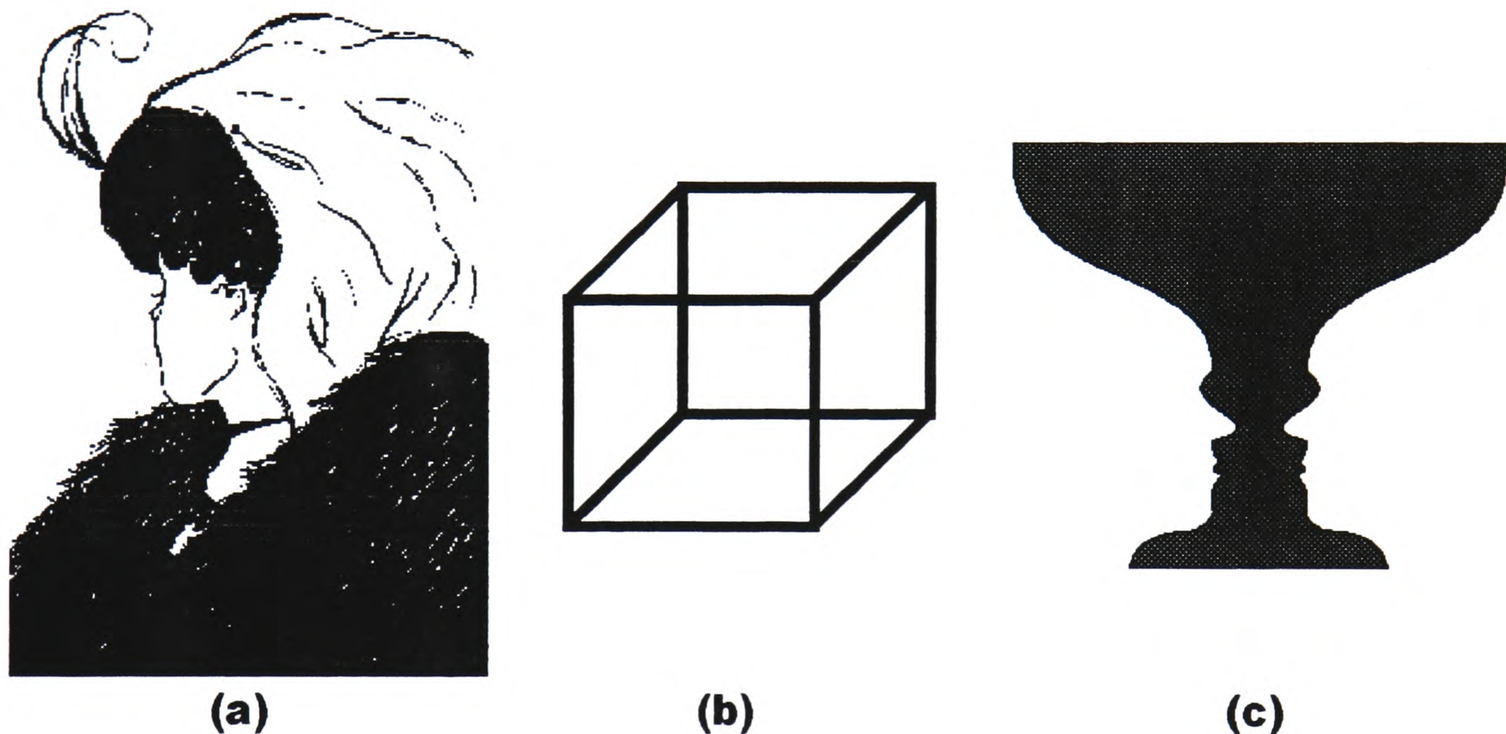


Figure 3.2: Ambiguous Visual Interpretations

Figure 3.2(a) is an example of a standard illusion where there are two almost equally plausible interpretations. The image may initially appear to be a view from the left rear of the head of a young woman but the features have been cunningly arranged by the artist so that we can easily pick an alternative interpretation—that of an old woman. The choice of interpretation, old woman or young woman, sets a context for how the individual features are interpreted. Initially identification of features or of shape helps to select an interpretation. The interpretation then reinforces the view of the features in a way that is consistent with the interpretation. The two alternative views are stable and we can switch between the views with some effort. There are many famous examples of such interpretation ambiguities including the Necker cube[1](Figure 3.2(b)), the face and vase illusion(Figure 3.2(c), and various concave/convex surface illusions. While few complex natural images are ambiguous in such a striking way ambiguity is a common part of most interpretation tasks. Interpreting the real world through sensors is an inherently ambiguous task. Ambiguity exists at all levels of interpretation. The task of the interpretation problem is to find the most likely interpretation. The correct choice

[1]The Necker Cube is named after the Swiss crystallographer Louis Albert Necker

of interpretation of a feature from a set of alternatives is often governed by context. In Figure 3.2(a) the ear of the young woman is interpreted as an eye when the image as a whole is seen as an old woman.

Interpretation problems represent an interesting and important class of problems. Many important problems in AI can be characterized in this way as can problems from other domains. Our architecture is designed to allow complex programs to be structured around the notion of finding the most likely interpretation. Frequently our interpretations at one level affect interpretations at other levels of processing. For example the choice of interpretation of Figure 3.2(a) as an old woman affects the likelihood of interpreting the features contained within it (such as the young girl's ear) and contrariwise. Sound and image fragments taken in isolation are often unintelligible because without context there is not enough information to construct a reasonably likely interpretation of the fragment.

Interpretation in our test application involves:

1. **Interpreting the corpus** The corpus is interpreted to produce a number of descriptions:
    - Texture models to drive the segmentation.
    - Grammar models to drive the image parser.
    - Labeling models to drive the parser.
    - Specification models to drive system behavior.
2. **Interpreting the specification as a program** The specifications are interpreted as a program that builds a description of the image in terms of segmentation, labeling, and parsing of the image. The program consists of a collection of agents connected together to form a program.
3. **Interpreting the image as segments** The image is segmented to produce a representation that divides the image into segments.
4. **Interpreting the image as named regions** The segmented image is interpreted as a set of labeled regions based on their content.
5. **Interpreting the regions as a parse tree** Grammatical rules are applied to the labeled regions to produce an image parse.

Interpreting the image corpus

The highest level in our test domain is the interpretation of an annotated corpus as a specification of interpretation intent. The goal of the system is to interpret aerial

images so as to produce descriptions of the images that are similar to those that would be produced by a domain expert. The manner employed by the system is to imitate by example. A large number of examples are provided by having the domain expert manually produce a description by annotating the image using an annotation tool. The annotation tool developed for this task is described in Appendix C and the image corpora developed for this work are described in Appendix A.

In learning by example/imitation, the student learns the task taught by the teacher in terms of capabilities of the student. While the surface capability may appear similar, the underlying processes may be completely different (Weber, Mataric, & Jenkins 2000; Mataric & Pomplun 1998; Byrne & Russon 1998). When a human teaches a robot to perform some action, such as picking up an object from a conveyor belt, the sensory and motor processes in the human teacher are very different from those possessed by the robot; but the robot learns the surface level task with entirely different lower levels of processing. In our test domain, the task of interpreting an aerial image is learnt in terms of the capabilities that are available within the system and are implemented in the form of agents.

What we mean in this case by "interpreting the corpus as a specification" is to describe the annotations of the corpus in terms of capabilities that we have in our toolbox. The details of those capabilities are elaborated in succeeding chapters but they fall into two main categories: texture and color primitives used by the segmentation algorithm (described in chapter 4); and grammatical rules for the patchwork parser (described in chapter 5).

Interpreting the specification

Interpreting the specification involves building a program out of agents to interpret the image. Program synthesis is covered in Chapter 7. There are many ways of interpreting the specification. The task of this layer is to synthesize the program [which may be thought of as a plan or a program] that is *most likely* to produce the correct interpretation of the image.

Interpreting the program

This level applies the program to an image in order to produce the description of the image. This layer is essentially the program interpreter.

3.3.2 Objects in the GRAVA Architecture

The architecture is built from a small number of objects: Models; Agents; Interpreters; Reflective Levels; and Descriptions.

All of these terms are commonly used in the literature to mean a wide range of things. In the GRAVA architecture they have very specific meanings. Below, we describe what these objects are and how they cooperate to solve an interpretation problem.

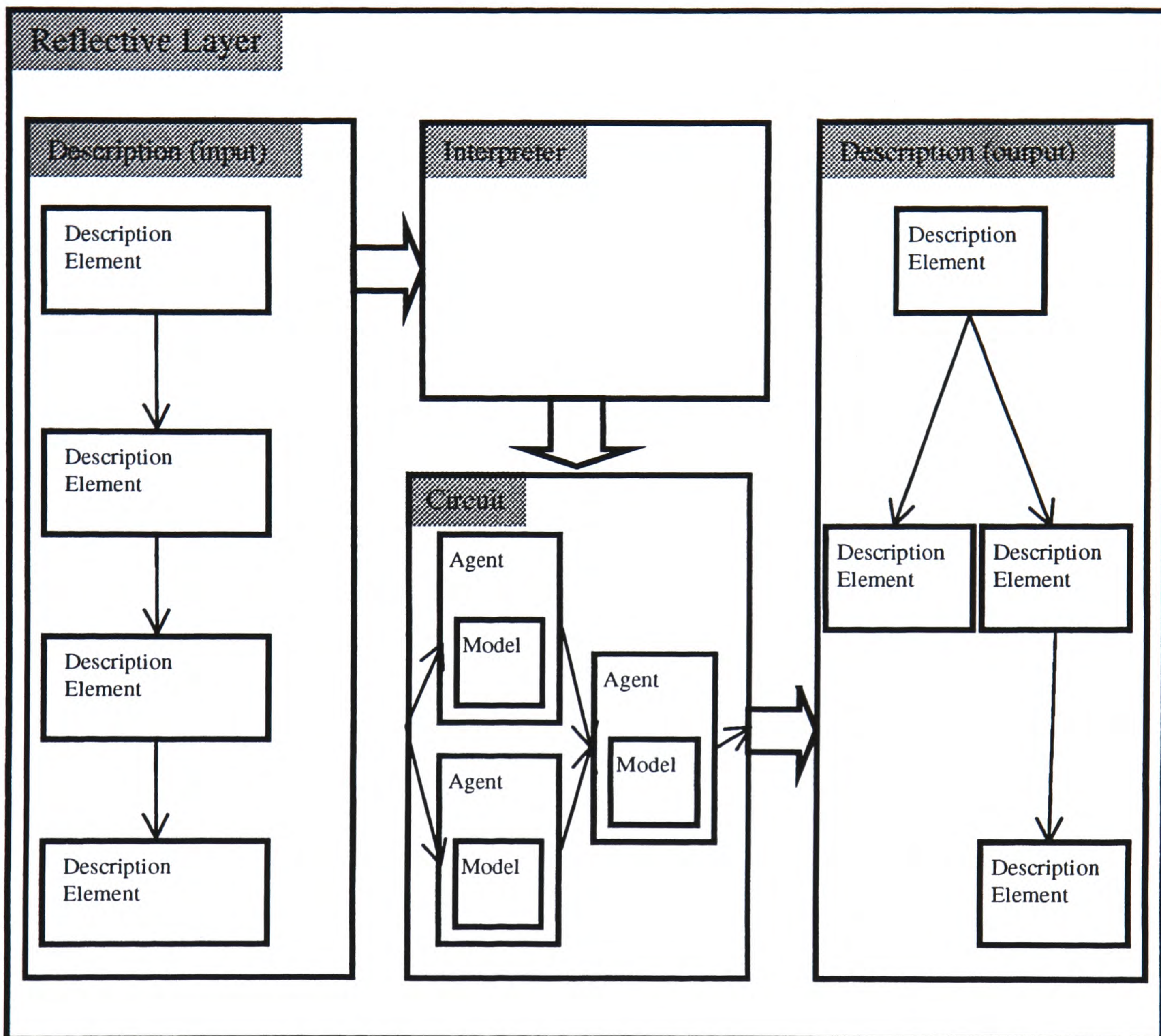


Figure 3.3: Objects in the GRAVA Architecture

Figure 3.3 shows the objects that make up the architecture. A reflective layer takes an input description Δ_{in} and produces an output description Δ_{out} as its result. A description consists of a collection of description elements $< \epsilon_1, \epsilon_2, ..., \epsilon_n >$. The output description is an interpretation $(I \in Q(\Delta_{in}))$ of the input where $Q(x)$ is the set of all possible interpretations of x.

$$\Delta_{out} = I(\Delta_{in}) \tag{3.11}$$

The goal of a layer is to find the best interpretation I_{best} which is defined as the interpretation that minimizes the global description length.

$$arg \min_{I_{best}} DL(I_{best}(\Delta_{in})) \tag{3.12}$$

The interpretation function of the layer consists of an interpretation driver and a collection of connected agents. The interpretation driver deals with the formatting peculiarities of the input description (the input description may be an array of pixels or a symbolic description). The program is made from a collection of agents wired together. The program defines how the input will be interpreted. The job of the interpreter/program is to find the most probable interpretation of the input description and to produce an output description that represents that interpretation.

The GRAVA architecture allows for multiple layers to exist in a program and there are [reflective] links between the layers. The reflective links, how the program is synthesized, and the details of how self-adaptation occurs is covered in Chapter 7.

Below, we describe in greater detail the purpose, protocol, and implementation of the objects depicted in Figure 3.3. We maintain a dual thread in the following. On the one hand, we describe the GRAVA architecture abstractly, on the other, we also describe the actual implementation that we have developed.

Description

A description Δ consists of a set of description elements ϵ.

$$\Delta = < \epsilon_1, \epsilon_2, ..., \epsilon_n > \tag{3.13}$$

Agents produce descriptions that consist of a number of descriptive elements. The descriptive elements provide access to the model, parameters, and the description length of the descriptive element. For example, a description element for a face might include a deformable face model and a list of parameters that deform the model face so that it fits the face in the image. A description element is a model/parameters pair.

The description length must be computed before the element is attached to the description because the agent must compete on the basis of description length to have the descriptive element included. It makes sense therefore to cache the description length in the descriptive element.

The description class implements the iterator:

```
(for Description|des fcn)
```

This applies the function "fcn" to every element of the structural description, and this enables the architecture to compute the global description length:

$$DL(\Delta_{out}) = \sum_{i=1}^{n} DL(\epsilon_i) \tag{3.14}$$

To get a better fix on notation, this is implemented as:

```
(define (globalDescriptionLength Description|des)
  (let ((dl 0))
    (loop for de in des
      (set! dl (+ dl (descriptionLength de))))))
```

DescriptionElements

Description elements are produced by *agents* that *fit models* to the input.

Description elements may be implemented in any way that is convenient or natural for the problem domain. However the following protocol must be implemented for the elements of the description:

```
(agent <Element>)
```

Returns the agent that fitted the model to the input.

```
(model <Element>)
```

Returns the model object that the element represents.

```
(parameters <Element>)
```

Returns the parameter block that parameterizes the model.

```
(descriptionLength <Element>)
```

Returns the description length in bits of the description element.

Implementations of description elements must inherit the class DescriptionElement and implement the methods "agent", "model", "parameters", and "descriptionLength".

Other parts of the protocol for DescriptionElements are introduced in chapter 7.

For readability we print description elements as a list:

```
(<model name> . <parameter list>)
```

Models

Fitting a model to the input can involve a direct match but usually involves a set of parameters.

Consider as input, the string:

```
''t h r e e  b l i n d  m i c e''
```

We can interpret the string as words. In order to do so, the interpreter must apply word models to the input in order to produce the description. If we have word models for "three", "blind", and "mice" the interpreter can use those models to produce the output description:

```
((three) (blind) (mice))
```

The models are parameterless in this example. Alternatively we could have had a model called "word" that is parameterized by the word in question:

```
((word three) (word blind) (word mice))
```

In the first case there is one model for each word. In the case of "three" there is an agent that contains code that looks for "t", "h", "r", "e", and "e" and returns the description element "(three)". In the second case there is one model for words that is parameterized by the actual word. The agent may have a database of words and try to match the input to words in its database.

Consider the two examples above. If the probability of finding a word is 0.9 and the probability of the word being "three" is 0.001 the code length of "(word three)" is given by:

$$DL(wordthree) = DL(word) + DL(three) = -log_2(p(word)) - log_2(p(three)) \quad (3.15)$$

$$= -log_2(0.9) - log_2(0.001) = 0.1520 + 9.9658 = 10.1178bits \quad (3.16)$$

The second approach, in which a separate agent identifies individual words would produce a description like "(three)". The model is "three" and there are no parameters. The likelihood of "three" occurring is 0.001 so the description length is given by:

$$DL(three) = -log_2(p(three)) = -log_2(0.9 * 0.001) = 10.1178bits \quad (3.17)$$

That is, the choice of parameterized vs. unparameterized doesn't affect the description length. Description lengths are governed by the probabilities of the problem domain. This allows description lengths produced by different agents to be compared as long as they make good estimates of description length.

For a more realistic example, consider the case of a principle component analysis (PCA) model of a face (Cootes, Edwards, & Taylor 1998). A PCA face model is produced as follows. First a number n of key points on a face are identified as are their occurrences on all of the images. The shape of the face ψ_i is defined by a vector containing the n points. A mean shape is produced by finding the average position of each point from a set of example face shapes.

$$\bar{\psi} = \frac{1}{n} \sum_{i=1}^{n} \psi_i \quad (3.18)$$

The difference of each face from the mean face $\bar{\psi}$ is given by:

$$\delta\psi_i = \psi_i - \bar{\psi} \tag{3.19}$$

The covariance matrix S then is given by:

$$S = \sum_{i=1}^{n} \delta\psi_i \delta\psi_i^T \tag{3.20}$$

The eigenvectors p_k and the corresponding eigenvalues λ_k of the covariance matrix S are calculated. The eigenvectors are sorted in descending order of their eigenvalues. If there are N images, the number of eigenvectors to explain the totality of nN points is N, typically large. However, much of the variation is due to noise, so that $p << N$ eigenvectors suffices to account for (say) 95% of the variance. The most significant of the eigenvector-eigenvalue pairs are selected as the principal components.

The resulting face model consists of a mean face shape $\bar{\psi}$ and a set of eigenvectors and weights such that any face shape ψ_p can be approximated by:

$$\psi_p = \bar{\psi} + \mathbf{Pb}, \tag{3.21}$$

where $\mathbf{P}$ is the vector of eigenvectors and $\mathbf{b}$ is the vector of weights. The weights are a measure of how much the model must be distorted in order to match the face ψ_p.

The above formulation of a face shape model describes a parameterized model. The weights are the parameters and the mean shape and vector of eigenvectors is the model. Algorithms exist for fitting such shape models to data. These algorithms first identify a key component and then, using the mean shape model, search for the other features (often edges) near the place where the mean suggests it should be. When the feature is found, its actual location is used to define a distance from the mean. This is repeated for feature points in the model. A set of weights is calculated which represents the parameterization of the model.

Agents

The primary purpose of an agent is to fit a model to its input and produce a description element that captures the model and any parameterization of the model.

We implemented the atomic computational elements in GRAVA as agents. The system manipulates agents and builds programs from them but does not go beneath the level of the agent itself. The agent allows conventional image processing primitives to be included in the GRAVA application simply by providing the GRAVA agent protocol. We might have used methods if we were building a language rather than an architecture. GRAVA agents are not autonomous agents. They depend upon other agents to reason about them and to connect them together to make programs.

An agent is a computational unit that has the following properties:

1. It contains code which is the implementation of an algorithm that fits its model to the input in order to produce its output description.

2. It contains one or more models [explicitly or implicitly] that it attempts to fit to the input.

3. It contains support for a variety of services required of agents such as the ability to estimate description lengths for the descriptions that it produces.

An agent is implemented in GRAVA as the class "Agent". New agents are defined by subclassing "Agent". Runtime agents are instances of the appropriate Agent class. Generally Agents are instantiated with one or more models. In our system all models are learned from the corpus.

The protocol for agents includes the method "fit" that invokes the agent's model fitting algorithm to attempt to fit one or more of its models to the current data.

```
(fit anAgent data)
```

The "fit" method returns a (possibly null) list of description elements that the agent has managed to fit to the data. The interpreter may apply many agents to the same data. The list of possible model fits from all applicable agents is concatenated to produce the candidate list from which a Monte Carlo selection is performed.

Interpreters

An *interpreter* is a *program* that applies *agents* in order to produce a structural description output from a structural description input.

A scene interpretation program may include agents for face recognition—such as the PCA face shape agent described above—and may include other agents that recognize other things that would be found in an image such as trees, buildings, and roads. The interpreter could be hand-assembled or it could be generated.

In this thesis we develop the following interpreters:

1. A Segmentation Interpreter.

   The segmenter takes an image as its input description and produces an interpretation as a collage of regions as its output. The segmentation interpreter employs a number of agents that are used for: preprocessing the image to find textures suitable as a basis for segmentation, finding seed points suitable as a basis for establishing regions, and modeling region shapes. The segmentation interpreter is developed in Chapter 4.

2. A Statistical Patchwork Parser Interpreter.

   The patchwork parser takes a set of regions as its input and produces a parse tree as its result. The parser interpreter uses region content agents that assign content descriptors to the regions of the image and a statistical parser agent that applies a grammar model to the regions. The statistical patchwork parser interpreter is developed in Chapter 5.

3. A Model Inducer Interpreter.

   GRAVA is intended to be *grounded* so that the models are derived from real data rather than being constructed by hand. To do this, an annotated corpus is used as the basis for model induction. Shape models, texture models, image content models, and grammatical models are extracted from the corpus during a training phase.

   A model induction algorithm is developed in Chapter 6 that supports model induction from a corpus.

4. A Program Synthesis Interpreter.

   Given the models produced from the annotated corpus, interpreters are synthesized for the segmenter and the parser described above. Program synthesis is described in Chapter 7.

Monte Carlo Agent Selection

A recurring issue in multi-agent systems is the basis for cooperation among the agents. Some systems assume benevolent agents where an agent will always help if it can. Some systems implement selfish agents that only help if there is something in it for them. In some cases the selfish cooperation is quantified with a pseudo market system.

Our approach to agent cooperation involves having agents compete to assert their interpretation. If one agent produces a description that allows another agent to further reduce the description length so that the global description length is minimized, the agents *appear* to have cooperated. Locally, the agents compete to reduce the description length of the image description. The algorithm used to resolve agent conflicts guarantees convergence towards a global MDL thus ensuring that agent cooperation "emerges" from agent competition. The MDL approach guarantees convergence towards the most probable interpretation.

When all applicable agents have been applied to the input data the resulting lists of candidate description elements is concatenated to produce the candidate list.

The *monteCarloSelect* method chooses one description element at random from the candidate list. The random selection is weighted to the probability of the description

element.

$$P_{elem} = 2^{-DL(elem)} \tag{3.22}$$

So, for example, if among the candidates, one has a description length of 1 bit and one has a description length of two bits, the probabilities of those description lengths is 0.5 and 0.25 respectively. The *monteCarloSelect* method would select the one bit description twice as often as the two bit description.

The monteCarloSelect algorithm is given below:

```
(define (probability DescriptionElement|de)
  (expt 2.0 (- (descriptionLength de))))

(define (monteCarloSelect choices)
  (callWithCurrentContinuation
    (lambda (return)
      (let* ((sum (apply + (map probability choices)))
             (rsel (frandom sum)))
        (dolist (choice choices)
          (set! rsel (- rsel (probability choice)))
          (if (<= rsel 0.0) (return choice)))))))
```

3.4 An Illustrative Example of MDL Agents

A key idea of this thesis is that of agent cooperation that spans semantic levels. The idea of semantics affecting lower level processes is not new. In the late 1970's there was much interest within AI concerning heterarchical programming approaches (Fahlman 1973; McDermott & Sussman 1973). These systems suffered from behavior that defied understanding and, equally importantly, analysis. Marr noted that in the human visual system that there was no evidence that higher level processes had any control over lower level vision and that we could profitably study low-level vision without addressing such issues. Computer vision has largely followed that path for the past 20 years. The issue has not gone away however and the notion of how differing levels of semantic interpretation are made to interact is at the very heart of the AI problem. One particularly interesting approach is the parallel distributed processing (PDP) work of Rumelhart and McClelland (McClelland & Rumelhart 1986a). The PDP approach is to develop neural network architectures that implement useful processes such as associative memory. Of particular relevance to the MDL agents described in this chapter is an Interactive activation model for word perception (McClelland & Rumelhart 1986b) developed by Rumelhart.

Figure 3.4 shows the multi-level word recognition architecture that is similar to the example presented later in this chapter. Each plane contains exclusive possibilities for an interpretation. So for example the word "Time" is exclusive of the word "Trim"

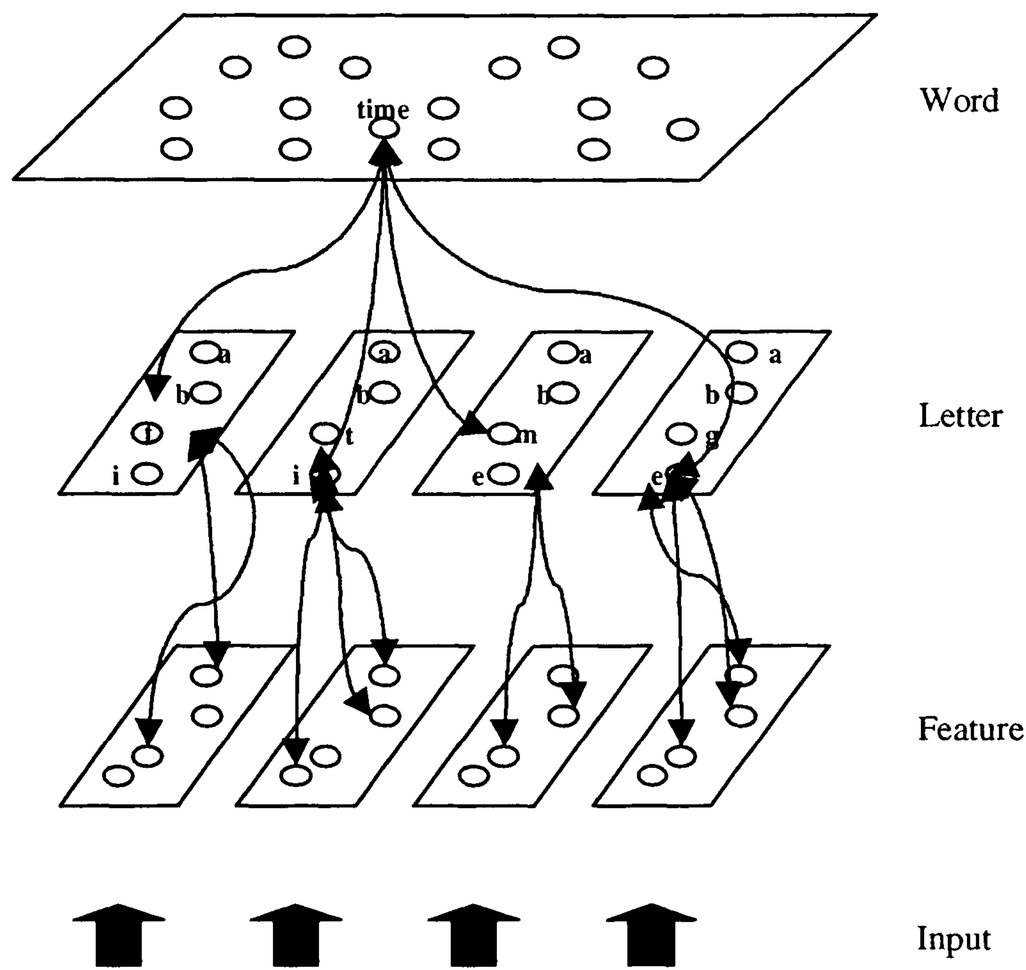


Figure 3.4: Interactive activation model for word perception

and the letter "T" is exclusive of the letter "J". Each possibility on the plane has a mutually inhibitory link to every other entry on the plane. To the extent that "Time" is believed "Trim" is believed less so "Trim" and "Time" mutually inhibit each other. The connections between layers is excitatory. So the letter "T" in the first position of the word is excitatory to both "Time" and "Trim" in the word layer. The links are bi-directional so that the belief in "Time" at the word level is excitatory to "T" at the character level which in turn is inhibitory to all other characters at the character level.

The PDP approach depicted in Figure 3.4 has a number of problems, including how neural systems of this form can be mapped over an input string. Even if these issues were to be successfully addressed, however, the approach throws out the programming model completely and leaves us to develop a completely new computation model based on neuronal systems.

A goal of the GRAVA architecture is to enable systems to be built that exhibit the kind of semantic interaction sought by these earlier systems that:

1. is based on a well understood concept that permits the behavior to be reasoned about [unlike the heterarchical approaches]; and

2. retains a more conventional programming model [than the PDP approach].

The former is achieved by appealing to well understood ideas from communication theory and Monte-Carlo methods, and the latter is achieved by providing an architecture in which the modules of competence are implemented as agents using familiar programming practices.

In this section we demonstrate this by solving the problem that Rummelhart et. al. solved using PDP but within the GRAVA architecture. This serves to clearly illustrate the power of the architecture on a simple problem, which will in turn facilitate understanding and to focus on the problem domain in Chapters 4 and 5 where we develop special purpose agents for aerial image understanding.

3.4.1 Recognizing a Hand Written Phrase

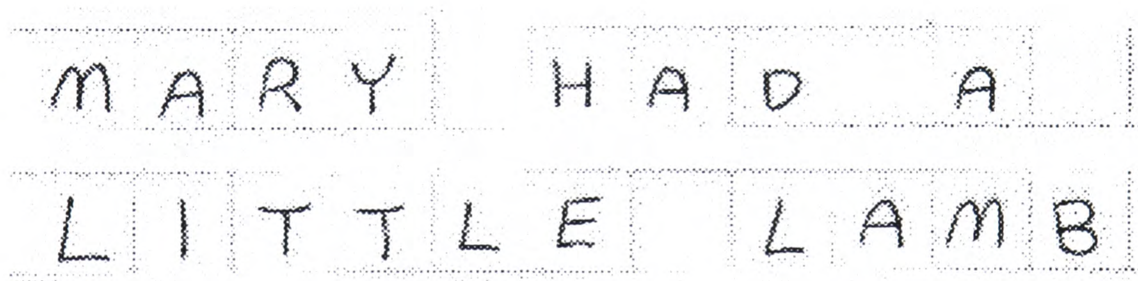


Figure 3.5: Nursery Rhyme Test Data

The experiment described below was trained on a very small corpus and tested on the hand-written phrase "MARY HAD A LITTLE LAMB" which is shown in figure 3.5. Although the test data is simple the system illustrates well the approach described in this chapter and the robustness of the approach. The application is simple but it illustrates issues that arise in the aerial image analysis domain. It served as a test case for our debugging of the implementation of the architecture.

In this example there are three levels of agents. The lowest level agents detect low level features of the written text. There are four feature detectors at this level. Each agent reports on features discovered within a character position.

1. Top stroke endpoints. This agent reports on the number of stroke endpoints at the top of the character. For example the letter 'N' has one stroke endpoint on the top and 'W' has two.

2. Bottom stroke endpoints. This agent reports on the number of stroke endpoints at the bottom of the character. For example the letter 'A' has two endpoints at the

bottom of the character and the letter 'I' has one.

3. Stroke Junctions. This agent reports on the number of line junctions formed from three or more lines. For example the letter 'A' has two such junctions. The letter 'N' has none.

4. Character present. This agent detects whether the character position contains anything at all. Everything but the space character contains something.

The character boxes are segmented into "top" and "bottom" so that the top stroke and bottom stroke features can be determined from a simple line endpoint filter. The system contains two very simple filters that detect line endpoints and complex junctions.

Most uses of semantics in vision are to make up for ambiguity that results from insufficient data. For example, when looking for lines in image data there may be places where the line disappears from the image for a while. We can use various semantic models to "hallucinate" a line onto the data. Example of such approaches include the Hough transform (Hough 1962) and the use of snakes (Kass, Witkin, & Terzopoulos 1987) and bubbles in active vision.

In order to simulate the dearth of low level cues typical of vision problems, we designed this experiment to utilize feature detectors that are by themselves incapable of uniquely identifying the characters in the handwritten text. Even if there is no noise and the feature detectors detect their features with 100% accuracy there is insufficient information using only the features described above to unambiguously identify a character. For example 'S', 'C', 'I', 'L' and 'N' all have one endpoint at the top, one at the bottom, and no junctions.

The next level of agents attempts to build character descriptions from the evidence collected by the feature detectors. The character agent contains a database that has information in the form of a set of models. The models represent evidence from low level sensors and the frequency of the letters in the nursery rhyme. In this case the nursery rhyme acts as a corpus from which statistical evidence is collected and from which the models are constructed.

The word agent attempts to find evidence for words in the input by looking at evidence from the character finding agents. The universe of possible words is derived from the words found in the corpus.

The structure is similar to our agents for segmentation and parsing developed in Chapters 4 and 5. We deliberately mirrored the structure of the aerial image understanding program so that we could debug various aspects of the architecture.

All together the following symbols can be used to build a description of the input:

- *Low level features:* t=0, t=1, t=2, b=0, b=1, b=2, j=0, j=1, j=2, p=0, p=1.

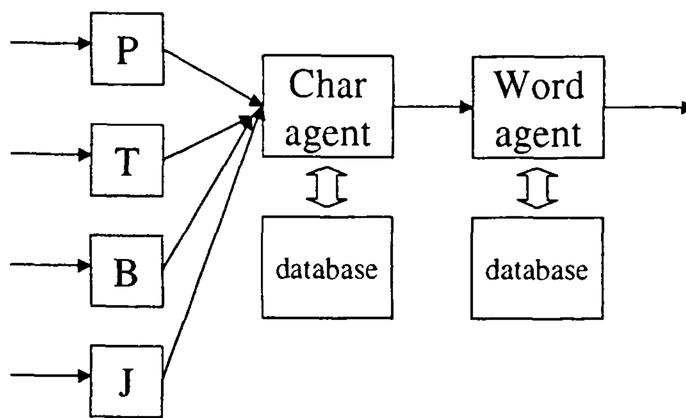


Figure 3.6: Hand Written Phrase Recognizer Program

- *Character level symbols:* A, B, C, D, E, F, G, H, I, L, M, N, O, R, S, T, U, V, W, Y, Space, and
- *Word level symbols:* A, And, As, Everywhere, Fleece, Go, Had, Its, Lamb, Little, Mary, Snow, Sure, That, The, To, Was, Went, White.

The problem can be divided into two phases—each an interpretation problem. The first is to interpret the corpus as models of characters and words for the character and word agents to fit to the image data in the second phase.

Interpreting the Corpus

The character recognizer agent fits character models to the low level data on a per character basis. The word recognizer applies word models to the description produced by the character agent. The description of the corpus as models therefore involves producing two descriptions—in each case a list of models.

The character agent must be able to recognize a character from the low-level features provided and also to estimate the description length. The same is true of the word agent.

Interpreting the Image

Two models are defined for this problem: CharacterModel, and WordModel.

```
;;; Model for characters fitted by the character agent.
(defineClass CharacterModel (Model)
  ((character) ; Character for which model applies
   (template) ; List of image features to match
   (descriptionLength) ; -log2(P(this character in corpus))
   (occurrences)) ; frequency in corpus.
  documentation "Model for characters based on evidence from image features")

;;; Model for words fitted by the word agent.
(defineClass WordModel (Model)
```

```
((word) ; Word for which model applies
 (template) ; List of character features to match
 (descriptionLength) ; -log2(P(this word in corpus))
 (occurrences)) ; frequency in corpus.
documentation "Model for words based on evidence from characters")
```

After analyzing the corpus the following models were available for fitting to the data.

Model	Feature	Template	DL
CharacterModel	Space	(t=0 b=0 j=0 p=0)	4.27
CharacterModel	M	(t=0 b=2 j=0 p=1)	6.73
CharacterModel	A	(t=0 b=2 j=2 p=1)	5.27
CharacterModel	R	(t=0 b=2 j=2 p=1)	6.41
CharacterModel	Y	(t=2 b=1 j=1 p=1)	7.15
CharacterModel	T	(t=2 b=1 j=1 p=1)	5.56
CharacterModel	E	(t=2 b=1 j=1 p=1)	5.15
CharacterModel	F	(t=2 b=1 j=1 p=1)	8.73
CharacterModel	H	(t=2 b=2 j=2 p=1)	6.41
CharacterModel	D	(t=2 b=2 j=2 p=1)	7.73
CharacterModel	O	(t=0 b=0 j=0 p=1)	7.15
CharacterModel	H	(t=2 b=2 j=2 p=1)	6.41
CharacterModel	L	(t=1 b=1 j=0 p=1)	6.41
CharacterModel	I	(t=1 b=1 j=0 p=1)	7.15
CharacterModel	S	(t=1 b=1 j=0 p=1)	6.15
CharacterModel	C	(t=1 b=1 j=0 p=1)	8.73
CharacterModel	N	(t=1 b=1 j=0 p=1)	7.15
CharacterModel	B	(t=0 b=0 j=2 p=1)	6.41
CharacterModel	W	(t=2 b=0 j=0 p=1)	6.41
CharacterModel	V	(t=2 b=0 j=0 p=1)	8.73
CharacterModel	U	(t=2 b=0 j=0 p=1)	8.73
...			
WordModel	A	(A)	8.73
WordModel	AS	(A S)	7.73
WordModel	GO	(G O)	8.73
WordModel	TO	(T O)	8.73
WordModel	AND	(A N D)	8.73
WordModel	WAS	(W A S)	8.73
WordModel	THE	(T H E)	8.73
WordModel	HAD	(H A D)	8.73
WordModel	ITS	(I T S)	8.73
WordModel	LAMB	(L A M B)	7.73
WordModel	MARY	(M A R Y)	7.73
WordModel	SURE	(S U R E)	8.73
WordModel	SNOW	(S N O W)	8.73
WordModel	THAT	(T H A T)	8.73
WordModel	WENT	(W E N T)	8.73
WordModel	WHITE	(W H I T E)	8.73
WordModel	LITTLE	(L I T T L E)	8.73
WordModel	FLEECE	(F L E E C E)	8.73
WordModel	EVERYWHERE	(E V E R Y W H E R E)	8.73

...

3.4.2 Results

the following run shows the result of running this simple example with the six agents described above (as shown in Figure 3.6).

```
=> (runCycles #t)

Description Length=306.979919
Description=(t=0 b=0 j=0 p=0 t=0 b=2 j=0 p=1 t=0 b=2 j=2 p=1
             t=0 b=2 j=2 p=1 t=2 b=1 j=1 p=1 t=0 b=0 j=0 p=0
             t=2 b=2 j=2 p=1 t=0 b=2 j=2 p=1 t=0 b=0 j=0 p=1
             t=0 b=0 j=0 p=0 t=0 b=2 j=2 p=1 t=0 b=0 j=0 p=0
             t=1 b=1 j=0 p=1 t=1 b=1 j=0 p=1 t=2 b=1 j=1 p=1
             t=2 b=1 j=1 p=1 t=1 b=1 j=0 p=1 t=2 b=1 j=1 p=1
             t=0 b=0 j=0 p=0 t=1 b=1 j=0 p=1 t=0 b=2 j=2 p=1
             t=0 b=2 j=0 p=1 t=0 b=0 j=2 p=1)

Description Length=116.690002
Description=(M A A E HAD A   I L T E S T   I R M B)

Description Length=65.699996
Description=(M R A Y HAD   R LITTLE LAMB)

Description Length=61.749996
Description=(M R A E HAD A LITTLE LAMB)

Description Length=41.649997
Description=(MARY HAD A LITTLE LAMB)
```

The agents start out with a description based on the identifications of the low level agents that detect tops, bottoms, junctions, and non-space. The description length is calculated as the sum of the description lengths of each of the symbols in the description and in this case comes to 306.979919. By the next iteration the description length has improved to 116.690002. This has involved identifying characters from the low level cues. As can be seen from the description there are a number of errors in the interpretation due to ambiguity. The word 'HAD' is correctly identified but in the remaining description there are 9 character identification errors out of 18. By the next step matters have improved so that the description length is 65.699996, three words are correctly identified and 3 character identification errors remain. The next improvement brings the number of correct words to four and still has three character errors and finally the sentence is correctly interpreted with a description length of 41.649997 and no errors.

This example while simple illustrates two important properties of the architecture:

1. *MDL formulation leads to the most probable interpretation.* The final interpretation was indeed the correct one.

2. *Global MDL mobilizes knowledge to address ambiguities.* In the second stage of this example fully half of the characters were 'guessed' wrongly by the system. By the fifth stage all of the ambiguous choices had been made correctly because the correct choices were what led to the globally minimum description length.

While the input data is inadequate for correctly and unambiguously identifying the characters, the semantic constraints allow correct identifications to be made. The semantic constraints are propagated by letting competing agents 'win' in proportion to the reduction in global description length. The Monte-Carlo sampling used in this algorithm prevents the system from getting wedged in local minima and ensures that the globally minimum description length (most probable interpretation) is converged upon.

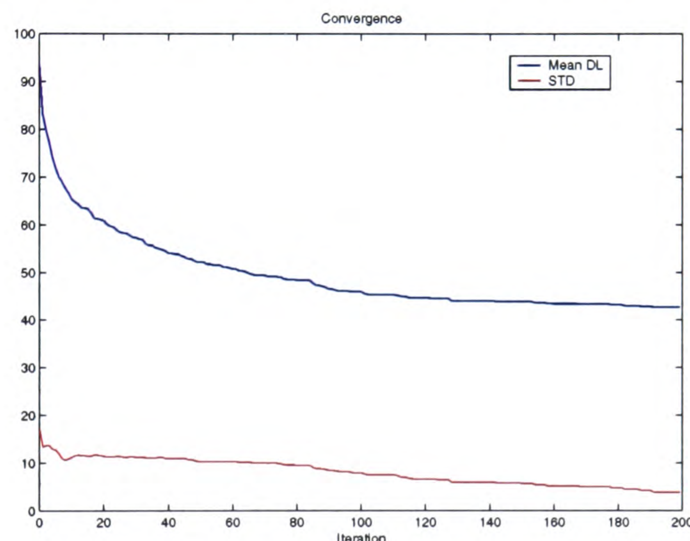


Figure 3.7: Convergence

When the input doesn't make sense, as expected the system is unable to find a good solution. We tried the same example with the characters of each word reversed: "YRAM DAH A ELTTIL BMAL"

```
=> (runCycles #t)

Description Length=306.979919
Description=(t=0 b=0 j=0 p=0 t=0 b=2 j=0 p=1 t=0 b=2 j=2 p=1
             t=0 b=2 j=2 p=1 t=2 b=1 j=1 p=1 t=0 b=0 j=0 p=0
             t=2 b=2 j=2 p=1 t=0 b=2 j=2 p=1 t=0 b=0 j=0 p=1
             t=0 b=0 j=0 p=0 t=0 b=2 j=2 p=1 t=0 b=0 j=0 p=0
             t=1 b=1 j=0 p=1 t=1 b=1 j=0 p=1 t=2 b=1 j=1 p=1
             t=2 b=1 j=1 p=1 t=1 b=1 j=0 p=1 t=2 b=1 j=1 p=1
             t=0 b=0 j=0 p=0 t=1 b=1 j=0 p=1 t=0 b=2 j=2 p=1
             t=0 b=2 j=0 p=1 t=0 b=0 j=2 p=1)

Description Length=128.700012
Description=(T A A M  O A H A  T L E E L L  B M A L)
```

Since the input consists of the same starting characters the starting description length is the same (306.979919). The best interpretation that it is able to generate however correctly identifies 11 of the 18 characters and has a description length of 128.700012. Note that due to ambiguity in the representation it is unable to get all of the characters correct. There are no higher-level semantics to guide it since all of the words spelled backwards are nonsense words that do not appear in the corpus. The resulting description length is more than three times the description length of the meaningful example even though the same characters were used. Poor performance could be detected by the interpretation system by virtue of the description length being larger than normal for a correct parse.

3.5 Conclusion

The architecture described so far is based on two ideas:

1. That for interpretation problems global MDL is the goal;

2. That global MDL can be approximated by Monte-Carlo sampling.

Because MDL is not specific to any particular problem it provides a common currency or "gold standard" for use in market model agent systems. Because the architecture is specialized to solving interpretation problems there are problems for which it is not appropriate. Nevertheless a great many interesting problems can be cast in the form of interpretation problems.

Because the foundations upon which the architecture is built are well understood the behavior and performance of the architecture can yield to some level of analysis—such as convergence analysis. The architecture has some interesting characteristics:

1. Cooperation between agents at different semantic levels is an emergent property of the architecture as was demonstrated by the "reading" example given in this chapter.

2. Robustness within the limits of the programs domain is realized by virtue of measuring how local choices affect global description length.

3. There is an implicit information fusion model.

The third point is worthy of greater discussion. Most attempts at reasoning about uncertainty (Shafer 1976; Mann & Binford 1994; Dempster 1968; Zadeh 1999) attempt to bring together contributions so as to make a local decision. When local evidence is sufficient these methods work well but when the sources of evidence become less straightforward the approaches get bogged down. Numerous problems occur—most notably that

required probabilities are often not available and that in order for the approaches to be tractable it is necessary to assume conditional independence. In any complex system the practical issues are immense. Imagine a program that attempts to combine evidence to interpret the feature that is the young girl's ear in Figure 3.2(a). The information fusion problem must consider what interpretation model is being used and what relationship that has with feature models being considered for the "ear". Locally the interpretation of the "ear" feature as an "eye" or as an "ear" may be quite similar. The choice however effects the interpretation of surrounding features (nose and chin for example). By trying to treat this as a local information fusion problem conventional methods of information fusion potentially demand that the entire structural complexity of the problem be considered at each local decision point.

The implicit information fusion model in the GRAVA architecture depends upon the effect that local decisions have upon the global interpretation. In the reading example described in this chapter information from four different kinds of sensor were utilized in providing evidence in support of interpretations as characters. A less probable interpretation of a feature will be selected if it gives rise to a shorter global description.

The architectures discussed in Chapter 2 (Schemas, Blackboards, Rule Systems, and Subsumption) all attempted to find a single path towards a solution and as we observed the manual hard wiring of control information was one of the major problems with those approaches. In GRAVA we have chosen a method that avoids those problems at the cost of having to pursue multiple paths. When multiple agents are available we try them all and choose one based on a Monte-Carlo sample.

It can be objected that our approach is computationally expensive compared to the approaches discussed in Chapter 2. There are a number of important observations on this issue:

1. The architectures in Chapter 2 all searched for the first solution that could be found. They hoped that by making local decisions a reasonably good solution would result. None of the architectures made any attempt to find the *best* solution. for interpretation problems it is important to find the best solution (or one very close to it).

2. Processing is cheap. It is reasonable to have multiple parallel computations occurring at once. It is reasonable in our quest to build robust vision to have multiple processors, perhaps as many as one per agent or one per model. These are problems that scale well. You really can have one processor per model and derive benefit from the parallelism. There are of course within the single processor view numerous opportunities to build optimization methods. For example for certain problems we can do better than using Monte-Carlo sampling. We have already

noted that for SCFG's that efficient solution can be achieved by using the Viterbi algorithm. We have not pursued these ideas for two reasons.

- We wanted an architecture that was general enough to solve a large number of problems. Chapters 4, 5, and 8 give examples of how this approach works effectively for several combinatorially hard problems.
- Optimization is not a priority in this thesis.

There are a few comments that should be made regarding the use of Monte-Carlo sampling to estimate the global MDL description. Most uses of Monte-Carlo sampling attempt to estimate a PDF by counting frequencies of occurrences. For complex situations such as those described in this thesis (especially the parser described in Chapter 5), a great many samples would be required before anything like an accurate estimate of the PDF would result. In our use we are able to measure the global description length by adding up the description length of the components. Since we only want the most probable solution and not the whole PDF we can get by with far fewer iterations than would be required to estimate the whole PDF with any accuracy. As successive samples are taken we always know which is the best solution so far and can terminate the search at any point. This allows us to trade off interpretation accuracy against computational cost.

In some agent architectures an agent is a wholly autonomous entity that determines its own applicability and negotiates to participate in the ongoing computation. The agents described in this architecture are woven together into a program. The choice of which agents may participate and where has already been made when the program was constructed (at run time). Agents in GRAVA are computational units that support the protocol for agents. The way that the agents are woven together into a program is similar to the way that methods are woven together in CLOS. Chapter 7 describes how programs are synthesized and how the program may self-adapt at runtime as a result of reflection. The MDL approach to agent selection described here should be equally effective in other agent architectures as long as they are solving interpretation problems.

Chapter 4

Semantic Segmentation

4.1 Introduction

In this chapter we describe a segmentation algorithm that is built upon the MDL agent architecture introduced in chapter 3. The algorithm extends work by Leclerc (Leclerc 1989) and by Zhu & Yuille (Zhu & Yuille 1996) which are described below. The algorithm presented here, which we call "Semantic Segmentation", realizes a conjecture of Leclerc that the MDL formulation of the scene partitioning problem could be extended to encompass higher level semantics. We argue that the segmentation problem cannot be done purely as a low level process. Noise and low contrast situations make it hard to achieve perceptually pleasing results robustly with a purely low-level process. In like manner, small regions provide too few points to support any but the simplest statistical models.

Image segmentation is an important step in most image analysis programs; but achieving good segmentations can be as elusive as the subsequent interpretation. Segmentation algorithms are notoriously unstable in the face of changing image quality. We contend, and aim to demonstrate that adding semantics can help significantly. We demonstrate a multi-agent approach to segmentation that generalizes region competition (Zhu & Yuille 1996) in such a way that semantics can be introduced naturally and which permits image interpretation and segmentation to proceed cooperatively instead of sequentially thus improving robustness. Segmentation and interpretation are viewed as parts of a process of producing a description of the image. A minimal description length (MDL) formulation is used to control competition and collaboration among the agents.

In analyzing images, semantic units often correspond to regions, areas of the image that are semantically homogeneous. For example, in an aerial image, a town may be a simply connected region of the image. At a different scale, a house may be a simply connected region. Often, perceptual regions are also approximately homogeneous in a signal characteristic such as color, intensity, or texture. However, this neat picture breaks down for all but the simplest images: cast shadows and highlights may suggest breaking up a region that is semantically unitary, noise may reduce contrast between

regions, and shading may further erode differences between regions making intra-region variation large.

Signal-based techniques face tough challenges in constructing image representations that accord with perceptions as shown by the succession of region finding techniques developed over the past 20 years, none of which work perfectly on images that are slightly removed from the domain for which they were developed, or images that don't satisfy the conditions implicit in the algorithm design. That is, performance degrades disgracefully. We continue to embrace state-of-the-art techniques for signal-based region *segmentation* (Zhu & Yuille 1996; Xie & Brady 1996; Adams & Bischof 1994), but investigate ways in which these techniques can be generalized in such a way as to permit image segmentation to proceed as a cooperative process with the other processes of image interpretation. By allowing semantic considerations to participate in the segmentation of the image, a more robust segmentation and interpretation can result. We don't view segmentation as something that is done once as a low-level preliminary step in the image interpretation process. We suggest instead that segmentation is at first a low-level process that gets the interpretation process started, it is a semantic process that maps interpretation to the underlying image, and it is a result of the image interpretation process. Segmentation is a matter of continual refinement. It is driven from the low level by characteristics of the signal and from the other end by the goal-directed needs of the vision system. This view of segmentation has lead us to architect a segmentation system whose relationship with other parts of the image interpretation system is *cooperative.*

The idea of applying semantics to the segmentation problem, is that if more is known about the image contents, knowledge of the subject matter can influence the manner in which the segmentation is applied. This is de-rigueur in most practical systems. For example, in medical image understanding shape models are fitted to ultrasound images of the heart (Makowski *et al.* 2000).

Segmentation begins with a signal-based approach. When an approximate segmentation has been constructed from signal-based means, agents with a knowledge of textures or shapes (for example) can begin to make hypotheses about region contents. In turn, structural relationships between regions of certain types may allow a higher level interpretation of the image contents. Once agents with a higher level understanding of the image components get involved they can assist in better segmenting the image. Knowing that a region depicts a leaf (for example) allows knowledge of leaf shapes to assist in the segmentation of the leaf boundary even when noise is present or detail is lacking in the image signal. Clearly semantics used in this way can lead to the segmentation (and interpretation) of images that would fail for purely signal-based approaches and in this sense the approach is more robust.

When we know what a feature in an image is, we can know what is salient about the

feature—and therefore what must be a part of the description. What is salient is not just a function of the feature. It is also a function of what we want to do with the description. It is semantics that allows us to decide what can be omitted from an image description. Without semantics, we cannot safely discard any information. Our algorithm is called "semantic segmentation" because it allows semantics to govern what is retained in an image description. Leclerc's list of criteria for image partitioning insisted on a lossless representation. That is because the descriptions had no idea of salience and therefore had to retain everything. Our algorithm and architecture doesn't define what is salient, but provides a framework within which salience can be specified.

The image interpretation problem can be viewed as one of efficiently communicating a description of the salient image content so as to allow sufficient reconstruction of the image for the purposes that the application demands. One way of reducing the description length for communication is on the basis of shared information. If the transmitter and the receiver share knowledge, that shared knowledge need not be part of the transmitted message. If an image contains a chair it is not necessary or useful to transmit every detail of the chair. We can represent it as a "chair". To be useful it is probably necessary to parameterize "chair" to specify enough details for a "useful" reconstruction. "Chair" in this example has three related roles: it is part of the description language, it is image content semantics, and it is a parameterized model. Sometimes it may be structurally decomposed and other times it may be a single level model. What is salient and what can be elided depends upon the goals of the vision system at the time that the interpretation takes place.

Where the "chair" model—or models in general—come from is not important. They can be learned from data, or they can be manually constructed. What is important for our purposes is that there is a mechanism for establishing the probability of the parameterized model occurring in the image given whatever prior information is available.

There is a danger that semantics will cause the segmenter to hypothesize a segmentation that is not supported by the signal and is in fact incorrect. That is, it might have a tendency to hallucinate instances of prior knowledge such as models onto images. However, we readily see a 'man in the moon' or figures in the fire (Minsky 1975). Whenever there is ambiguity (as there usually is) in the interpretation of an image there is a chance that the ambiguity will be resolved erroneously. Here too, semantics comes to the rescue. While it is hard to avoid local ambiguity resolution errors, within a semantic context consistency requirements greatly reduce the number of possible combinations of such errors. By selecting the globally most probable assignment, most of these errors can in practice be avoided.

Marr (Marr 1976; 1982) argued that segmentation was not a well founded problem for a variety of reasons, but principally because it did not seem possible to find low level

processes that could account for the apparent ability of human vision to perceive regions.

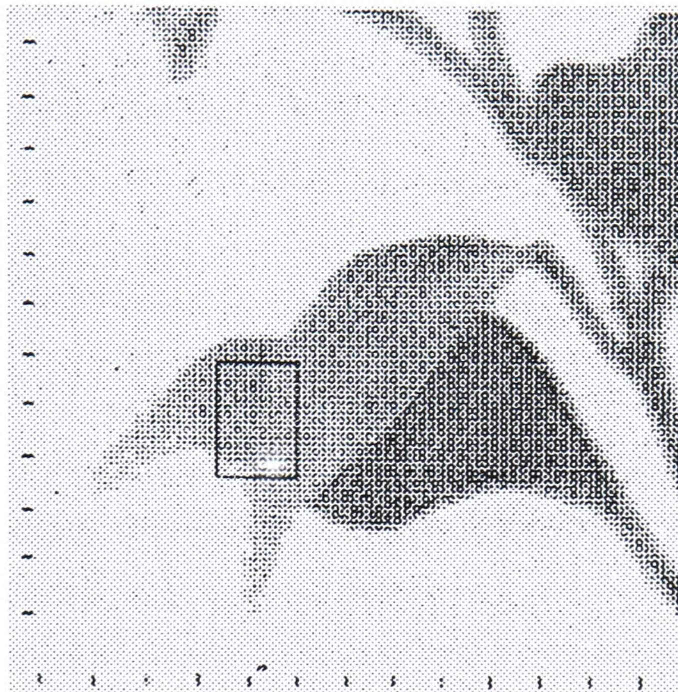


Figure 4.1: Marr's two leaves

He introduced the image of overlapping leaves (Figure 1.2 reproduced for convenience as 4.1) to support his point as follows:

> "This image of two leaves is interesting because there is not a sufficient intensity change everywhere along the edge inside the marked box to allow its complete recovery from intensity values alone, yet we have no trouble perceiving the leaves correctly." Marr.

Marr proceeds to show the pixel values of the inside of the box where it is clear that there is no difference in intensity in some places where we perceive the leaf boundary. In fact, the boundary that we perceive is an illusion. Marr concludes that our ability to segment images is an illusion and therefore not a low level process of human vision. We agree that it is hopeless to try to extract such boundaries as the result of any low level process. Semantics are clearly necessary in order to deal with tough but important cases like this. How semantics can be brought to bear on the segmentation process so as to be able to perceive the leaf boundary is the topic of this chapter.

Leclerc began his 1989 paper (Leclerc 1989) as follows:

> "The partitioning problem is one of the most important unsolved problems in computer vision. In its broadest sense, the task is to delineate regions in an image that correspond to semantic entities in the scene, such as objects and/or coherent physical processes. We shall refer to this as the *scene partitioning problem*. In this sense, the scene partitioning problem is almost isomorphic to the entire image understanding problem and probably cannot

be solved unless a solution to the image understanding problem in its entirety is achieved as well." Leclerc.

This chapter takes up the challenge of building a framework that allows image understanding to proceed *cooperatively* with scene partitioning.

4.2 Prior Work

There are many signal-based approaches to image segmentation.

One approach is to use local filtering such as with an edge detector (Canny 1986) to locate the boundaries between the regions. Such an approach suffers because it only uses information at the boundaries and no guarantee is made that boundaries will be closed.

Closed boundaries can be achieved by snake/balloon methods (Kass, Witkin, & Terzopoulos 1987; Horritt 1999). These methods can be computationally efficient and produce smooth closed boundaries but as with filtering methods they suffer from using information only along the boundaries. Snake/balloon methods require good starting points if they are to converge onto the correct solution.

Seeded region growing (Adams & Bischof 1994; Nair & Aggarwal 1996) uses information within the boundaries as well as along the boundaries but the results often suffer from small holes and boundaries that are overly influenced by noise. Seeded methods share with snake/balloons the problem of having a good starting point. The problem is less acute than with snake/balloon methods but still the quality of the segmentation is strongly influenced by the starting set of seed points.

Texture based relaxation methods, especially those based on Markov Random Fields MRF (Andrey & Tarroux ; Noda, Shirazi, & Kawaguchi), allow pixels to be labeled based on a MRF texture assignment and the pixels can then be grouped to produce a texture segmentation

Global methods based on energy minimization/MDL (Leclerc 1989) overcome some of the limitations of the above methods due to their local processing by providing a global minimization problem for the segmentation of the whole image but they can be cumbersome to work with and finding the global minimum can be expensive.

Hybrid approaches combine the advantages of the above. For example, region competition (Zhu & Yuille 1996) combines the benefits of snakes/balloons for good boundary shape with the benefits of using statistical information about the entire, iteratively developing, region from seeded region growing with the benefits of a global criterion from MDL to produce good overall results. Nevertheless, significant problems remain for producing useful segmentations.

1. A major problem with existing low-level segmentation algorithms is that they build

a poor foundation for higher level interpretation. Often, a poor segmentation necessarily leads to poor performance of the later levels of processing. These approaches are blind to the quality of segmentation that they produce and there is no consideration for the needs of the later stages of the image interpretation process.

2. While the seeded region growing and hybrid approaches utilize information within the region as well as along the boundaries there is no provision for mobilizing contextual information. When humans look at an image and "see" segments they are looking at the image as a whole; but region-based algorithms are expected to form good segmentations on the basis of purely local information. The region-based algorithms ignore the crucial importance of context.

3. Because these methods are inherently low-level, they are based upon low-level constraints about the real world. The cohesive nature of matter gives rise to regions with edges (local filtering). Objects tend to have low local curvature (snakes/balloons). High-level constraints are also important. For example, the recognition of certain shapes and models of occlusion and lighting can in principle play an important part in producing good segmentations in low contrast regions of an image.

In the *cooperative* approach described in this chapter, higher levels of processing can influence the segmentation. We believe that the tighter bi-directional interaction between higher level semantics and image segmentation should lead to more capable and robust image understanding systems.

4.2.1 Constructing stable descriptions

Leclerc (Leclerc 1989) describes an image segmentation algorithm based on the idea of MDL. The segmentation algorithm itself has been improved upon by Zhu & Yuille and is discussed below.

Leclerc defined the image partitioning problem as finding the best description of the image and enumerated a set of criteria for producing the best description. Briefly these criteria are:

1. The description must be complete. The description must determine an image uniquely.

2. The description language and the descriptions themselves must be computationally feasible.

3. The best description must be stable. Small changes in the image should result in small changes in the description.

4. The description should be efficient–it should not be larger than the image.

To partition an image, Leclerc defines a descriptive language. The descriptive language contains elements that allow the image to be described in its entirety. The language describes the application of models to the image. The models describe the shape and contents of regions. If a region fits a model well, its description length will be less than if it fits poorly. When the entire image is described as a patchwork of regions, the description that has the shortest description length is deemed to be the best.

The success of the approach is largely governed by the applicability of the models. Leclerc's approach to this is to consider the contributing components to the image formation process to be separate parts of the description. There are many contributions to the image formation process; some of the more obvious ones are enumerated below:

1. **Noise:** When an image is captured by a sensor, the signal is corrupted by a number of sources including: atmospheric particles and imperfections in the lens and sensor. The noise component of the image is usually random and often modeled effectively as a statistical distribution. Often, CCD camera noise is Gaussian or a sum of Gaussian processes.

2. **Lighting:** The appearance of physical surfaces is affected by the lighting of the surface. A single light source near one side of a flat table may cause the brightness of the table surface to vary so as to be lighter near the lamp and darker towards the opposite end. Lighting also causes shadows.

3. **Surface reflectance:** The surface itself interacts with the lighting. A smooth matte (Lambertian[1]) surface will produce a different effect from a surface with a high specular component. A surface with a mixed or grainy consistency will produce a textured region in the image.

4. **Geometry:** The positioning of objects in the scene affects the image formed on the sensor. Noise effects may be greater for more distant objects, spatial configuration may generate occluding boundaries, and textures will smoothly change their scale as they recede into the background.

The signal received from the sensor is thus a complex composition of the above, and other, processes. Fitting lighting, geometry, reflectance, surface, and noise models to the composite image might provide a way of describing the image more simply. Regions

[1]A Lambertian surface is a surface of perfectly matte properties, which means that reflected or transmitted luminous intensity in any direction from the surface varies as the cosine of the angle between that direction and the normal vector of the surface. As a consequence, the luminance of that surface is the same regardless of the viewing angle.

of consecutive pixels that are governed by the same set of models would constitute a low-level partitioning of the image upon which higher level interpretation could build.

Leclerc develops two models for image decomposition that deal with surface and noise. So, for example, if z is the image to be partitioned, it can be considered as being formed by an underlying image u and a noise term r such that $z = u + r$. The problem involves dividing the image into regions and fitting the noise and surface models so as to find the most likely values for u and r that account for the image.

Leclerc's models differ in the kind of underlying surface that they model. Leclerc's first model assumes that the underlying image consists of regions that are homogeneous. His second model assumes that the underlying image is piecewise smooth. He makes no attempt to model lighting, surface reflectance, or geometry.

The piecewise smooth model is more robust because it allows some of the effects of geometry, lighting, and surface reflectance to fall within the piecewise smooth characterization. The piecewise smooth model effectively divides the physical image process into "noise" and "everything else".

Within this family of approaches, the most likely interpretation is found by characterizing the problem as an MDL problem and using an iterative algorithm similar to graduated non-convexity (GNC) (Blake & Zisserman 1987). While this algorithm approximates a global minimum description length it is very cumbersome to define the model equations in such a way that the GNC-like method can be applied.

4.2.2 Region Competition

The region competition algorithm of Zhu & Yuille depends upon a description language that incorporates outline curvature smoothness and upon the "energy" of the region. The algorithm is parametric in that the relative importance of outline smoothness and region energy is controlled by a parameter. An iterative algorithm is described that converges on a (local) description length minimum. Because the algorithm cannot guarantee a global minima the choice of seed points is crucial.

Region competition begins by selecting a set of seed points. Each seed point establishes a small region around the seed point which it then attempts to grow. The outcome of the segmentation depends upon selecting the right seed points.

The regions once established grow by expanding the boundary of the region to capture pixels that are more like the region than the background region based on a suitably chosen statistical model. Region growing continues until region boundaries bump into each other.

The regions that have adjoining boundaries attempt to steal pixels from their neighbors if the pixels are more "like" themselves than the region to which they are currently assigned. Region competition continues until there are no more opportunities for pixels

to be reassigned.

When two regions meet that are similar, it often reduces the global energy if the regions are merged to produce a single region.

The manner in which the outlines of the regions are expanded and in which region competition takes place involves two mechanisms. The first is the "statistical force" acting upon points along the boundaries of a region. The statistical force is calculated by considering whether the pixels along the boundary are more like the adjacent boundary (an outward force) and the other region's pixels are more like this region (an inward force). The second is the smoothing force which attempts to produce a smooth boundary by creating a force that tries to move pixels on the boundary in a way that reduces curvature. The combination of these two forces controls the attempt to move the region boundary and capture/relinquish pixels. The energy is computed for each of the regions and the global energy is computed for the entire region by adding the energy of each of the individual regions. Region competition continues until after some (arbitrary) number of iterations the global energy hasn't reduced.

The energy function $E[\Gamma, \{\alpha_i\}]$ where Γ is the collection of all region boundaries in the segmented image and $\{\alpha_i\}$ is the set of parameters for the distributions of each region in the segmented image. The image is segmented into a set of M regions $R = \bigcup_{i=1}^{M} R_i$.

$$E[\Gamma, \{\alpha_i\}] = \sum_{i=1}^{M} \left\{ \frac{\mu}{2} \oint_{\delta R_i} ds - logP\big(\{I_{(x,y)} : (x,y) \in R_i\} | \alpha_i\big) + \lambda \right\} \qquad (4.1)$$

μ is the length of the all region boundaries (divided by 2 because the boundaries are counted twice). λ is the cost of representing the distributions of the regions.

The region competition algorithm produces fairly good outlines due to the smoothing force however the smoothing force appears less effective than the authors expected. The regions are based on homogeneous regions not piecewise smooth regions. That gives rise to unstable boundaries.

4.2.3 Shortcomings of Low-Level Methods

The image formation process is complex. Trying to model the process analytically gets hard very quickly. Region competition didn't bother even with a piecewise smooth model even though it would fit the combination of unmodeled parts of the image formation process better than the homogeneous region model. Problems cited in the introduction can easily throw these algorithms off and there is no obvious way to recover from failures of the algorithm making the image interpretation process that depends upon it brittle.

Once these approaches are started, they can't be stopped. They ultimately emit a segmentation solution that you can take or leave. If the segmentation is poor, later

processing stages will fail, and there is little that can be done to improve the segmentation short of rerunning the algorithm with different seeds and parameters.

4.3 Semantic Segmentation

As noted earlier, the view of segmentation taken by this thesis is that, contrary to the common view that segmentation is a low-level process, we take segmentation to be a cooperative process with all levels of processing. Where other algorithms pick *a priori* methods for determining things like curvature characteristics, we let such things be driven by the semantics. Rather than trying to produce an objectively good segmentation and then hoping that the later levels of processing can do something useful with the result, we take the view that the nature of the segmentation should be controlled by the later levels of processing so that the resulting segmentation *is* in the form expected by later processing stages.

The image interpretation problem is viewed from the communication theory view as one of intelligent compression. We wish to throw away all information that we are not interested in and then to package up the result in such a way as to minimize the message length. Most image compression algorithms decide what to retain on the basis of being able to reproduce a good approximation to the original image as perceived by a human viewer. The lossy compression may, however, throw away some crucial detail for certain tasks even though the reconstructed image looks like a close approximation to the original. Vision has historically been viewed as a data reduction process. To be successful, the data reduction process must be intelligent. The salient information in the signal must be maintained and the rest should be discarded. "Saliency" can sometimes be defined by low-level techniques due to certain physical properties of the world and the physics of the image formation process. Edge detectors are one form of saliency detection that derives from basic properties of the world. At some level, however, saliency can only be defined by semantic properties.

We are less interested in image reconstruction for human viewing but would like to reconstruct a semantic rendition of the image for whatever purpose the interpretation system is engaged in accomplishing. For example, a robot that is attempting to negotiate its way through a cluttered environment will want to know where the obstacles are but may not need to retain enough detail to be able to identify what the obstacles are. When the robot reaches its destination and has to find and pick up a widget it may need a different kind of segmentation that retains different kinds of detail appropriate for identifying widgets.

Compression algorithms like MPEG (ISO 1995; 1992; Lee *et al.* 1994) are designed as architectures that permit special purpose methods to be plugged in to achieve the

compression. Our approach to image segmentation is similar. We provide a basic low-level base level segmenter—a knowledge-free segmenter—that makes few assumptions and discards nothing. We also provide an architecture that permits semantic agents that cooperate with the segmenter to be plugged in.

The choice of which agents are used in performing the segmentation, and what inputs are made available to the segmenter, determine the nature of the resulting segmentation. Those choices are made by the theorem prover/compiler in the self-adaptive architecture described in chapter 7. The choice is made such that the MDL program results. The MDL program is the program most likely to succeed given what is known about the state of the world at that point in time.

Describing an image in terms of regions naturally breaks down into three main representational issues: representing the region sizes and shapes; representing the contents of the regions; and representing the relationship between the regions.

Size and shape includes the characteristics of the outline of the region. Different regions will want different outlines depending on semantic attributes. For example, a leaf outline might have a smoothly curved outline, or an outline with a number of points of high curvature, depending on the leaf type. Low level algorithms such as Zhu & Yuille (Zhu & Yuille 1996) prefer smooth outlines for all region types.

4.3.1 The Semantic Segmentation Base Algorithm

We recast the region competition algorithm as cooperative (or competing) agents using the architecture described in chapter 3. In this formulation, the image is described as a collection of regions. The collection of regions describes the entire image. The description of each region is produced by a base segmentation agent. The base segmentation agent describes the region in terms of the bounding contour and a description of the region's contents.

Initially, the image is segmented as background and seed regions. The agents then compete on behalf of their respective regions for pixels along their borders. If the image description length can be reduced by giving a pixel to a neighboring region the exchange will happen. In this view, the algorithm consists of agents negotiating with each other for pixels along their common borders. The basis for whether a negotiation for a pixel succeeds depends upon whether the total image description length (global energy) is decreased by the exchange. If r_1 and r_2 are the regions before a pixel is moved from one region to the other and r_1' and r_2' are the regions after the pixel has been moved to another region, the change in global description length is given by:

$$DL(r_1') - DL(r_1) + DL(r_2') - DL(r_2) \tag{4.2}$$

A separate region merging agent continually tries to reduce description length by merging regions that share a common border. A third kind of agent is responsible for placing the initial seed regions.

Hence the semantic segmentation algorithm is implemented as three kinds of agent. One for seed point placement, one for boundary contraction/competition and one for region merging. Implemented this way, the algorithm performs in a way that is very similar to the way that Zhu/Yuille region competition does; but now there is an opportunity to reduce the description length even further by applying semantic models. For example, by identifying the contents of the regions as (say) urban, it is possible to characterize the distribution of the region in terms of a database of urban areas. This is analogous to reducing the description length by representing the word "MARY" instead of the characters "M" "A" "R" and "Y" in the reading example described in Chapter 3. By adding agents that attempt to recognize region contents, the algorithm continues as long as better labelings of the regions can be accomplished. Identification of a region as a leaf of a particular tree may enable the boundary of the region to be described more compactly also. Hence agents can be added that offer shape descriptions based on databases of parameterized shapes. The reduction in description length that results from the successful application of these agents to labeled regions causes the segmentation of the image to converge to regions that fit well with what is known by the added semantic modules.

Semantics are represented by models which are fitted by agents as described in Chapter 3.

An agent has the responsibility of having regions of the image represented with a minimum description length. To achieve this, once an agent instantiates a representation of a region of the image, it begins to search for other agents that can represent the region with a lower cost description.

We want higher level semantics to influence segmentation but we don't want every interpretation agent to have to have built-in support for segmentation. We prefer to have a general segmenter that knows as much as is necessary about maintaining a segmentation and as little as possible about built-in assumptions that lead to a segmentation. For this reason, in order to ground the cooperative process between segmentation and interpretation, we define a base-level segmenter whose sole task is to get the process started.

Figure 4.2 shows the flow of the semantic segmentation algorithm. Initially a single region is initialized that contains the whole image. Its description length is computed based on the null semantics representation of the base segmenter (described below).

The first step (*1) is to attempt to find regions that can be introduced that can reduce the description length. One way is to just have a collection of seed points. If

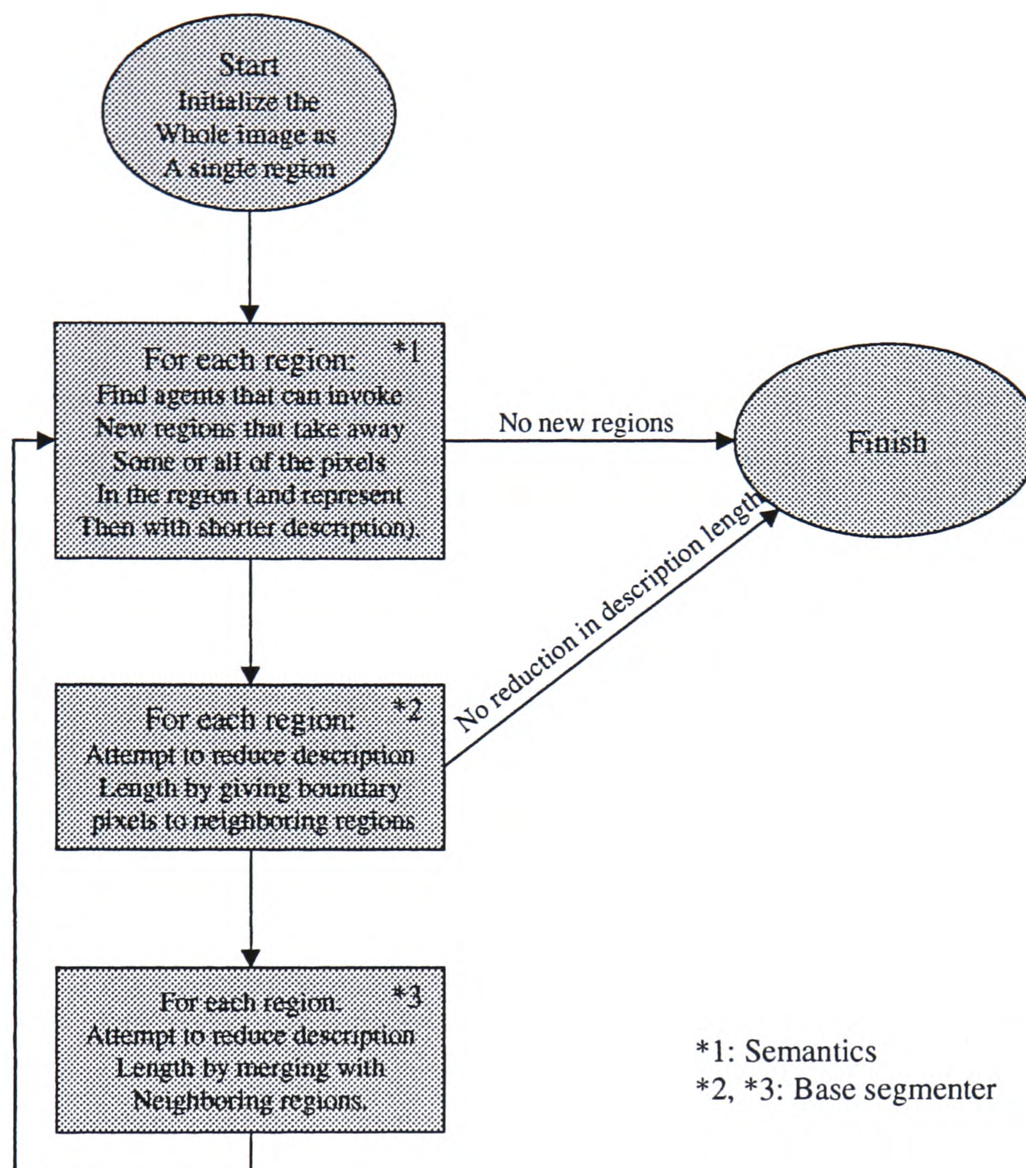


Figure 4.2: High level schematic of the semantic segmentation algorithm

no such opportunities can be found the algorithm terminates. New regions can be null semantic regions (like the original region) or can be semantic regions.

In the second step (*2) each region attempts to reduce the description length by giving away boundary pixels to neighboring regions. This is repeated until no more reduction can be achieved by giving pixels away to neighbors. If no reduction occurred during this step the algorithm terminates.

The third step (*3) attempts to reduce the description length by merging neighboring regions. When neighboring regions are merged the total boundary cost is decreased but the cost of representing the internal pixels may increase if the regions are not similar.

The above three steps are repeated until the algorithm terminates in step *1 or step *2.

Step *1 corresponds to seed point selection when the new regions are null semantics base level regions.

Step *1 introduces semantics by allowing agents that implement semantics to reduce the description length.

Steps *2 and *3 implement the base level segmenter.

We begin by describing the base level segmentation algorithm and demonstrate how it performs without any additional semantics. Then we describe the protocol for semantic agents and show how image interpretation and segmentation can proceed cooperatively.

4.3.2 Base Level Segmentation

The problem of defining the base segmenter is to define a compact and useful representation of the image in terms of regions. The purpose of designing the message format is to enable us to compute the message length that would be required to communicate the image. As with all uses of MDL in this thesis, we don't actually need to construct the message—or to communicate it[2].

The input pixels z are to be represented as an image im as n regions $R_0..R_{n-1}$. We begin the image representation with n, the number of regions. In order to represent a region, it is necessary to describe the pixels that the region contains. One way to do that is to represent the boundaries of the region. Given the boundaries of the region, the pixels can be described in order from top left to bottom right staying within the boundaries.

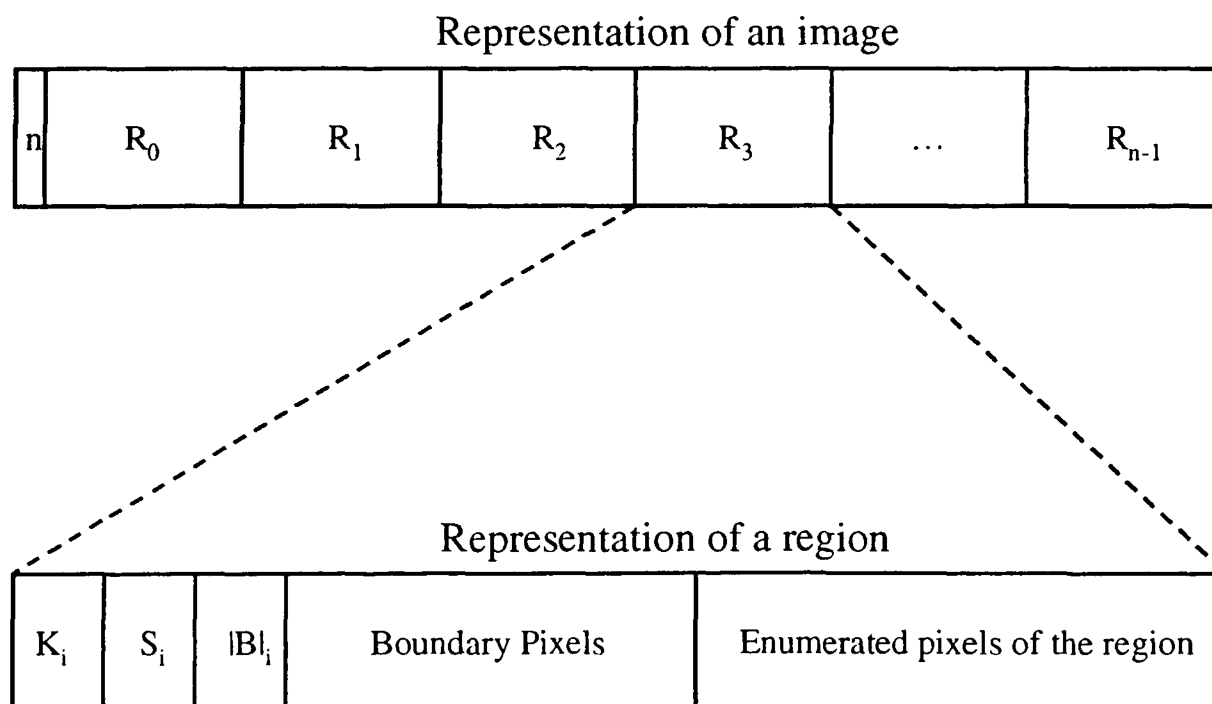


Figure 4.3: Base Description Language for Images

We may have many different approaches to representing regions so we first represent the kind K_i of region R_i.

[2]However, semantic MDL image reconstruction offers interesting possibilities that are discussed in Chapter 9

$$DL(K_i) = -log_2 P(K_i) \tag{4.3}$$

The description length for a given region type will depend upon the model for that region type. We begin by specifying the representation for the base region type that makes no assumptions about the shape or contents of the region.

It is not necessary to represent the number of pixels in the region because the number is uniquely determined by the boundary. We begin by describing the external boundary of the region. We define an (arbitrary) starting pixel S_i as the starting boundary pixel for region R_i. With no other knowledge about where the starting pixel is, the probability of the pixel beginning in any location depends upon the size of the image. If the number of pixels in the image is given by x,

$$DL(S_i) = -log_2 \frac{1}{x} \tag{4.4}$$

We next specify the number of pixels $k = |B_i|$ that comprise the boundary.

$$DL(|B_i|) = -log_2 P(|B_i| = k) \tag{4.5}$$

Then for each boundary pixel in order we describe the position as a relative move from the previous pixel $M_{i,j}$. There are 7 possibilities at each step (a boundary cannot double back on itself) the length move can be represented as:

$$DL(M_{i,j}) = -log_2 P(M_{i,j}|M_{i,j-1}, M_{i,j-2} \ldots, M_{i,j+1}, M_{i,j+2}, \ldots) \tag{4.6}$$

The conditional probabilities can be learned from a corpus or representative boundaries. This is a knowledge-free estimate of smoothness constraints for the boundary of the region. Making the representation too sparse requires too many outlines to obtain a good estimate. We use the following in our implementation:

$$DL(M_{i,j}) = -log_2 P(M_{i,j}|M_{i,j-2}, M_{i,j-1}, M_{i,j+1}, M_{i,j+2}) \tag{4.7}$$

This determines the position and shape of the region as well as the number of pixels in the region. It also allows us to define a unique ordering of the pixels. Figure 4.4 shows how the region boundary is described as a chain starting from an arbitrary starting boundary pixel and how the pixels themselves can be described in scan line order starting from the top left pixel.

Given such an ordering all that remains is to represent the individual pixels in the region.

Our intuitions about regions are that they are homogeneous or smooth. If they are homogeneous, all the pixels of a region are taken from the same distribution. We can

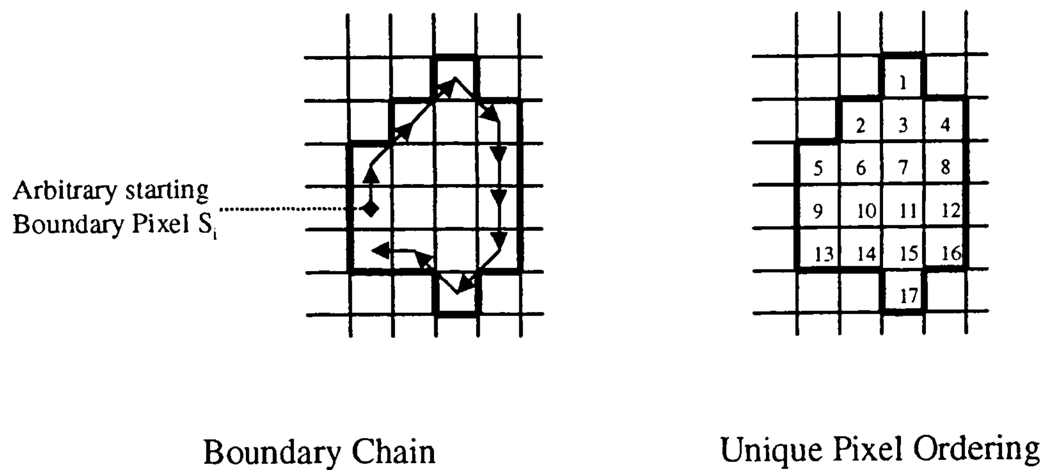


Figure 4.4: Representation of Regions

estimate that distribution by building a histogram of the pixels in the region. Then the coding length of a pixel whose intensity is q is given by:

$$DL(q) = -log_2 P(q|region = R_i) \tag{4.8}$$

To summarize:

$$DL(im) = -log_2 P(|im| = n) + \sum_{R \in im} -log_s P(K_R) + DL(R) \tag{4.9}$$

The description length of R depends upon the kind of region. For the homogeneous base region discussed so far, the description divides into two parts: the description of the boundary B and the description of the individual pixels contained in the region C.

$$DL(R_i) = DL(B_i) + DL(C_i) \tag{4.10}$$

Where

$$\begin{aligned} DL(B_i) = &- log_2(\frac{1}{x}) - log_2 P(|B_i| = k) + \\ &\sum_{j=0}^{k} -log_2 P(M_{i,j}|M_{i,j-2}, M_{i,j-1}, M_{i,j+1}, M_{i,j+2}) \end{aligned} \tag{4.11}$$

and

$$DL(C_i) = \sum_{l=0}^{|C_i|} -log_2 P(q|region = R_i) \tag{4.12}$$

If the pixels of the region are all the same, $-log_2 P(q|region = R_i) = 0$, which means that the pixels don't have to be included in the message (because they are all the same).

The assumption that regions are smooth is slightly different. It says that the probability that a pixel belongs to a region depends upon where in the region the pixel lies.

A ball may form a region whose illumination is brightest at the point of maximum reflection and darkest towards the edges. In spite of significant change in intensity across the region it is piecewise smooth at all points. There are no edges in the ball so we might reasonably expect it to be segmented as a single region. With the homogeneity assumption (above) the ball would be segmented into arbitrary regions in order to produce regions with small variances. This violates Leclerc's third criteria—that the description should be stable. The boundaries between the arbitrary regions are unstable and reflect idiosyncrasies of the original seed locations and the region boundary adjustment algorithm rather than the image itself. Since the regions thus created and the boundaries between them are arbitrary the piecewise smooth description seems better.

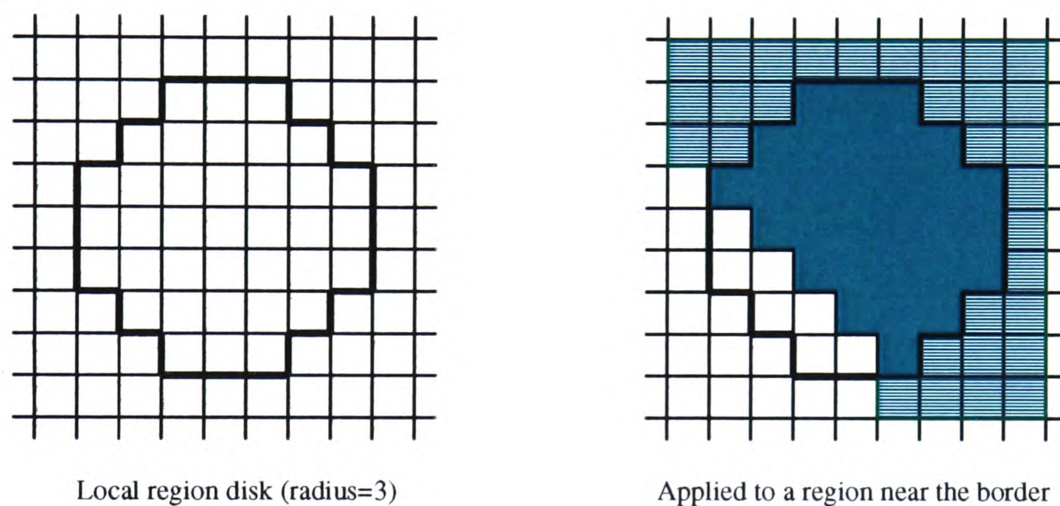


Figure 4.5: Disk for Local Piecewise Smooth Region

In order to compute the local mean, a circular disk shape around the pixel to be described is used. The pixels that fall in the region, and are within the disk, are used to calculate the local mean (a circular window 2D moving average). Figure 4.5 shows a disk with radius 3 (left) and the disk superimposed on a region. The lightly shaded pixels are those that belong to the region and the dark shaded pixels are the region pixels that fall within the circular window.

A region therefore doesn't have a single distribution, rather it has a PDF with a single shape but with a mean that varies smoothly throughout the region. For every pixel $q_{x,y}$ in the region the local mean $m_{x,y}$ is computed by centering the disk over the pixel and then computing the mean of all pixels that are both within the region and within the disk. Then the deviation of the pixel from the mean $d_{x,y} = m_{x,y} - q_{x,y}$ is calculated. The pixel is then represented in two parts: the local mean $m_{x,y}$ and the deviation $d_{x,y}$. Hence the description length for each pixel $q_{x,y}$ is given by:

$$DL(q_{x,y}) = -log_2 P(m_{x,y}|region = R_i) - log_2 P(d_{x,y}|region = R_i) \quad (4.13)$$

The description length of the region contents therefore becomes:

$$\begin{aligned} DL(R_i) = &- log_2(\frac{1}{x}) - log_2 P(|B_i| = k) + \\ &\sum_{j=0}^{k} -log_2 P(M_{i,j}|M_{i,j-2}, M_{i,j-1}, M_{i,j+1}, M_{i,j+2}) + \\ &\sum_{l=0}^{|C_i|} -log_2 P(m|region = R_i) - log_2 P(d|region = R_i) \end{aligned} \tag{4.14}$$

If the mean remains constant throughout the region $-log_2 P(m|region = R_i) = 0$, that means that the representation of the region contents will be identical to the homogeneous case developed above. Similarly if the pixels in the entire region are identical the deviations will all be the same too so $-log_2 P(d|region = R_i) = 0$. Consequently, as in the homogeneous case, if all of the pixels are the same they don't need to be individually communicated. There is therefore no additional cost associated with representing the region as piecewise smooth. We implemented both the homogeneous and the piecewise smooth versions of $DL(C_i)$ and found that the piecewise smooth case gave significantly better results. The disk radius used in our implementation was 5. There is a tradeoff to be considered in the disk size choice. If the disk is too large it tends towards the homogeneous case with its disadvantages. If the disk is too small the number of pixels that contribute to the local mean computation is too small to estimate the mean accurately and hence the cost calculation becomes inaccurate.

4.3.3 Changes in Region Topology

When a region cannot be represented as a contiguous region because the boundary has contracted so as to touch itself the region is split into multiple regions each independently contiguous.

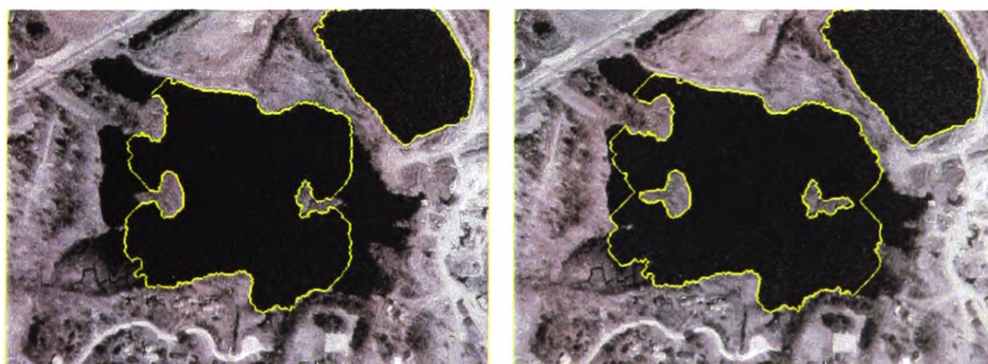


Figure 4.6: Region Splitting

Figure 4.6 shows region splitting in action. On the left, a region is growing around the two islands. At this point the islands are part of the background region. Twenty iterations later the growing (lake) region has expanded past the islands and the islands have split away from the background and are now separate regions.

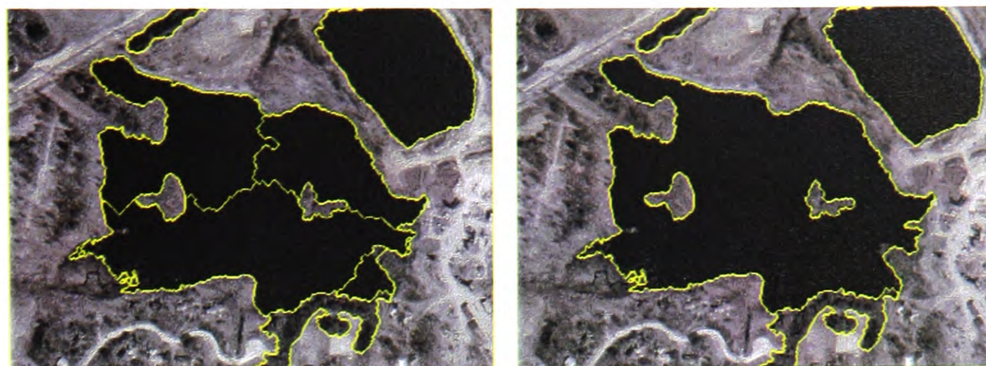


Figure 4.7: Region Merging

Figure 4.7 shows region merging in action. On the left a number of regions have grown to accommodate the lake. In the next iteration (the right image) the regions internal to the lake have been merged. It is more compact to represent them as a single region than as separate regions.

4.3.4 Color and Multi-plane Segmentation

In a color image the distinction between different regions may be more apparent in one color plane than another. Two regions may have the same red component but have very different green components. Instead of RGB, we can divide the image into CMY planes or HSI planes. The number of planes doesn't have to be three. We can use texture recognizer filters to create planes that select for certain textures.

In general, the color of a pixel q is a point in n-dimensional space. For example, an RGB pixel is represented as $q =< q_r, q_g, q_b >$. The description length of a pixel q_c in region R_i, as with the monochrome case, is given by:

$$DL(q) = -log_2 P(q_c | region = R_i) \tag{4.15}$$

The piecewise smooth region model (equation 4.14) used in our segmenter depends upon computing the mean color and the color distance of any pixel in the region from the mean. The distance between any two colors in such an n-dimensional color space is given by:

$$d(q, r) = \sqrt{\sum_{i=1}^{n} (q_i - r_i)^2} \tag{4.16}$$

An 8 bit gray level image should have the same description length as a 24 bit RGB representation of the same gray level image since in either case there are only 256 color values that can have a non-zero probability. That satisfies our intuitions about description length and duplicated information.

If the color planes were encoded separately, the RGB description of a gray image would be three times the description length of the gray level image. Furthermore, the

success of the segmentation would be different depending upon whether RGB, HSI, or CMY representations were used.

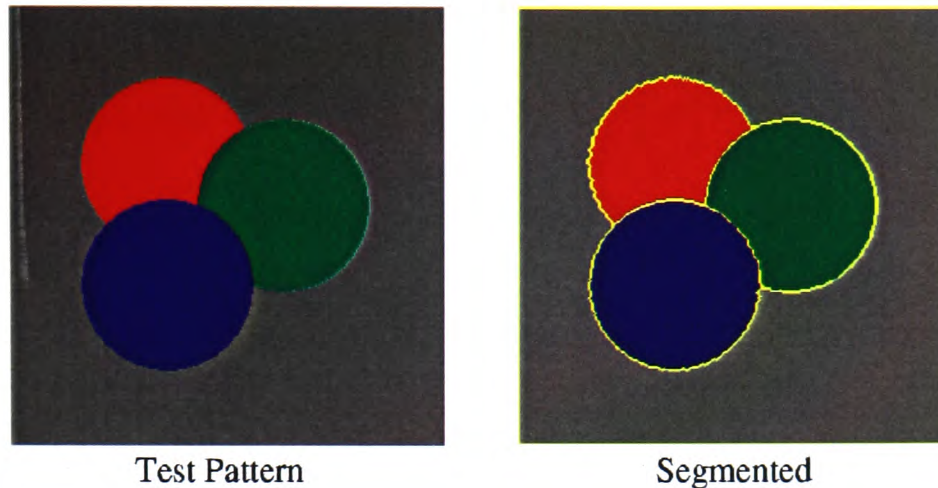


Figure 4.8: Color Segmented

Figure 4.8 is a synthetic image in which the red circle (top left) has RGB color (255,0,0), the green circle (top right) has RGB color (0,255,0), and the remaining blue circle has RGB color (0,0,255). The background has RGB color (85,85,85) Every pixel in the image has RGB color that adds up to 255.

Our implementation of the MDL base segmenter accepts any number of image planes that have equal dimensions and 8 bits per pixel.

4.3.5 MDL Agent Implementation

The segmentation algorithm described abstractly so far is implemented using the agent language described in Chapter 3. Here we give an outline of how the base segmentation algorithm is structured as agents.

Seed Points

We defined some simple seed point agents for the purpose of generating seed points to get the segmentation started. The selection of good seed points is important to the quality of the final segmentation. The seed point agents that we defined were simple but have so far proved to be effective.

The seed point agents take a region as input and produce sub-region seed points. The algorithm used for seed point generation is as follows:

1. For every pixel of the region, using a moving window, compute the window mean $\mu_{i,j}$ and the deviation from the window mean $d_{i,j}$. This produces a landscape in which smooth regions have low deviations and in which boundaries have high deviations. The maximum d_{max} and minimum d_{min} deviations in the region are recorded during the pass.

2. The midpoint between d_{max} and d_{min} is chosen as a cutoff point. All pixels with a deviation less than the midpoint are part of new sub-regions. The new sub-regions have their borders traced.

3. All new regions smaller than the minimum region size are discarded.

The seed point agent is initially applied to the whole image. Once the major regions have been identified and have settled down, the seed point agent is applied to the individual regions. The algorithm terminates when all new regions are too small to qualify as new regions.

This algorithm successively refines regions in a way similar to graduated non-convexity (Blake & Zisserman 1987). The seed points generated are already close to the final shape and so few iterations on the competition algorithm have to be run in order to refine the boundary edges.

Region Shapes and Contents

A Region consists of its boundary and the pixels contained within the boundary. We define a chain shape model for the base segmenter. Shape models and content models have cooperative roles. The shape models define the set of pixels that constitute the region and the content models define the nature of the pixels themselves.

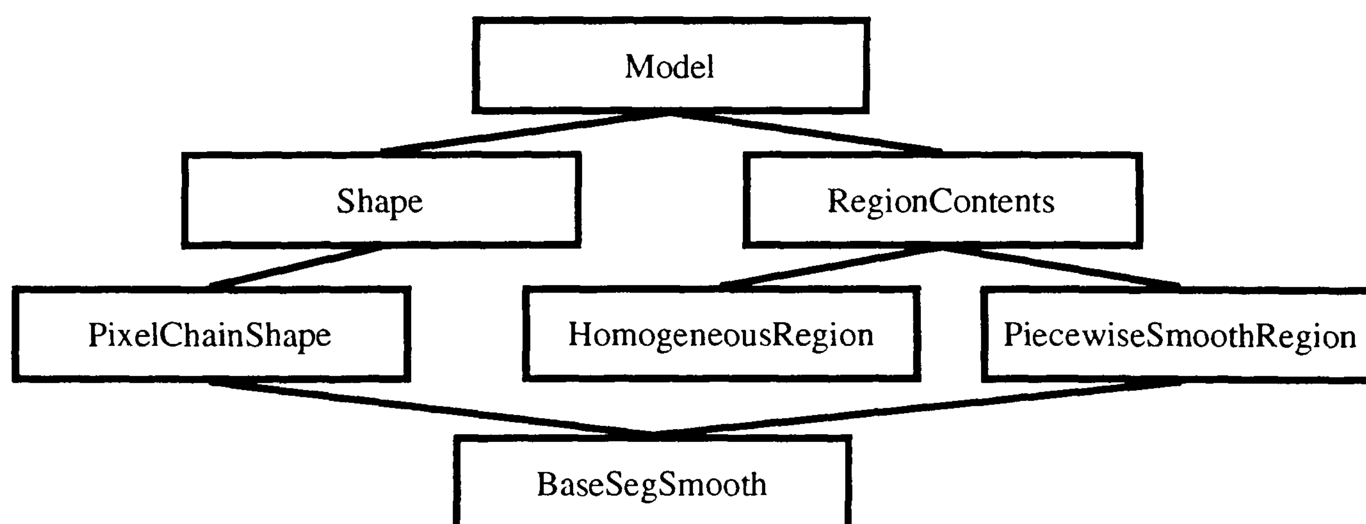


Figure 4.9: Models for the Base Segmenter

The PixelChainShapeModel is parameterized by a starting pixel and a chain list. The agent that produces the "PixelChainShapeModel" sets the "startPixel" and the "chainCodes" as the result of applying the "fit" operation.

```
(defineClass PixelChainDescriptionElement (DescriptionElement)
  (...
   (startPixel)
   (chainCodes)))
```

The description length of a PixelChainDescriptionElement PC is given by:

$$DL(PC) = -\,log_2(\frac{1}{p}) - log_2 P(|B_i| = k) + \sum_{j=0}^{k} -log_2 P(M_{i,j}|M_{i,j-2}, M_{i,j-1}, M_{i,j+1}, M_{i,j+2}) \quad (4.17)$$

where p is the number of pixels in the image, $|B_i|$ is the number of pixels in the pixel chain, and $M_{i,j}$ is the chain direction. $P(M_{i,j})$ is learned by from the corpus of hand annotated images.

The "HomogeneousContents" description element is produced by an agent in response to the "fit" operation. Fitting the "HomogeneousContents" model involves finding the mean and variance of the pixels in the region. The Gaussian representation assumes that the homogeneous region has been corrupted by Gaussian noise.

```
(defineClass HomogeneousContentsDescriptionElement (DescriptionElement)
  (...
   (mean)
   (variance)))
```

The description length of a HomogeneousContentsDescription is given by:

$$DL(C_i) = \sum_{l=0}^{|C_i|} -log_2 \left\{ erf\left(\frac{(q_l - \mu_p + \epsilon/2)}{\sigma_p}\right) - erf\left(\frac{(q_l - \mu_p - \epsilon/2)}{\sigma_p}\right) \right\} \quad (4.18)$$

where ϵ is the resolution of discretization. So for example, if the pixels q are quantized into 256 discrete values, $\epsilon = 256$.

The Piecewise smooth case is similar except that fitting the model involves finding four quantities:

1. **The mean mean:** The local mean is computed for each pixel in the region using a disk-shaped window.

2. **The mean spread:** The distribution of the local mean is represented as a histogram.

3. **The mean deviation from the local mean.**

4. **The spread of the deviations from the local mean.** We employ a simple model in which the distribution of deviations is assumed to be constant for all local windows. Other possibilities exist. The distribution could be a function of the mean such as when well-illuminated parts of a surface have more visible texture than the dimly lit parts of the same surface. In other cases, the texture

may not be related to the mean at all. For example, when a texture recedes into the background the mean may remain constant but the closer part of the surface may offer greater variety of intensities.

```
(defineClass SmoothContentsDescriptionElement (DescriptionElement)
  (...    (meanMean)    (meanSpread)    (devMean)    (devSpread)))
```

In this implementation, the PDFs for the mean and deviations from the mean are maintained as histograms of frequencies; so, if $freq_{m,i}$ and $freq_{d,i}$ are the number of occurrences of mean m and deviation d respectively, the probabilities $P(m|region = R_i)$ and $P(d|region = R_i)$ are calculated as

$$\begin{aligned} P(m|region = R_i) &= \frac{freq_{m,i}}{|C_i|} \\ P(d|region = R_i) &= \frac{freq_{d,i}}{|C_i|} \end{aligned} \tag{4.19}$$

So the description length for a piecewise smooth region C_i is given by:

$$DL(C_i) = \sum_{l=0}^{|C_i|} -log_2 \frac{freq_{m,i}}{|C_i|} - log_2 \frac{freq_{d,i}}{|C_i|} \tag{4.20}$$

Base Segmentation

The base segmentation description element is constructed from a choice of shape description and contents description and the model (BaseSegSmooth) is generated via multiple inheritance from the classes PixelChainShape and PiecewiseSmoothRegion as shown in Figure 4.9.

Figure 4.10 is an example sequence of the segmentation of an image with snapshots taken every 40 iterations that demonstrates region merging, region splitting, and region competition.

So far, we have gone to a lot of trouble to build a rather simple segmenter (the base segmenter) that follows a strict MDL approach. The agent architecture described in Chapter 3 has provided a protocol that has allowed us to structure the segmenter in terms of models. The true power of this approach is that, given this framework, it is easy to add semantics to the segmentation process by defining more complex shape and content models. Below, we present a simple demonstration of how this capability enables us to segment a subjective contour example.

4.3.6 Adding Semantics

We introduced the general idea of a shape model and a content model above. Here using the same protocol we introduce some new shape models. In order for subjective contours

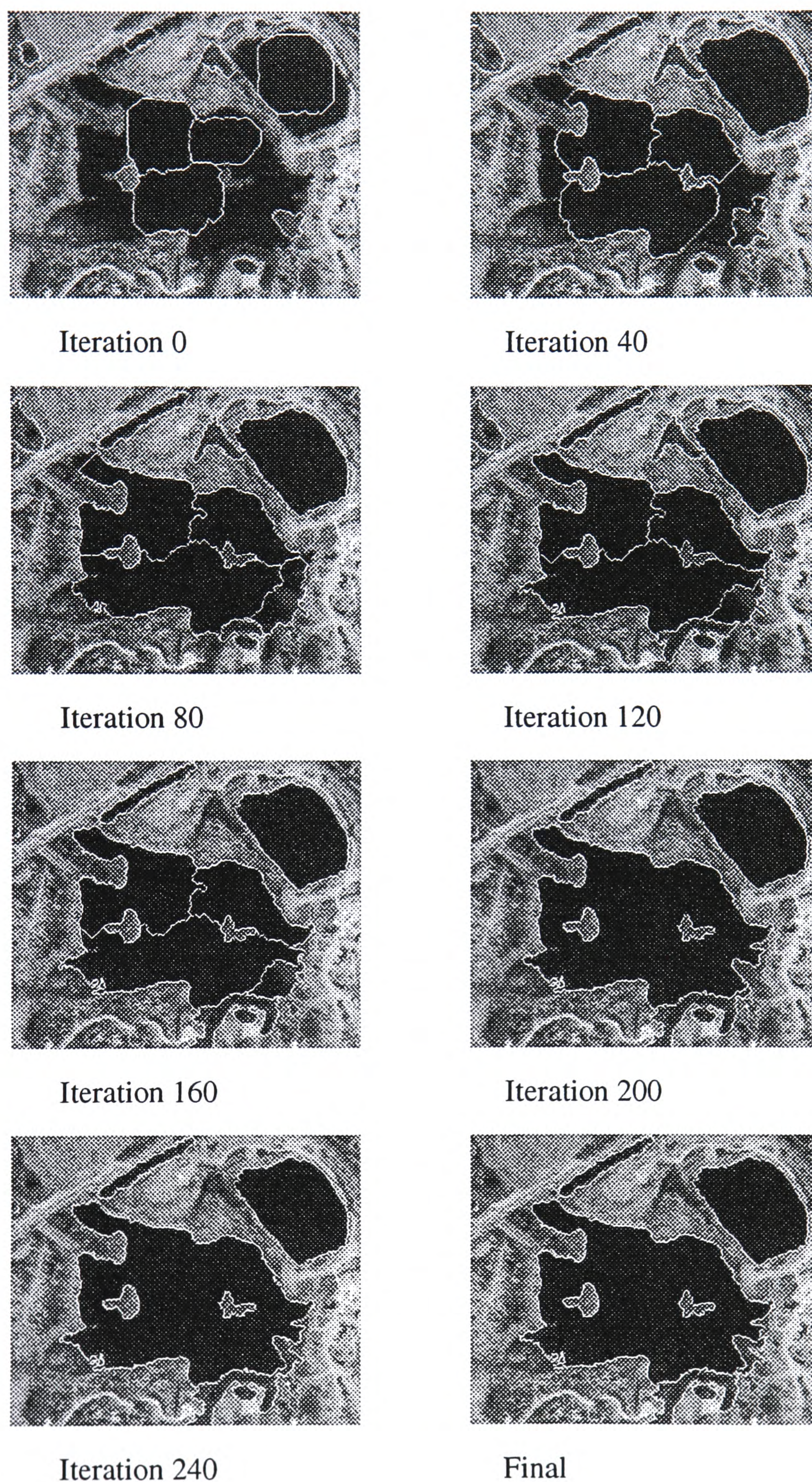


Figure 4.10: Segmenting an Image with a Lake

or the overlapping leaf examples to segment correctly it is also necessary to introduce the idea of occlusion. The algorithm for occlusion is introduced after the shape models.

Shape Models

The shape model used by the base segmenter involves a chain of pixels that surround the region. The pixel chain representation is ideal for region growing algorithms. Other kinds of algorithm can segment images. With the benefit of the MDL formulation these approaches can be mixed. However a region model is fitted to the image data, as long as the description length can be computed, the different representations can compete to produce a global description.

Examples of other approaches to shape models include geometric shapes and splines with parameterized join points. To test the mechanism, we implemented some simple shape models and agents. We describe the circle and triangle models below and conclude by showing their effectiveness in segmenting a subjective contour example.

Shape Model for Circles

Circular regions can be parameterized as $< x, y, r >$ where x and y specify the center of the circle and r its radius.

```
(defineClass CircleShapeDescriptionElement (DescriptionElement)
  (...    (center)    (radius)))
```

The description length of the circle shape is given by:

$$DL(circle_{x,y,r}) = -log_2 P(circle) - log_2(\frac{1}{p}) - log_2 P(r = n) \tag{4.21}$$

The shape model described above can be "fitted" by any number of agents that use different fitting algorithms. We implemented a single circle fitting agent that attempts to adjust x, y, r to an existing region. The fitting algorithm uses a generalized Hough transform (Hough 1962) for circles. The points along each boundary of each base region are used to vote for a circle.

Shape Model for Triangles

Triangular regions can be parameterized as $< x_1, y_1, x_2, y_2, x_3, y_3 >$ where the three corners of the triangle are located at (x_1, y_1), (x_2, y_2), and (x_3, y_3).

```
(defineClass TriangleShapeDescriptionElement (DescriptionElement)
  (...    (corner1)    (corner2)    (corner3)))
```

The description length of the triangle shape depends upon the position of one corner (which is assumed to be equally likely at any position) and the x and y displacements (xd_2, yd_2, xd_3, yd_3) to the other two corners. The x and y displacements are assumed to be equally likely so as not to prefer any particular orientation of the triangle.

$$
\begin{aligned}
DL(triangle_{x,y,r}) = & - log_2 P(triangle) - log_2(\frac{1}{p}) \\
& - log_2 P(xd_2) - log_2 P(yd_2) \\
& - log_2 P(xd_3) - log_2 P(yd_3)
\end{aligned}
\tag{4.22}
$$

As with the circle shape model the triangle shape model can be "fitted" by any number of agents that use different fitting algorithms. We implemented a single triangle fitting agent that finds lines using a straightforward Hough transform. Line intersection points are computed for lines, and triangle hypotheses are computed from them.

Occlusion

When two or more shape models overlap the overlapping models, each lay claim to the pixels that they overlap. We call the pixels that fall within the overlapping regions "disputed". The intersection of overlapping regions form "disputed regions". The following rules govern the representation of disputed regions and their boundaries.

1. Disputed pixels are part of the shape description. An occluded shape does not change shape because it is occluded.

2. One and only one region represents the contents of the disputed pixels. When a shape is occluded, the visible pixels belong to only one of the objects involved in the occlusion.

3. Occlusion is local. In general if one region is in front of another region in one place, it may be behind somewhere else.

Which region gets to describe the disputed pixels is determined so as to produce the global MDL. Assigning content pixels to a region will affect the description length of that region and may affect the state of that region. For example, if the region describes its contents in terms of a distribution, the distribution will be different with the pixels included. That changed distribution may affect the likelihood of other occlusions being determined in favor of that region. Therefore, even though occlusion is local and one cannot reason about z-order, the assignment of pixels to a region may have non-local effects. The non-local effects are not caused by reasoning about non-local "in-front-of" relationships but are caused by changes to description lengths of the regions that represent the content pixels.

Locally however, "in-front-of" is a meaningful and constraining concept. Consider the case of three overlapping circles such as the red, green, and blue circles in Figure 4.8. In that figure there are 4 categories of overlapping shape. In one region red overlaps green, in another region green overlaps blue, in another region blue overlaps red and in

the center, red, green, and blue all overlap. If the overlap between red and green is $d_{r,g}$, the overlap between green and blue is $d_{g,b}$, and the overlap between blue and red is $d_{b,r}$.

The shape models compete using the Monte-Carlo methods provided by the agent architecture described in chapter 3. Locally, where regions overlap (intersect) they produce a disputed region. At the local point where multiple shapes overlap the regions can be sorted into "in-front-of" order. All possible orderings are evaluated by considering the sum of the description lengths of the affected regions for all possible layerings of the shapes.

The algorithm for resolving local overlapping simply depends upon finding the choice set. The first step is to produce the disputed set. From the disputed set, the choice set is produced. The built-in Monte-Carlo algorithm then finds the best solution globally by sampling from the choice sets of all choice sets in the image. A disputed set DS_i consists of a set of regions involved in the dispute and a set of disputed regions. We use the three overlapping circles case from Figure 4.8 to illustrate the algorithm.

1. All shapes are pair-wise intersected. Each intersection produces (1) a set of the regions involved in the intersection, and (2) a list containing structure consisting of a representation of the intersecting (disputed) region and a set of regions involved in competing for the pixels (the same list as (1)).

   Doing this for the overlapping circle example generates the following disputed regions.

$$\begin{gathered}(\{R_g R_b\}(< d_{g,b}, \{R_g R_b\} >)) \\ (\{R_g R_r\}(< d_{g,r}, \{R_g R_r\} >)) \\ (\{R_b R_r\}(< d_{b,r}, \{R_b R_r\} >))\end{gathered} \tag{4.23}$$

2. All disputed regions generated in the first step are intersected to see if any of them overlap. Disputed region representations that are found to overlap are merged by taking the union of the sets involved in the intersection and appending the disputed region list. With the overlapping circle example this generates the following disputed regions list.

$$(\{R_r R_g R_b\}(< d_{g,b}, \{R_g R_b\} >< d_{g,r}, \{R_g R_r\} >< d_{b,r}, \{R_b R_r\} >)) \tag{4.24}$$

   In this example, because all of the overlapping regions intersect we are left with a single disputed regions list. In general there will be several. Each disputed region entry will decide the region that "loses" the disputed region by producing an ordered list of regions involved in the dispute. Given an ordering, for each region

associated with the disputed region, the most buried region "loses" the disputed pixels.

3. Each disputed region list is expanded into a choice set. If there are n regions involved in the choice set there are $n!$ ways of ordering the regions. For the overlapping circle example the following choice set is produced.

$$\begin{aligned}&\{(R_rR_gR_b)(<d_{g,b},\{R_gR_b\}><d_{g,r},\{R_gR_r\}><d_{b,r},\{R_bR_r\}>)\\&(R_rR_bR_g)(<d_{g,b},\{R_gR_b\}><d_{g,r},\{R_gR_r\}><d_{b,r},\{R_bR_r\}>)\\&(R_gR_rR_b)(<d_{g,b},\{R_gR_b\}><d_{g,r},\{R_gR_r\}><d_{b,r},\{R_bR_r\}>)\\&(R_gR_bR_r)(<d_{g,b},\{R_gR_b\}><d_{g,r},\{R_gR_r\}><d_{b,r},\{R_bR_r\}>)\\&(R_bR_gR_r)(<d_{g,b},\{R_gR_b\}><d_{g,r},\{R_gR_r\}><d_{b,r},\{R_bR_r\}>)\\&(R_bR_rR_g)(<d_{g,b},\{R_gR_b\}><d_{g,r},\{R_gR_r\}><d_{b,r},\{R_bR_r\}>)\}\end{aligned} \tag{4.25}$$

4. Each choice in each choice set is assigned a description length that is computed by assuming the "in-front-of" order of the choice and taking the sum of the description lengths of the regions involved in the dispute. So for the overlapping circles example, the first choice of the choice set in which the red region is in front of the green region and the green region is in front of the blue region:

$$\begin{aligned}DL(choice_{r,g,b}) =& DL(R_r)+\\&DL(R_g|excluded = d_{g,r})+\\&DL(R_b|excluded = d_{g,b}, exluded = d_{b,r})\end{aligned} \tag{4.26}$$

As described in Chapter 3, the Monte-Carlo selection scheme samples the choices at random by computing the probability of each choice $P(choice) = 2^{DL(choice)}$, normalizing the probabilities of the choice set so that they sum to 1 and then selecting the choices according to the resulting probabilities.

All regions participate in the occlusion computation. Base regions never occlude other base regions so the base regions are not intersected with other base regions as an optimization. Base regions can however occlude, and be occluded by, non-base regions.

Using the circle and triangle shape models described above the subjective contour example in Figure 4.11 was successfully segmented. First the base segmenter segmented the regions to produce the three pacman shapes. The corner detectors invoked the triangle agent which found the three corners and hypothesized the triangle. Adding the triangle, however, doesn't reduce the description length because the pixel that it takes from the base segment (background) are no smaller when part of the triangle than they

were as part of the background. In addition the triangle has to represent its shape. The blob detectors invoke the circle agents which fit circle models to the pacman shapes but again the description length of the circles themselves exceeds the description length of the base segmented pacman shapes because although the shape description is cheaper for the circle the circle must describe the pixels in the wedge which are a different color. The overlapping algorithm described above however finds that selecting the "triangle on top" interpretation leads to a shorter global description length and so the subjective contour representation prevails.

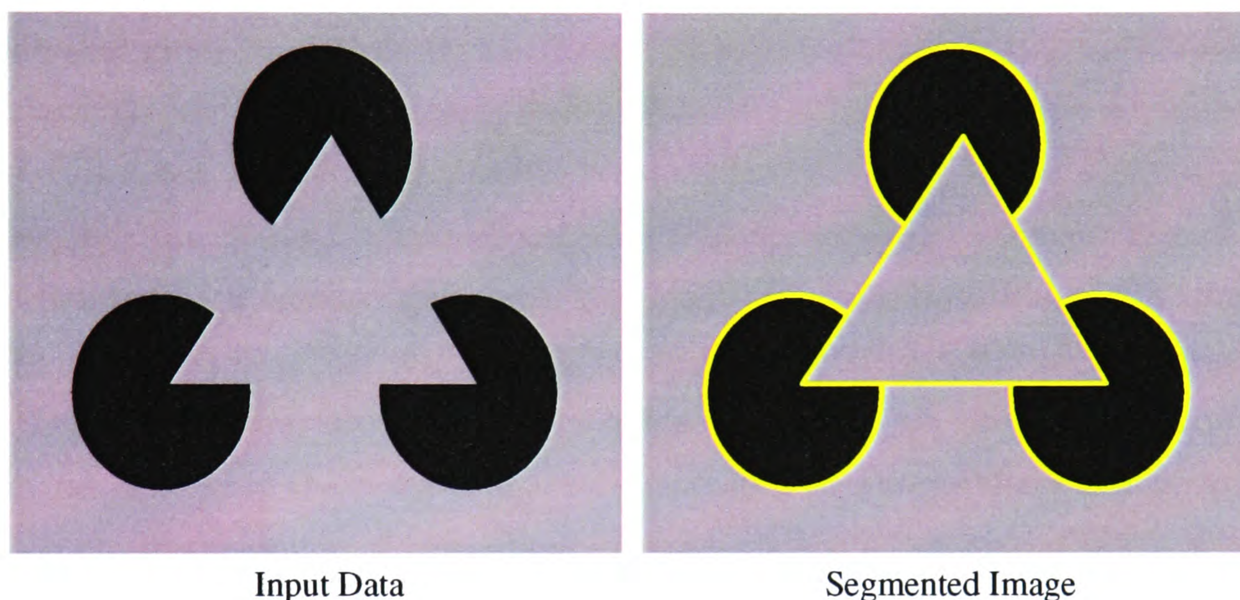


Figure 4.11: Segmentation of Subjective Contours

4.4 Conclusion

We have developed a novel algorithm, semantic segmentation, for image segmentation. The algorithm consists of: a null semantics base segmenter that partitions the image into piecewise smooth regions in such a way as to minimize the MDL description of the partitioned image; a framework for managing occlusion, and a protocol for defining new shape and content models.

The base segmenter produced much better segmentations than expected. The results are comparable to those achieved by Leclerc and Zhu & Yuille even though the base segmenter is simpler than either. Our original goal and rationale was that the base segmenter should have minimal complexity because it was better to have things like outline smoothness governed by semantics than to arbitrarily build in a preference for smooth outlines into the base segmenter. The hard-wired boundary smoothness semantics of Zhu & Yuille seem in practice to have minimal impact on the quality of the segmentation. On the other hand our piecewise smooth model allows much of the un-modeled components of the image formation process such as geometry, lighting, and

surface texture to be accommodated. Once the regions have been correctly segmented, models of lighting, geometry, and surface texture could in principle be applied to the regions in an attempt to further reduce the description length. We did not try modeling these things but the same mechanism that permits high-level semantics to be introduced allows further models of image formation to be attempted. The multistage approach to separating different factors in the image formation process may be easier to implement than trying to build complex analytic models that account for all of these factors at once.

The subjective contours demonstrated as an example of introducing semantics to image segmentation problem is an extreme example of Marr's overlapping leaf phenomenon. While the leaf outline in Marr's example is impossible to see in the case of subjective contours the lines are actually not present at all. We claim that we see the subjective contours because the image interpretation of occluded objects is more likely and thus can be represented by a smaller message length than the representation of the irregular objects. We are not saying that the mechanism of subjective contours presented here is similar to the way that they are perceived in humans but if we accept that human vision is also engaged in seeking the most likely interpretations of images there is a certain inevitability to the phenomenon of subjective contours. What is important is not whether the mechanism is similar to anything the human vision system does—it is that we are able to handle a class of segmentation problems described by Marr (Marr 1982).

We have addressed Marr's concerns about segmentation by liberating it from a low-level process to one that spans the processing stages of image interpretation. Marr was concerned by semantics because they tended to be *ad hoc* and difficult to apply. It is not necessary for semantics to be ad-hoc. In many cases the semantics can be extracted automatically from a corpus.

We have realized a conjecture of Leclerc that the MDL paradigm could be extended beyond low-level segmentation to include later stages of image interpretation.

The semantics used to illustrate the architecture in this chapter were chosen for their simplicity. In the next chapter we develop linguistic semantics that can bring contextual information to bear on the image segmentation and interpretation problem.

Chapter 5

Patchwork Parsing

5.1 Introduction

We concluded Chapter 4 with an example of a shape model. In this chapter we look at another dimension of semantics directly relevant to the problem domain of aerial image interpretation: region context. Aerial images are distinguished by colors and textures more than by shape. Occasionally a shape is identifiable in aerial images, such as central park in New York city, but most shapes are natural and reflect features of the terrain, while man made shapes become part of the texture of regions of the image, such as the regular structure that makes up the streets and avenues of New York city. Partly by design and partly by necessity the relationship between different regions follows a recognizable structure. Beaches are found between the sea and the land. Suburban areas appear next to urban areas which in turn are often adjacent to rivers, estuaries, and oceans. The context provided by the layout of regions in a natural scene is key to correctly identifying the region's contents and also to understanding the image as a whole. In this chapter we develop a linguistic approach to image understanding that addresses both the region content interpretation and the interpretation of the whole image as a parse tree.

Another approach to adding *context* is Markov random fields (MRFs) (Geman & Geman 1984). Markov random fields provide probabilities that can be learned from the data but the models are structurally weak. Markov random fields are similar to the linguistic approach described in this chapter; but their formulation makes it hard to incorporate other differing approaches to understanding. The syntactic rules that we learn from images and the way that we use them to make sense of the pattern of regions in an image are similar to the cliques of MRFs. With the approach described in this chapter, however, the agent language [described in Chapter 3] provides the framework and MDL provides the common currency for incorporating a variety of approaches to semantics within a single program.

Language research has enjoyed considerable success in applying statistical techniques to the problem of inducing realistic grammars from corpora and parsing natural speech and text. The main innovation in the work described in this chapter is the extension of those ideas to image interpretation. The major challenges of this work are that, unlike

natural language, there is: the lack of an obvious grammar for images and the complexity of special relationships between image components.

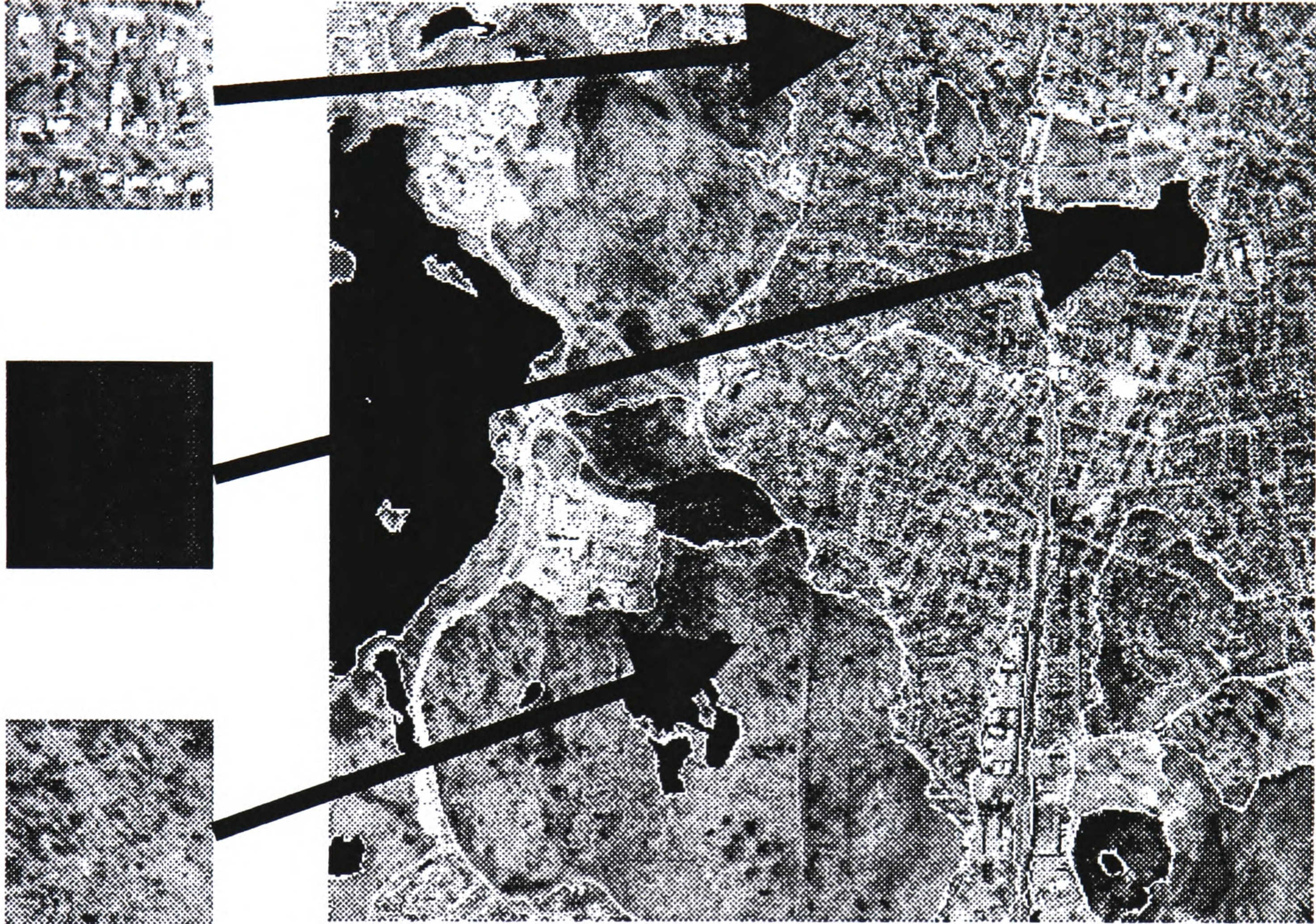


Figure 5.1: Patches without context are hard to identify

An initial approach to introducing semantics into the problem of understanding an image divided into regions is to identify the region types using signal-based region identification algorithms—such as texture and color models (Ducksbury 1993; Agosta 1990). However, much of our ability to assign content descriptions to regions seems to come from context, since when we are shown a region out of context, it is usually not possible to assign an interpretation. Figure 5.1 shows a few examples of spots taken from regions of an aerial image. While much evidence can be obtained from the regions, without bringing in evidence from context, reliable region content identification is extremely difficult. Purely signal-based approaches to region identification are easily defeated in practice by complications arising from real images. In many cases the signal characteristics of a region may be the same and distinguished only by context and shape. For example a suburban region may have similar signal characteristics to a town but is recognizable as suburban because of its relationship to an urban area. Similarly islands, lakes, seas, and rivers are distinguishable because of their shape and context.

Signal-based methods allow hypotheses to be formulated as to the contents of regions based on things like texture and color. By imposing constraints on the structural relationship between regions of certain types, the selection of ambiguous region interpretations can be improved and the initial segmentation of the image can be modified to yield a globally more probable interpretation of the image. The cooperation between signal-based algorithms for segmentation and identification along with structural constraints can allow the system to converge on a good global interpretation that is more robust to noise and other optical phenomenon that cause purely signal-based approaches to be fragile.

In this chapter we limit our discussion to the problem domain of aerial images although the results may be applicable more generally. Aerial images are convenient in that it is not necessary to consider foreshortening resulting from three dimensions and in most cases object occlusion is not an issue either (except for bridges, tunnels, and, particularly clouds).

Two image banks (the corpora) were developed in order to obtain the picture language and optical models required by the algorithms described here. One corpus is based on multi-spectral satellite images and the other is gray scale images taken from a plane. These corpora are described in Appendix A and the facility for annotating the images is described in Appendix B.

This chapter is organized as follows. Section 5.2 provides a discussion of related prior work in language/speech understanding. Section 5.3 defines the statistical model for patchwork image parsing and develops the grammatical aspects of the approach. Section 5.4 develops algorithms for patchwork image parsing.

5.2 Prior Work

There are many parallels between the problems in understanding visual processing, human speech, and natural language:

1. *Input signal processing*: In the case of speech it is a one dimensional sound signal whereas in vision the signal is two dimensional. In natural language, the signal is in the form of text that may or may not contain noise.

2. *Input signal segmentation*: As we said in Chapter 4, despite years of research in vision and speech, robust algorithms for segmentation remain elusive. Segmentation without simultaneous interpretation seems at best difficult, at worst hopeless. In speech recognition there is usually an attempt to divide the wave form into separate words. In vision programs, segmentation usually plays a significant role, but the nature of the segmentation varies greatly depending on the application.

3. *Component identification*: In the case of speech and natural language words are identified. In vision regions are identified.

4. *Structural understanding*: Speech recognition involves producing a word list, a task usually aided by a context that controls expected words. In natural language there is usually an attempt to parse the words into phrases or sentences. Similarly in vision there is a need to identify the structural relationship between the image components.

There are also some important differences. Natural language has a reasonably useful intermediate representation (written text) between the low level signal processing aspects of speech recognition and the higher level aspects of natural language understanding. Vision does not appear to have a viable intermediate form. Early attempts at using line drawings for this purpose did not scale to natural scenes (Waltz 1975).

The work in this chapter draws upon statistical techniques that have been common in speech understanding (Bahl, Jelinek, & Mercer 1983; Jelinek, Lafferty, & Mercer 1992) since the early 1980s and natural language understanding (Charniak 1996; Schabes 1992) since the late 1980s. The statistical techniques used in our implementation of these ideas (GRAVA) are appealing to us for three reasons:

1. GRAVA is "grounded" in that the models that it applies are derived from the real world. Building models by hand is extremely difficult. While building models by hand is feasible for a small class of highly specialized tasks (such as face recognition), the diversity present in the real world demands that models be extracted from real data rather than being hand crafted. The statistical techniques described below allow for automatic grammar induction.

2. GRAVA uses MDL as a common currency for choosing among alternative interpretations. MDL demands that for any descriptive element the probability of its occurrence in a description be known so that its description length can be computed using the formula $DL(x) = -log_2 P(x)$. The corpus based statistical techniques make the necessary probabilities easily accessible.

3. It was once believed that in order to learn models, it was necessary to provide both examples and counter examples in order to properly learn the phenomenon. Winston's system (Winston 1975) for learning structural descriptions from examples was an example of a learning system that required both examples and "near miss" counter examples. MDL and statistical methods, however, allow learning on the basis of examples without any requirement for counter-examples. Since the solution chosen will be the one with the minimum description length it is not a

problem that a sentence (say) may have a meaningless but grammatical parse as long as the meaningful parse has a lower description length. By learning grammars from corpora the required probabilities can be computed from frequencies.

5.2.1 Speech Recognition

Speech understanding starts with a signal. The signal is divided into words, the words are identified, assigned parts of speech, and [depending upon the application] parsed into meaningful sentences. Speech applications are at present rather limited in scope and so the definition of the task of speech understanding has historically been defined as extracting a word list from an input signal. The context is generally carefully constrained to make this task possible. Parsing the word list into a meaningful sentence or phrase is generally categorized as a natural language task.

Evidence for the words comes from analysis of the signal but it is often impossible to identify the words from analysis of the signal alone. The context of the word provides a lot of constraint for: the identification of the word, assignment of the part of speech, and choosing from ambiguous parses of the sentence.

The principle problem in speech recognition is to identify the words that correspond to pieces of the signal that are segmented as words. The segmented pieces of wave form are compared against templates to try to determine the words that they represent but invariably this cannot yield an unambiguous determination. If we hear an isolated word we too are often unable to determine the word being uttered. However, when we hear the word in context we can identify the word with confidence. This is the idea of speech modeling.

A sentence W can be considered to be a sequence of words. The task is to find the sequence of words that is *most probable* given the signal.

Words occur with a certain frequency in natural language but the probability of their occurrence is not independent of the surrounding words. As we noted in Chapter 3, the Shannon information measure (Equation 3.1) makes the assumption that the data stream is ergodic. To correct for this we must consider the probability of each word in context. For consistency with the rest of the thesis we describe the following in terms of description lengths. The description length of a word sequence W is:

$$DL(W) = \sum_{i=1}^{n} -log_2 P(w_i|w_1, ..., w_{i-1}) \tag{5.1}$$

In that form, it is not very useful, because we would have to know the probabilities of all possible word sequences of arbitrary length. Instead an approximation is used. A common approximation is the trigram model in which word sequences of three consecutive words are collected from a representative corpus for the application.

$$DL(W) \cong \sum_{i=1}^{n} -log_2 P(w_i|w_{i-1}, w_{i-2}) \quad (5.2)$$

The probability $P(w_i|w_{i-1}, w_{i-2})$ is approximated by $f(w_i|w_{i-1}, w_{i-2})$ which is computed by counting occurrences in a representative corpus of text (or transcribed speech).

Even this simplification is not sufficient for practical uses because there are a lot of possible three word sequences, even for a constrained vocabulary system, requiring an unreasonably large corpus to provide adequate language coverage. This is dealt with by smoothing: in the cases where a matching three word sequence cannot be found in the corpus, a two word sequence may be used. In cases where even two word sequences may not appear in the corpus, the probability of the single word must be used. The approximated description length of the word sequence is given by:

$$DL(W) = \sum_{i=1}^{n} -log_2\Big(\lambda_3 f(w_i|w_{i-1}, w_{i-2}) + \lambda_2 f(w_i|w_{i-1}) + \lambda_1 f(w_i)\Big) \quad (5.3)$$

Given this formulation, the task of finding the most likely word can be represented as a Hidden Markov Model (HMM) and solved efficiently using the Viterbi (Viterbi 1967) algorithm. Good weights for λ_1, λ_2, and λ_3 can be computed using the Baum-Welch algorithm (Baum 1972). Systems built this way have shown remarkably good performance.

The difficulties with such methods are predominantly problems of size. It is impractical to have a corpus large enough to reasonably approximate the probabilities—methods for dealing with sparse data are essential. The number of states in the hidden Markov models can also become excessive. This problem is best addressed by dividing the problem space into smaller contexts, each of which is accompanied by an appropriately limited vocabulary. This way, a number of manageable HMM implementations are able to handle reasonably large vocabularies.

5.2.2 Natural Language

Understanding natural language is similar in spirit to speech recognition but differs in emphasis. In most cases the word sequence is already known. Either it was produced as the output of a speech recognition program, or it was simply read in as text. Either way, the problem is to parse individual sentences into intelligible structures. There are two chief activities involved in parsing the sentences.

1. *Identifying the part of speech of the words.* Some words have no part of speech ambiguity whereas other words have many possible parts of speech. An example of the latter is the word *like*. *Like* can be a preposition as in "(she walks) *like* a

lady" or a verb as in "kids *like* sweets". Resolving the correct part of speech of a word is essential if the parser is to pick the correct parse.

2. *Parsing the words into a parse tree.* Computer languages are carefully designed to be unambiguous and the syntax is designed to permit efficient parsing. Natural languages (like English), however, are highly ambiguous, so it is not uncommon for a sentence to have thousands of (syntactically) correct parses. While all the parses may be grammatically correct, they are not semantically equivalent. A sentence may mean different things when parsed in one way rather than in another, while other parses may seem devoid of any meaning. Selecting the correct parse from the many possible parses is essential to machine understanding of the sentence.

We consider each of these activities in turn.

Part of Speech Tagging

Part of speech tagging is the process of assigning for each word in a sentence a part of speech that correctly identifies its grammatical mode. The "sparseness of data" problem is much less of a problem for part of speech tagging than for word identification in speech recognition systems since there are far fewer parts of speech (between 10 and 150 depending on the system) than there are words. The speech recognition goal is to find the most likely sequence of words to correspond to a sequence of signal segments. The task in part of speech assignment is to determine the most likely sequence of parts of speech given a sequence of words.

The probability of a word sequence is the sum of the probabilities of the word sequence with all possible taggings:

$$DL(W) = -log_2\left(\sum_{t_{1,n}} P(w_{1,n}, t_{1,n})\right) \tag{5.4}$$

The goal of part of speech tagging is to produce the sequence of tags $t_{1,n}$ that minimizes $DL(W)$.

One approach takes the probability of a word as dependent only on its part of speech, where that part of speech is itself dependent on the previous two parts of speech:

$$P(W) \cong \sum_{t_{1,n}} \prod_{i=1}^{n} P(w_i|t_i)P(t_i, t_{i-1}, t_{i-2}) \tag{5.5}$$

Here, missing sequences in the corpus can be handled in the same way as with speech recognition by smoothing. Also, as with speech recognition systems, the assignment can be accomplished by using the Viterbi algorithm to solve the HMM representations of the

problem and by training the HMM to establish good weights for λ_1, λ_2, and λ_3 using the Baum-Welch algorithm:

$$P(W) \cong \sum_{t_{1,n}} \prod_{i=1}^{n} f(w_i|t_i)(\lambda_3 f(t_i, t_{i-1}, t_{i-2}) + \lambda_2 f(t_i, t_{i-1}) + \lambda_1 f(t_i)) \quad (5.6)$$

Performance of these systems is reasonably impressive. Whereas human experts agree on part of speech assignments around 98% of the time, the best machine taggers presently achieve around 97% agreement with human taggers (Charniak 1997).

Statistical Parsing

A context free grammar (CFG) consists of a set of terminal symbols W (the words), a set of non-terminal symbols N, a distinguished starting symbol N^1, and a set of parse rules R. A parse rule describes how a non-terminal symbol N^i can be broken down into a sequence of terminal and non-terminal symbols. A probabilistic context free grammar (PCFG) has for each rule a probability. The probabilities for all the rules with the same non-terminal symbol sum to one. The probability of a rule then is the probability that the rule will be used to expand that non-terminal symbol. The product of the probabilities of all rules and words used in the resulting parse tree is the probability of that parse of the word list occurring in the language. The sum of the probabilities of all legal parses of a word list is the probability of the word list occurring in the language. The probabilities of all possible parses of all possible word lists sums to one.

The probabilities of the parse rules can be induced from a hand parsed corpus just as the probabilities of the words can. The probabilities of the rules allows different parses of the same word list to be compared on the basis of probability. The preferred parse is the one with the greatest probability. Consider the following example (from Charniak (Charniak 1993)).

NT		Expansion	Probability
s	$\Rightarrow$	np vp	0.8
s	$\Rightarrow$	vp	0.2
np	$\Rightarrow$	noun	0.4
np	$\Rightarrow$	noun pp	0.4
np	$\Rightarrow$	noun np	0.2
vp	$\Rightarrow$	np vp	0.3
vp	$\Rightarrow$	np vp	0.3
vp	$\Rightarrow$	np vp	0.2
vp	$\Rightarrow$	np vp	0.2
pp	$\Rightarrow$	prep np	1.0
prep	$\Rightarrow$	like	1.0
verb	$\Rightarrow$	swat	0.2
verb	$\Rightarrow$	flies	0.4
verb	$\Rightarrow$	like	0.4

noun	⇒	swat	0.1
noun	⇒	flies	0.4
noun	⇒	ants	0.5

There are several interpretations of the word list "Swat flies like ants". Two of them are shown in Figure 5.2.

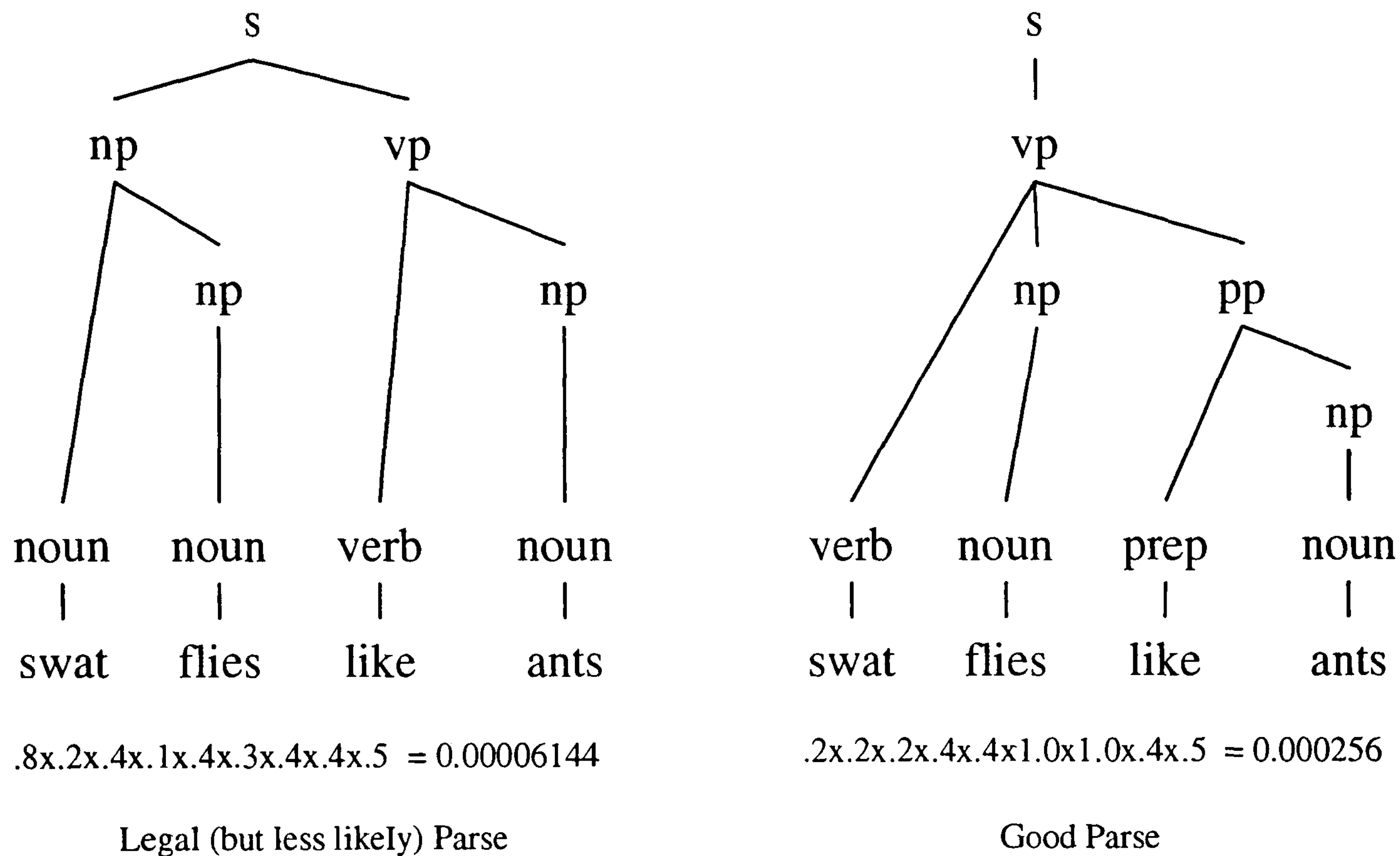


Figure 5.2: Example PCFG grammar

The left parse means that there is a kind of fly called a "swat fly" that likes ants. The right parse means "to swat flies as if they were ants".

Having the probabilities of the rules allows us to select the best parse from a number of possibilities that are all equally correct from the standpoint of the grammar. Without context the right parse (in Figure 5.2) is more likely. If the sentence is part of a larger context, such as a section about the likes and dislikes of the "swat fly", the less likely interpretation may yield the more likely "global" interpretation. One way to make that kind of interpretation work would be to have a semantic layer that built upon the parse layer. The Monte-Carlo selection would then pick the parses that achieved the MDL semantic interpretation.

Monte-Carlo selection allows less likely "locally" choices to be selected if they lead to an MDL "global" description. However, this by itself is probably not adequate. We would like the posterior probabilities of the words and grammatical rules to be different in different contexts. It is unlikely that interpretation is best served by using the same probabilities for grammar and words when reading a scientific document, a legal

document, a self help book, and so on. Each of these documents have their own style that makes certain word usages and grammatical constructs more likely. We could achieve this by having the self-adaptive architecture change the grammar and word models in response to different contexts.

In the remainder of this chapter we develop a statistical parsing scheme for images that is similar to the ideas discussed above but in which the "words" are image regions and the "grammar" is two dimensional. In Chapter 8 we go on to show how different models of image context can be applied by the self-adaptive architecture.

In summary, statistical grammars have a number of important advantages over their non-statistical counterparts:

1. Non-statistical grammars exclude nonsense sentences as a result of restrictive grammatical rules. The statistical techniques described above provide a robustness that comes from accepting almost every word order as legal. With smoothing almost everything can be accepted. Consequently parsing rarely fails. This at first would seem to exacerbate the problem of parse ambiguity; but it enables all sentences to be understood at some level regardless of how ridiculous they may seem. Instead, probabilities allow nonsense sentences a very low probability instead of making them illegal. Instead of excluding the bad sentences and thereby causing the parse to fail, they are included with a low probability which may then be used to signal difficulty in understanding. For sentences that are malformed and would defeat non-statistical parsers, smoothing can allow statistical parsers to parse the sentence and semantic routines may be able to recover meaning from the resulting parse.

2. Parsing allows grammar induction to proceed without the need for negative examples. If a grammar is to be exclusionary it is necessary to give negative examples. Statistical grammars don't require exclusion. The unlikely parses simply don't occur often and so those parses are rarely chosen.

The hidden Markov model (HMM) formulations used in the speech understanding and part-or-speech tagging models can be fairly easily adapted to the case of PCFG's. Efficient algorithms exist for PCFG grammar induction (Baum 1972) and finding the most probable parse (Viterbi 1967).

Unfortunately PCFG's may not be the best kind of grammars for representing natural languages like English. More interesting grammars like the Stochastic Lexicalized Tree Adjoining Grammars (SLTAG) (Schabes 1992) while being more useful meta-grammars for English cannot be parsed efficiently by the above methods.

5.3 Statistical Patchwork Parsing

In this section we develop an approach to mobilizing contextual information in an image by parsing the relationships between disjoint segments in the image, where the segments form a "patchwork" of non-overlapping regions[1]

This approach allows the global structure of an image's internal connected segments to play a part in the assignment of content descriptors of the segments. This is similar to the way that part of speech assignment in speech understanding aids in the identification of words.

We refer to this form of image parsing as "patchwork parsing" because it attempts to interpret an image formed from a patchwork of non-overlapping regions. The goal is to produce a structural description of an aerial image in terms of:

1. A segmentation of the image into non-overlapping regions (cf word segmentation). This issue was addressed in Chapter 4.
2. For each region an identification of its contents (cf word identification). This issue involves learning content models from the corpus. We call this the optical model because it deals with the interpretation of regions of pixels based on their optical characteristics (such as color and texture).
3. A parse tree that describes the grammatical relationships between the regions of the image.

5.3.1 Defining the Image Labeling Problem

A segmented image consists of n regions. We seek to find the labeling of those regions $l_{1,n}$ that is supported by evidence from the image signal and from contextual information. We can state this formally:

$$arg \min_{l_{1,n}} -log_2\left(\prod_{i=1}^{n} P(l_i|r_i, n_i)\right) \tag{5.7}$$

Where l_i is the labeling of region r_i; and n_i is the set of neighbors of region r_i. Applying Bayes' law gives us:

$$arg \min_{l_{1,n}} -log_2\left(\prod_{i=1}^{n} \frac{P(l_i|r_i)P(n_i|l_i, r_i)}{P(n_i|r_i)}\right) \tag{5.8}$$

We make the (reasonable) simplifying assumption that the neighbors n_i are independent of the signal that constitutes the region r_i for which they are neighbors. This allows us to define the image labeling problem as follows:

[1]A grammar for overlapping regions could be developed using a similar model of local occlusion to that employed in Chapter 4. We didn't pursue an occlusion grammar because it has limited applicability in high altitude aerial images.

$$arg \min_{l_{1,n}} -log_2\left(\prod_{i=1}^{n} \frac{P(l_i|r_i)P(n_i|l_i)}{P(n_i)}\right) \tag{5.9}$$

Since $P(n_i)$ does not depend on l_i we can ignore the denominator for the purpose of finding an optimal labeling:

$$arg \min_{l_{1,n}} \sum_{i=1}^{n} -log_2 P(l_i|r_i)P(n_i|l_i) \tag{5.10}$$

This divides the contribution of the knowledge from our two sources as $P(l_i|r_i)$ the *optical model* and $P(n_i|li)$ the *picture language model.*

5.3.2 Optical Model

We discuss specific optical models and their induction from the corpus in Chapter 6. Here we formulate the nature of the optical model and the way in which it is represented.

A segmented region consists of a boundary shape and its inner collection of pixels. These two pieces of information are what is available to the optical model since the neighboring regions are covered in the picture language model. There are many ways that we may go about recovering content information from a region. For example, we may

1. compare the boundary shape against deformable models of outline shape learned from the corpus (Cootes, Edwards, & Taylor 1998),
2. collect evidence of features such as patterns within the region that match templates learned from the corpus and use the presence of such features to categorize the regions,
3. compare textural representations of the regions contents against a database of textures learned from the corpus (Cross & Jain 1983; Xie & Brady 1996), or
4. combine evidence from a variety of sources, such as the sources listed above, to produce a composite description of the region's contents (Ducksbury 1993).

Whatever means is used to arrive at a region content description, we wish to estimate $P(l_i|r_i)$. We represent this in the form of a mapping from the individual regions to a mapping from the labels to a level of belief supported by the optical model as follows:

$$\mathbf{R} = \{< region_1, \{< label_1, confidence_1 > \ldots\} > \ldots\} \tag{5.11}$$

where $\mathbf{R}$ represents the set of regions in the image. Each region $\mathbf{r} \in \mathbf{R}$ consists of a region description and a set of pairs of interpretation and confidence of that interpretation

given the evidence. The confidence of all interpretations for a given label must sum to one.

$$\forall \mathbf{r} \in \mathbf{R}, \sum_{i \in \mathbf{r}} confidence_i = 1 \tag{5.12}$$

The goal of our program was to mimic the interpretation of an aerial image as provided by our human expert. The human interpreter labeled the regions using a vocabulary of symbols. For the SPOT images the following labels were used:

Industrial	Residential	Downtown	Urban
Suburban	Metro	Town	Field
Farmland	Park	Estate	Private Airport
Commercial Airport	Port	River	Sea
Lake	Reservoir	Beach	Island
Swamp	Frozen	Rock Formation	Forest
Cloud	Shadow	Unknown	

The image corpora and associated labels are described in detail in Appendix A.

5.3.3 Picture Language Model

Below we develop the picture language model as a grammar for a patchwork of regions. The grammar doesn't deal with occlusion in a manner suitable for dealing with three dimensional scenes for the reasons mentioned above. We do however implement a simple form of occlusion which we refer to as "obscuration" to distinguish it from a general occlusion model.

The simple obscuration model is necessary because unlike sentences in natural language, visual scenes don't have a beginning or an end. Visual scenes simply go on forever. We can only parse the part that the sensor captures. The rest of the landscape is obscured by the sensor itself (the image boundaries).

A picture grammar is a four-tuple $< N, N^1, T, R >$ where N is a set of non-terminal symbols, N^1 is the starting symbol, T is the set of terminals, and R is a set of rules. The terminal symbols are the labeled regions. The rules are context sensitive rules that aggregate collections of contiguous regions into a larger complex contiguous region. The general form of the rule is:

$$P(NT \;\; context \rightarrow primary \;\; inclusions) = n \tag{5.13}$$

where NT is the non-terminal symbol defined by the rule, *context* is the context in which the rule occurs, *primary* is the possibly null primary region of the non-terminal,

inclusions is a possibly empty sequence of included terminal and non-terminal symbols, and n is the probability that this rule will be used to expand the non-terminal symbol NT. For example a non-terminal LAKE in the context of a field where the lake contains an island would be written as follows:

$$P(LAKE\ (field) \rightarrow lake\ (island)) = 1.0 \tag{5.14}$$

The uppercase "LAKE" is the name of the non-terminal and the lowercase "lake" is the region labeled as a lake. The probability of 1.0 means that this is the only rule for parsing the non-terminal "LAKE".

Below we develop the meta-grammar for pictures based on regions.

Simple Non-Terminals

The segmentation program described in Chapter 4 represents segmented regions in such a way that it is trivial to read off the neighboring regions and included regions of any region in the segmented image.

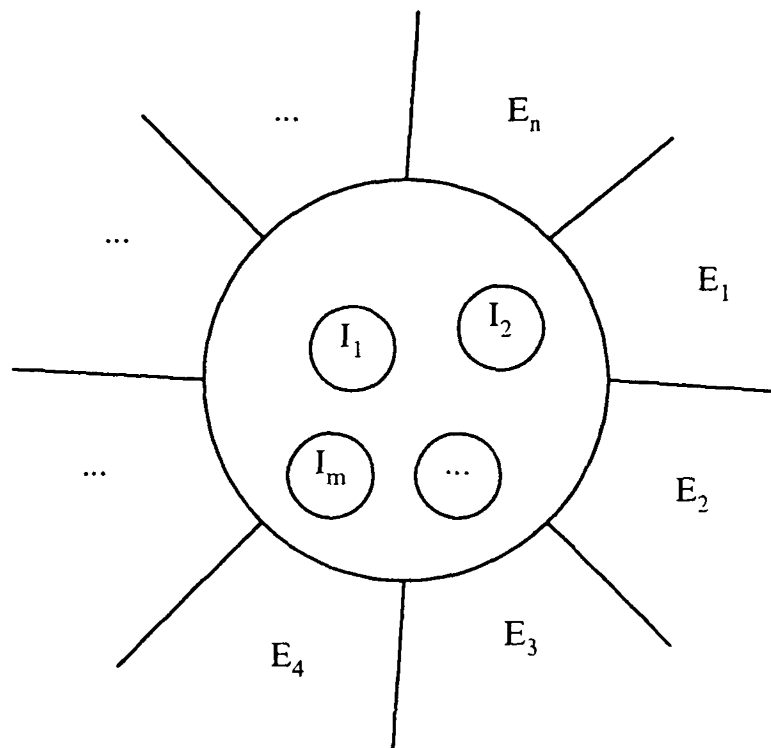


Figure 5.3: Region with internal and external boundaries

Figure 5.3 depicts a stylized region. The region may have internal "holes" which are themselves regions $(I_1 \ldots I_m)$ and may have one or more external neighbors $(E_1 \ldots E_n)$.

In this case there is a single main region with included regions. The regions that surround the main region are the "context".

To be useful the picture meta-grammar should deal with the following two issues:

1. The representation should be rotationally invariant. Since the image can be viewed from any orientation (from the satellite or airplane that took the picture) the

representation should not impose a direction on the regions.

2. The representation should be able to handle obscuration. There are two important forms of obscuration that must be addressed in the context of aerial imagery. The first is the image boundary issue described above. The second form of obscuration comes from fog and cloud.

The external neighbors of a region can be described by reading them off from the region boundary. Since the boundary is connected, there is no start or end point. Our original formulation of a patchwork grammar involved reading the neighbors off in clockwise order. This made it possible to distinguish cases where order of neighbors was significant. In our data there were no cases where clockwise ordering was significant, so the order of matching only had the effect of doubling the number of rules and putting a greater burden on smoothing, both of which make matching more expensive.

It seems unlikely for aerial images that clockwise versus anti-clockwise ordering would ever be useful so we ended up making clockwise/anti-clockwise ordering of neighbors irrelevant. Region order could be useful if we wanted to perceive handedness at the syntactical level as a form of constraint satisfaction (Waltz 1975). The advantages of things like handedness are likely to be of more use when extending this work to parsing 3D scenes such as might result from segmenting images from a mobile robot.

The rule for the region depicted in figure 5.3 is:

$$P(\text{EXAMPLE1}\ (E_1\ E_2\ E_3\ E_4 \ldots E_n) \rightarrow \text{region1}\ (I_1\ I_2 \ldots I_n)) = n \qquad (5.15)$$

where "EXAMPLE1" is the nonterminal defined by the rule and "region1" is the main region (represented by the large circle).

Structured regions

Figure 5.4 shows an important special case of syntactic category, in which the internal neighbors occupy all of the pixels of the region.

There are two important consequences of this kind of structure. The first is that the non-terminal becomes purely syntactic since there are no pixels that support the region from the optical model. The second is that the external border of the region is not simply read off from the segmentation boundaries as in the simpler case but must be constructed from the boundaries of adjacent regions. There are intermediate cases in which some but not all of the internal regions touch external regions directly.

Structured regions are formed by collections of adjacent regions. To the extent that a city often consists of an urban area, a suburban area, and a river, the collection of such things forms a grammatical rule for "city". At this point our meta-grammar looks

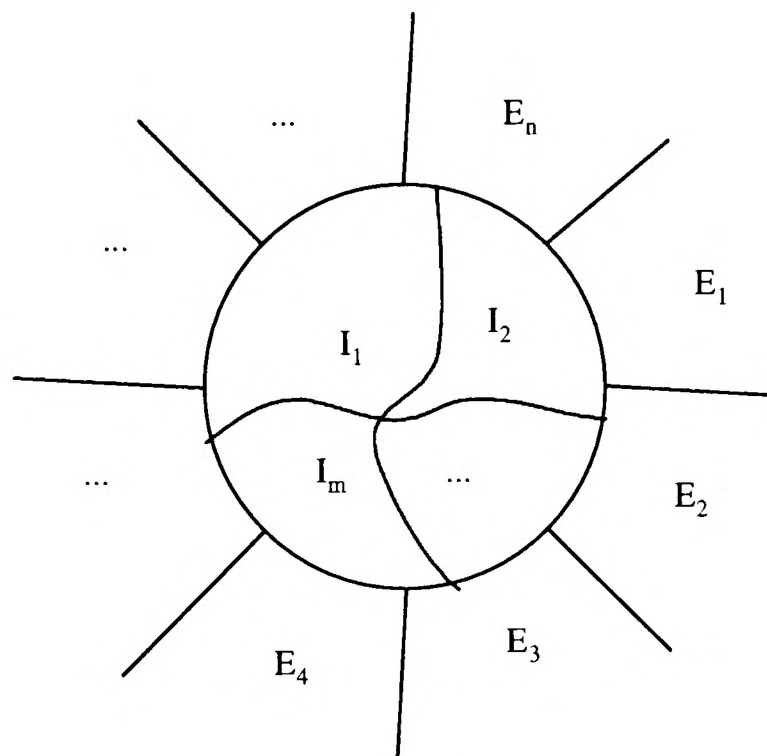


Figure 5.4: A structure region

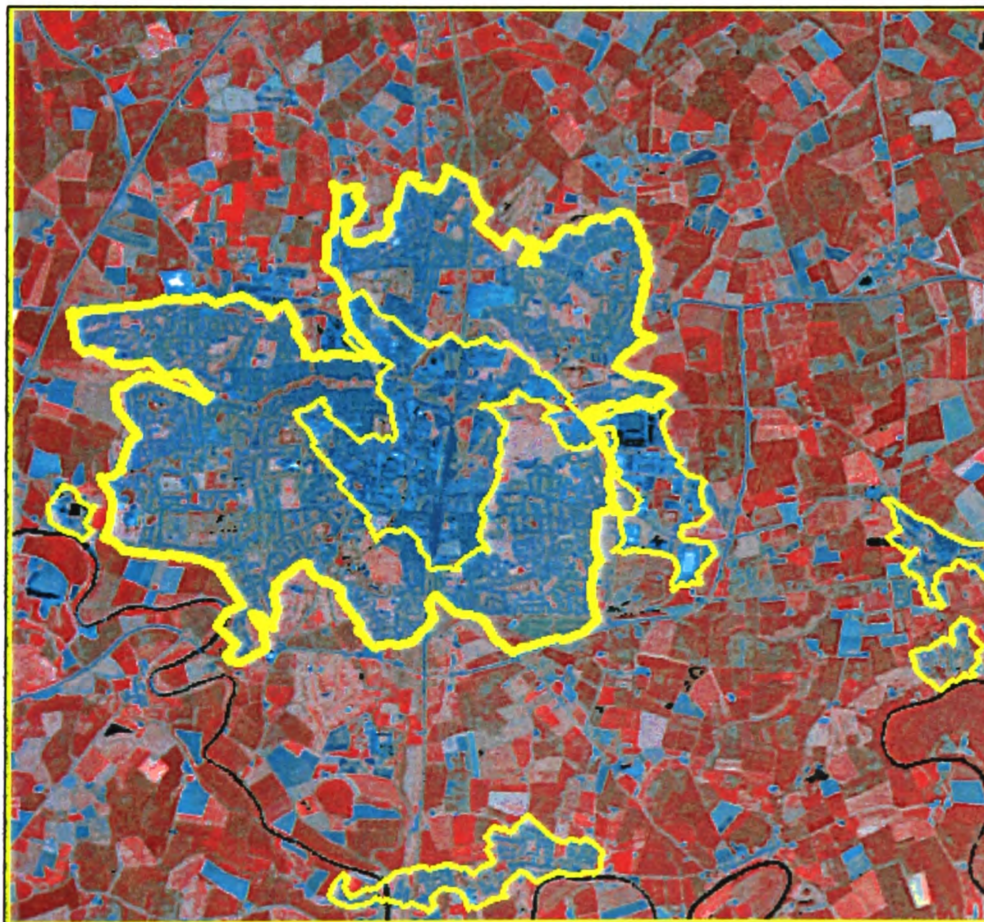


Figure 5.5: Example of a Structured Non-Terminal

more recognizable as a traditional grammar since now non-terminal symbols can be constructed out of an aggregation of other terminal and non-terminal symbols.

Figure 5.5 is an example of a structured non-terminal symbol. The thick boundary denotes the extent of the non-terminal of the rule. The rule consists of its internal (constituent) parts which in this case are an "urban" region (center) and two "suburban"

regions (above and below). The external regions are "farmland" and "industrial" (small region to the right). The rule for this is represented as follows:

$$P(\text{CITY (farmland industrial)} \rightarrow \text{() urban suburban suburban)}) = n \tag{5.16}$$

This structured non-terminal would be a simple non-terminal if the urban region didn't cut the suburban regions into two parts. If the suburban region completely surrounded the urban region center the non-terminal would be represented as follows:

$$P(\text{CITY (farmland industrial)} \rightarrow \text{suburban (urban)}) = n \tag{5.17}$$

Internal regions occur within the two dimensional constraints of the primary. No obvious ordering is available for internal neighbors so order is unimportant.

Obscured regions

Any region that has a cloud or image boundary as a neighbor is considered to be an obscured region.

Figure 5.6 depicts the region obscuration caused by image boundary or cloud cover. Obscured regions can affect the patchwork parsing problem in two ways:

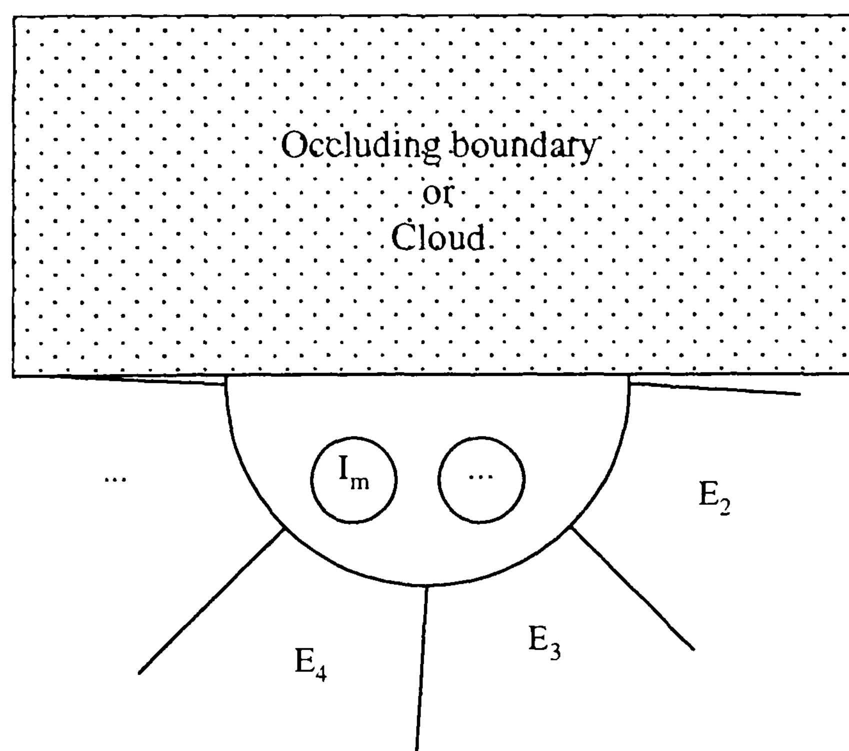


Figure 5.6: Partially Obscured Region

1. The obscuring boundary or cloud may keep hidden the region being hand parsed for inclusion in the image bank.

2. The obscuring boundary or cloud may keep hidden the region that is being parsed.

The first of these affects the rules that can be induced from a hand parsed corpus of images, while the second affects our ability to apply parse rules to regions that fall on the edge of an image or partially obscured by cloud cover.

One approach to this problem is to disallow regions that are obscured by clouds or image boundaries from participating in the parse either as a source of parse rules or as components of a successful parse. Unfortunately such a restriction is too limiting since in many cases the objects of interest are partially obscured in such a way; and some objects are usually obscured by virtue of their spatial extent. Rivers and highways for example typically have at least one image boundary.

Obscuring boundaries are essentially *wildcards* for the sequence of boundaries that would replace them if the object of obscuration was removed. This is almost symmetrical to the situation in which we have a neighboring region about which we have no knowledge.

We expect image parsing to provide a probabilistic context that helps to constrain what the unidentified region might contain. For example, when we see a lake with fields in front of it but obscured at the back by the image edge, we expect that the next frame (which exposes what is now hidden) will reveal more fields beyond the lake because that supports the most probable parse.

The picture meta-grammar contains the special symbol "*" to denote an obscured boundary. We would represent the rule for Figure 5.6 as follows:

$$P(\text{EXAMPLE2 } (* \; E_2 \; E_3 \; E_4 \ldots E_n) \rightarrow \text{region1 } (* \ldots I_n)) = n \tag{5.18}$$

All regions with an obscuring boundary have an internal wildcard because there may be internal regions in the obscured part of the region.

Grammar Induction

Figure 5.7 depicts an image divided into nine regions that have been labeled as lake, field, road, river, town, and swamp. We use this synthetic example first to illustrate parse rule induction and then to illustrate the parsing of the image using the rules.

$$\begin{aligned}
P(\text{FIELD(road *)} &\rightarrow \text{field(lake *)}) = 0.333 \\
P(\text{FIELD(* road town river)} &\rightarrow \text{field(*)}) = 0.333 \\
P(\text{FIELD(* river swamp)} &\rightarrow \text{field(*)}) = 0.333 \\
P(\text{SWAMP(* field river)} &\rightarrow \text{swamp(*)}) = 0.5 \\
P(\text{SWAMP(* river town road)} &\rightarrow \text{swamp(*)}) = 0.5 \\
P(\text{RIVER(* field town swamp)} &\rightarrow \text{swamp(*)}) = 1.0 \\
P(\text{TOWN(field road swamp river)} &\rightarrow \text{town()}) = 1.0 \\
P(\text{LAKE(field)} &\rightarrow \text{lake()}) = 1.0
\end{aligned} \tag{5.19}$$

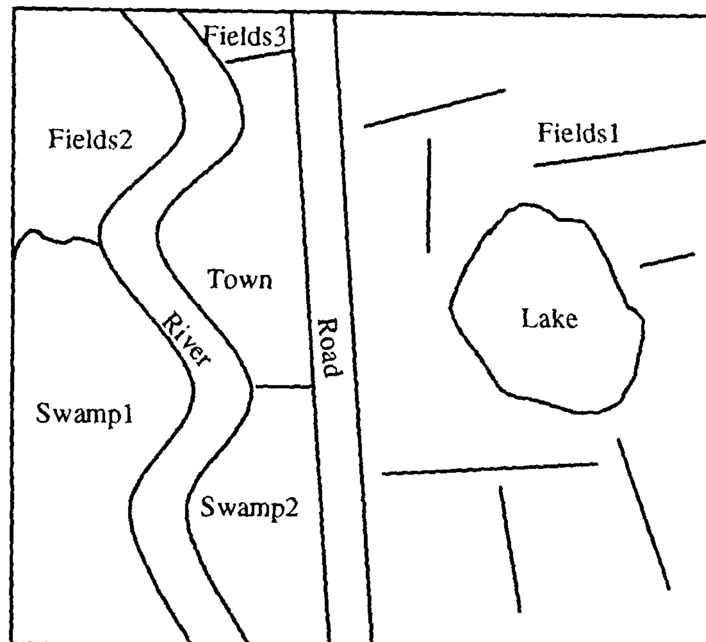


Figure 5.7: A segmented image

The probabilities are calculated by taking all rules with a matching non-terminal and distributing the total probability according to the frequency. In the example given, only a single image has been used to generate the rules and probabilities; but with a reasonable sized image bank, many rules would occur multiple times. These probabilities are the probabilities that, given the non-terminal the context and inclusions will correspond to the internal and external neighbors of the region.

Figure 5.8 shows an image that has been annotated as part of the color corpus. The rule induction software automatically extracts the following rules from the image:

```
("S-UK-23-KINGS-C-2.tif"
 (R666 (R20) ())
 (R666 (R3 R20) ())
 (R3 (R4 R20 R666 *) (*))
 (R4 (R10 R3 R20 *) (*))
 (R4 (R10 R20 *) (*))
 (R3 (R10 R20) ())
 (R3 (R20) ())
 (R3 (R10 R20 *) (*))
 (R31 (R10 R20 *) (*))
 (R18 (R10) ())
 (R10 (R31 R3 R20 R4 *) (R18 *))
 (R20 (R666 R10 R31 *) (*)))
```

The key to the region types is given in Appendix A. The above is a compact representation of the following rules:

Figure 5.8: Grammar Induction from an Annotated Image

$$
\begin{aligned}
P(\text{UNKNOWN(farmland)} &\rightarrow \text{unknown()}) = 0.5 \\
P(\text{UNKNOWN(suburban farmland)} &\rightarrow \text{unknown()}) = 0.5 \\
P(\text{SUBURBAN(urban farmland unknown)} &\rightarrow \text{suburban()}) = 0.25 \\
P(\text{SUBURBAN(river farmland)} &\rightarrow \text{suburban()}) = 0.5 \\
P(\text{SUBURBAN(farmland)} &\rightarrow \text{suburban()}) = 0.25 \\
P(\text{URBAN(river suburban farmland)} &\rightarrow \text{urban()}) = 0.5 \\
P(\text{URBAN(river farmland)} &\rightarrow \text{urban()}) = 0.5 \\
P(\text{TOWN(river farmland)} &\rightarrow \text{town()}) = 1.0 \\
P(\text{ISLAND(river)} &\rightarrow \text{island()}) = 1.0 \\
P(\text{RIVER(town suburban farmland urban *)} &\rightarrow \text{river(island *)}) = 1.0 \\
P(\text{FARMLAND(unknown river town)} &\rightarrow \text{farmland()}) = 1.0
\end{aligned}
\tag{5.20}
$$

In the rules above we used the frequencies of occurrence to calculate the probabilities.

Usually a much larger collection of rules drawn from a large corpus would be used to compute the probabilities.

Smoothing

Parsers that depend upon parse rules and probabilities derived from a corpus run a risk when interpreting a novel image: a rule may be needed that was not in the image bank. To avoid having the parse simply fail, various techniques are used to estimate probabilities for the missing items. For example, in part-of-speech tagging of natural language, words that have never been seen before may be encountered. To deal with this, tricks like looking at the word endings can be used to guess the part of speech that is appropriate for the word. This kind of heuristic method for missing words or rules is known as smoothing.

We now turn our attention to smoothing for missing patchwork parse rules. The parse rule structure that we have defined for patchwork images allows quite complex rules to be generated from an image bank. The likelihood that a particular rule will be missing from our rule bank is therefore quite high. We need a way of dealing with image structure that comes close to matching a rule but which doesn't actually match any particular rule in the grammar.

The seriousness of the problem is evident by looking at what percentage of the rules are missing in the test set when trained on the training set[2]. Using the color corpus we find that approximately 50% of the rules required to parse the test set are missing. Without smoothing then, we would not be able to parse the test set.

In smoothing, we wish to assign a non-zero probability to missing rules. This probability must be low enough not to interfere with parses that use rules found in the corpus but still be above zero in order for the parse to succeed. If smoothing is used very infrequently it may be acceptable simply to assign an extremely small probability to the smoothed element. Generally a statistical parser will perform better if the probabilities used are approximately correct. In view of this we construct a smoothing lattice to provide smoothing for missing rules. Smoothing is a very important part of any statistical parser. Performance of statistical parsers is highly dependent upon how good the smoothing algorithm is.

So far we have considered collecting the rules by non-terminal type, counting occurrences of the rules, and assigning probabilities to the rules so that they sum to 1. We still require that the probabilities of the rules sum to 1, but this must now include all missing rules that may be added by smoothing.

Consider the rule for town from our example:

[2]The color corpus of 105 images was arbitrarily divided into a training set of 94 images and a test set of 11 images. The training set of 94 images was used to induce the grammar and learn optical models

$$P(\text{TOWN (field road swamp river)} \rightarrow \text{town ()}) = 1.0 \quad (5.21)$$

The set of neighbors (field road swamp river) can be viewed as a proposition about the neighbors of the region town. If Θ is the set of all possible sets of neighbors of the region town, there are 2^{Θ} propositions that we can make about the neighbors of town. For example, if town could be surrounded by a road or a field $\Theta = \{road\ field\}$, the set of possibilities for the neighbors of town Θ would be $\{\emptyset\ \{road\}\ \{field\}\ \{road\ field\}\}$. We can represent our belief in each of the possible neighborhoods of a region with a belief function.

$$\begin{aligned} Bel(\emptyset) &= 0 \\ Bel(\Theta) &= 1 \end{aligned} \quad (5.22)$$

Belief in certain propositions can be estimated by the frequency they occur in the corpus. This approach of viewing propositions as sets in reasoning about probability was developed by Dempster and Shafer (Shafer 1976) and provides a way of generating plausible probability estimates for possible rules that have not been encountered in the corpus. If you want to find a rule with neighbors (road river) for example, all propositions in Θ that contain (road river) could be found and accumulated to produce the total support for that missing rule.

The smoothing lattice described below uses the problem definition to arrive automatically at the set Θ and then arranges the proposition into a lattice so that the Shafer rules of evidence can be calculated straightforwardly by propagating instances from the corpus through the lattice and then counting the number of times a node is visited.

Consider ways that the similar rules may vary from a rule with neighbors (field road swamp river):

1. All of the fields can be specified in the rule but additionally have one or more neighbors.

2. One or more of the required neighbors may be missing.

We can hypothesize rules using wildcards to deal with these cases as follows:

$$
\begin{aligned}
&\text{TOWN(field road swamp river *)} \rightarrow \text{town()}\\
&\text{TOWN(field road swamp river)} \rightarrow \text{town()}\\
&\text{TOWN(field road swamp *)} \rightarrow \text{town()}\\
&\text{TOWN(field road river *)} \rightarrow \text{town()}\\
&\text{TOWN(field swamp river *)} \rightarrow \text{town()}\\
&\text{TOWN(road swamp river *)} \rightarrow \text{town()}\\
&\text{TOWN(field road *)} \rightarrow \text{town()}\\
&\text{TOWN(field swamp *)} \rightarrow \text{town()}\\
&\text{TOWN(field river *)} \rightarrow \text{town()}\\
&\text{TOWN(road swamp *)} \rightarrow \text{town()}\\
&\text{TOWN(road river *)} \rightarrow \text{town()}\\
&\text{TOWN(swamp river *)} \rightarrow \text{town()}\\
&\text{TOWN(field *)} \rightarrow \text{town()}\\
&\text{TOWN(road *)} \rightarrow \text{town()}\\
&\text{TOWN(river *)} \rightarrow \text{town()}\\
&\text{TOWN(swamp *)} \rightarrow \text{town()}\\
&\text{TOWN(*)} \rightarrow \text{town()}
\end{aligned}
\tag{5.23}
$$

Whenever a rule is read from the corpus, the lattice for the rule is added into the existing lattice and all nodes visited by the rule have their frequency incremented. Imagine that we encounter the following rule for town in our corpus as follows:

$$\text{TOWN(field road swamp river)} \rightarrow \text{town()} \tag{5.24}$$

We can represent this as a lattice as shown in Figure 5.9.

We now must assign probabilities to each of the rules of the lattice in such a way that they sum to 1. We can no longer use the frequency counts the way that we did before because that way the top rule containing just (*) would be the most probable rule.

At each node in the lattice, we can think of the node representing a family of related rules. We consider just the external neighbors in this explanation; but the idea extends trivially to the internal neighbors as well.

Consider the node containing the rule with external neighbors (Field River *). This can be expanded as follows:

1. (Field River Road), (Field River Swamp), etc. Completely specify each rule.

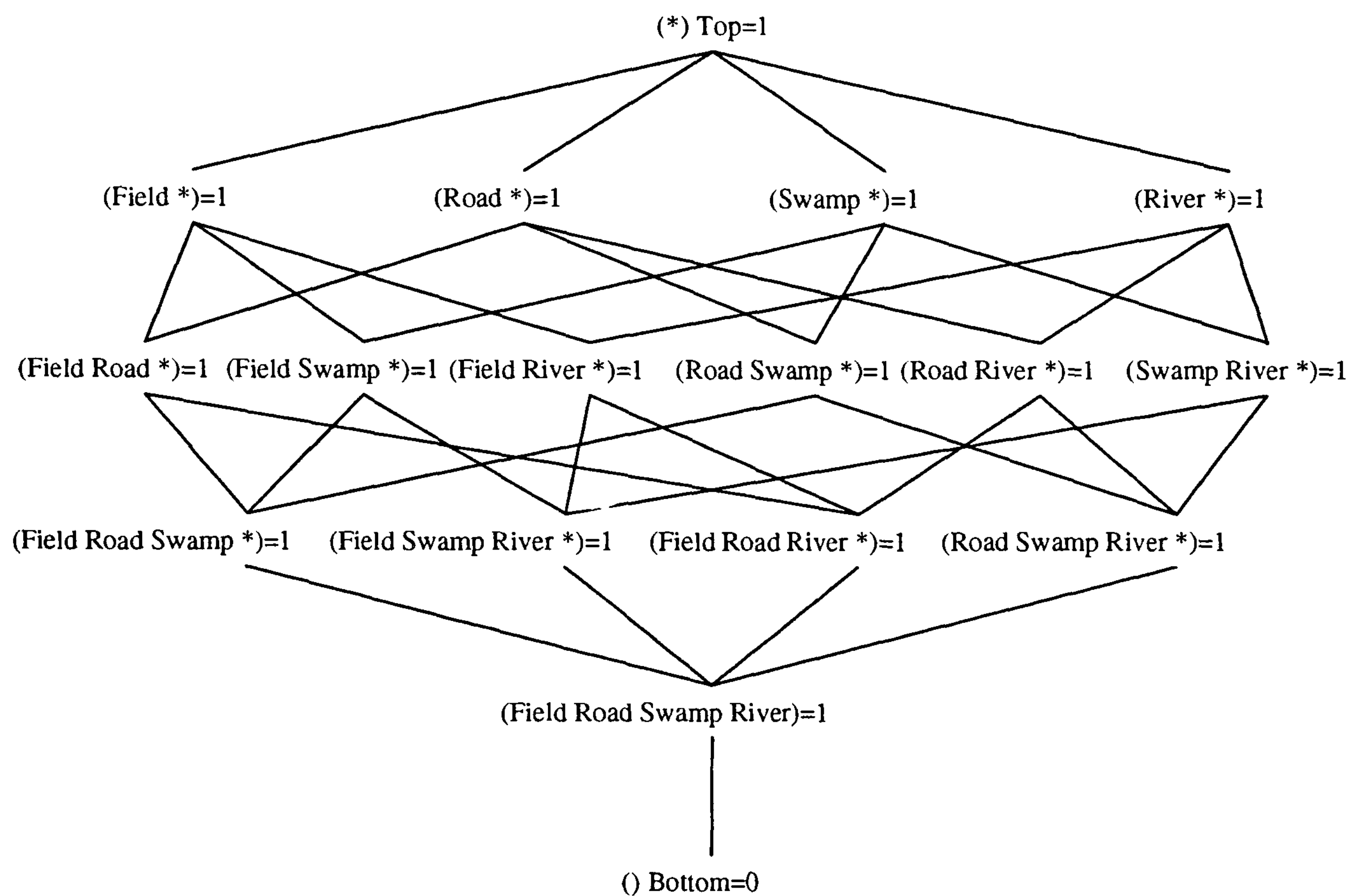


Figure 5.9: Smoothing Lattice for (field road swamp river)

2. (Field River $*_1$) All regions contain Field, River and one other region of any kind (other than Field or River).

3. (Field River $*_2$) All regions contain Field, River and two other different regions of any kind except Field or River.

4. Finally, (Field River *) is the parent of all of the (Field River $*_1$) ... (Field River $*_n$) rules.

If there are n region types defined in our corpus, we know that $*_1$ can match ${}^{n-2}C_1$ different region types. $*_2$ can match ${}^{n-2}C_2$ regions etc. Completely specified rules (without wildcards) such as (Field River Road) can only match one arrangement of regions. (Field River *) can match as many configurations as:

$$\sum_{i=1}^{n-2} {}^{n-2}C_i \tag{5.25}$$

The smoothing lattice shown in Figure 5.9 therefore needs to be elaborated further in order to support calculation of smoothed probabilities. Each node containing a wildcard must be elaborated to include the tree of expansions for *. For example, the node for (Road *) needs to be replaced with the sub-tree shown in Figure 5.10. Of course this can be compactly represented at the unelaborated nodes by keeping track of the number of regions that each corpus wildcard matches against.

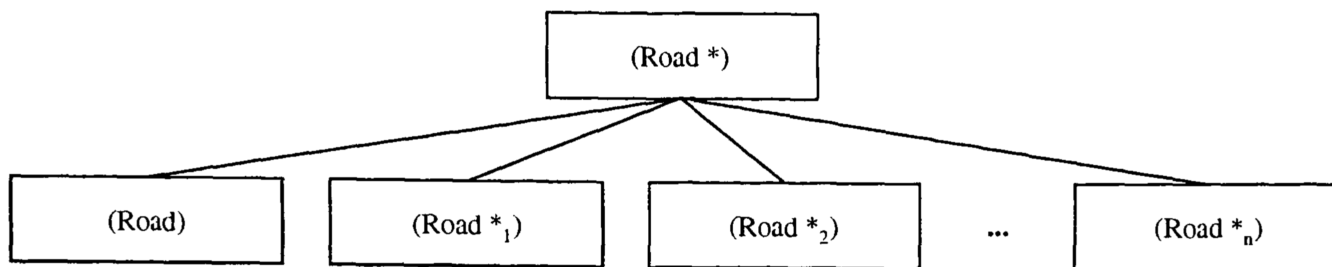


Figure 5.10: Expansion for nodes with wildcards

In this way we can uniquely define the number of different matches available for a rule. Rules without wildcards can match precisely one rule, whereas rules with a wildcard can match a larger number of possibilities as defined above. All of the rules of the lattice must sum to 1. The probability of any rule in the lattice is the number that is collected at the node as cases found in the corpus divided by the number of patterns that the node can match. Finally all nodes are divided by the sum of all of the nodes so that the sum of all of the rules comes to 1.

This formulation of the probabilities of nodes in a lattice of rules provides probabilities for rules that have not been evidenced by the corpus but which satisfy the following properties:

1. If there are n rules in the lattice that match a given pattern of neighbors, the most specific rule will have the highest probability assignment.

2. A rule that has been encountered in the corpus will always have a higher probability assignment than a rule that matches the same pattern and has been smoothed. This follows directly from (1).

3. If there are n rules in the lattice that match a given pattern and if the n matching rules are grouped by specificity, all rules in the group of level n specificity will have a higher probability assignment than all rules in the next lower group; and they will all have a lower probability than all of the rules at the next higher level.

These smoothing rules allow us to parse any image by giving uncited rules a low probability while retaining the crucial property that rules derived from the corpus are more representative of the images that will be encountered.

5.3.4 Parsing an image using rules extracted from an image bank

The parse of the image in Figure 5.7 consists of the image as the root node and the unordered list of its constituent parts. Parts with nested constituents produce a tree structure.

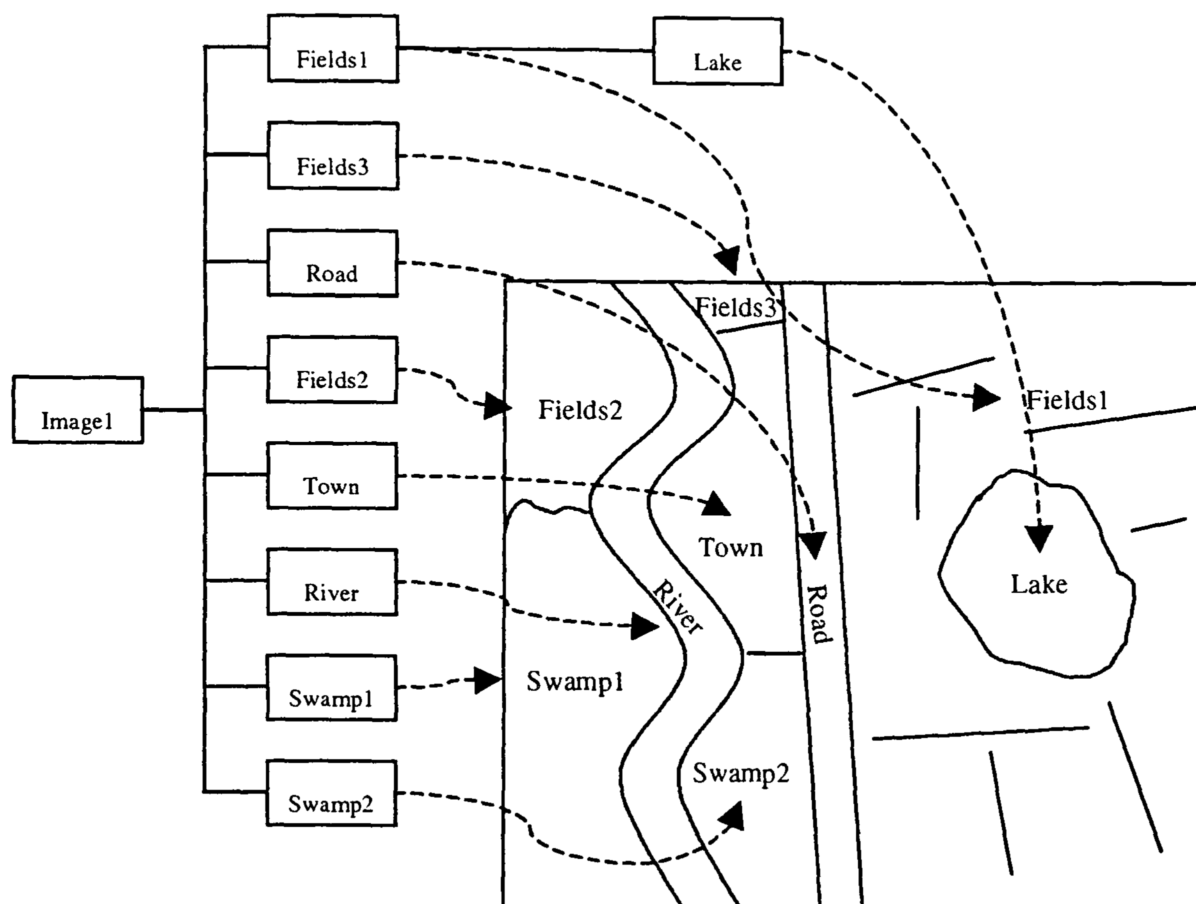


Figure 5.11: An example parse

Figure 5.11 shows the result of parsing the image in Figure 5.7. While the structure is determined by the initial region layout, the application of parse rules and hence the labeling of the regions is not. In the foregoing we examine algorithms for computing probable image labelings and the parse tree.

5.4 Algorithms for Patchwork Parsing

It is here that we part company with much of the work on speech recognition. In spoken or written natural language the sequence of words can be viewed as a left-to-right or right-to-left linear sequence that can conveniently be viewed as a hidden Markov model (HMM) (Bahl, Jelinek, & Mercer 1983). The more interesting natural language grammars do not yield to the efficient algorithms and neither does the context sensitive grammar described above. Below we describe the straightforward algorithm because of its clarity and then describe how the parser can fit into the Monte-Carlo infrastructure of the GRAVA architecture.

Network Traversal Algorithm

Finding the best parse of the image using the context-sensitive grammar described above must achieve the following:

1. Every region of the image must be accounted for in the parse.
2. Every region of the image must be assigned to the right hand side of one and only one rule (either primary or included).
3. Context constraints must be observed for all rules used in the parse.
4. A region must be interpreted as the same kind of region for all rules in which it participates.
5. Of all the possible parses of the image the one with greatest probability must be selected (Equation 5.10).

Included regions usually occur within a surrounding region. Structured regions must still be parsed in a conventional sense but the majority of structure comes for free from the segmentation. Despite the "free" structure that comes from embedded regions in the segmentation, the parsing problem is computationally expensive.

Each terminal region must appear exactly once in the resulting parse. The probability of the parse, which is the product of all the component rules and optical interpretations in the resulting description, can be computed incrementally. By taking the regions in arbitrary order, and at each region inserting the region into the parse structure, when the last region has been added the resulting parse is a legal parse of the scene. If this was done depth first it would find the parses in arbitrary order and we would have to find them all before we could determine which was the best parse. We can find the parses in best first order by performing the parse breadth first. In terms of MDL the description lengths of the components of the parse can be computed as:

$$-log_2(P(l_i|r_i)P(n_i|l_i)) \quad (5.26)$$

The parse state is represented by a list of 3-tuples of the form:

$$< bindingList, partialParse, descriptionLength > \quad (5.27)$$

The parse state list is sorted in ascending order of description length (most probable first).

The first element of the three-tuple is a region identification binding list (bindingList). The binding list is initially empty but as commitments are made as to the identification of regions the pair $< region_i, label_i >$ are added to the binding list. At the end of the parse every region will have precisely one binding in the binding list.

The second element is the path of terminals or non-terminals not yet included in the parse (partialParse). The path is represented as a list. At first the list is initialized to

the list of all regions in the segmented image. At the end of the parse the list consists of the parse of a single non-terminal (N^1) which is the parse tree for the entire image.

The third element is the cumulative description length of the partial parse (partialParse). The starting description length is 0.0 since nothing has been described yet. As uninterpreted regions in the partial parse list are replaced by non-terminals the description length is increased by the amount computed in equation 5.26.

In order to find the parses in the order of most probable first, we employ a pseudo parallel algorithm. The parse begins by setting the parse state list to a list of a single starting three-tuple as follows:

$$(<(), (R_1 \ldots R_n), 0.0>) \tag{5.28}$$

At each point in the parse the partial parse is extended by taking the parse that currently has the shortest description length (by popping it from the pre-sorted parse state list) and replacing it by taking the first element of the remaining path and finding all possible partial parses for that path.

This is implemented by the following algorithm.

1. Initialize the parser-state-list:

$$parseStateList \leftarrow (<(), (R_1 \ldots R_n), 0.0>) \tag{5.29}$$

2. Exit the parse if the partial parse list is empty (there are no more parses). Otherwise pop the best partial parse from the parse state list:

$$\begin{aligned} &mainloop: \\ &if\ \ empty(parseStateList)\ \ exit() \\ &else\ \ bestParse \leftarrow POP(parseStateList) \end{aligned} \tag{5.30}$$

3. If the partial parse of the best parse contains a single nonterminal that matches the starting symbol N^1, emit the parse and continue back to "mainloop". Otherwise pop the next element of the partial parse list:

$$\begin{aligned} if\ \ (isParseOf(N^1, first(bestParse.partialParse)))\ \ emit(bestParse) \\ else\ \ nextElement \leftarrow POP(bestParse.partialParse) \end{aligned} \tag{5.31}$$

4. If the nextElement is an uninterpreted region find all interpretations of the region supported by the optical evidence.

$$\begin{aligned} &if\ \ uninterpretedRegion(nextElement) \\ &\quad candidates \leftarrow candidateLabelList(opticalEvidence(nextElement)) \\ &else\ \ candidates \leftarrow nextElement \end{aligned} \tag{5.32}$$

5. Find all rules that are applicable to the current candidates. Rule matching involves: for every candidate interpretation finding all rules that have "candidate" as the primary (simple rules) and all rules that have a null primary and "candidate" as one of the inclusions (structured rules). For each rule found check that all of the context (external boundaries) match. Check also that all included regions match. Matching can involve binding previously unbound regions. Finally description lengths are calculated for the rule and for the candidate interpretation of the region, a new partial parse state is constructed, and a new binding list is constructed that contains any regions bound by the match. The rule, candidate, binding list, partial parse, and the description lengths are gathered up into a structure and returned from the rule matching procedure. Rules that fail to match are not included in the matching rules list.

$$rules \leftarrow findAllMatchingRules(candidates) \tag{5.33}$$

6. For each matching rule compute the new description length and add a new three-tuple to the parse state list at the place that maintains ascending order of description length.

$$\begin{aligned}
&for\ each\ rule\ in\ rules\ do \\
&\quad currentDL \leftarrow bestParse.descriptionLength \\
&\quad ruleDL \leftarrow rule.descriptionLength \\
&\quad labDL \leftarrow rule.candidate.descriptionLength \\
&\quad newDL \leftarrow currentDL + ruleDL + labDL \\
&\quad newTuple \leftarrow makeTuple(rule.bindings, rule.newPartialParse, newDL)) \\
&\quad parseStateList \leftarrow addCandidateInSequence(parseStateList, newTuple)
\end{aligned} \tag{5.34}$$

7. Go to "mainloop" (step 2).

Since the parse with the minimum description length is extended at each iteration, the first parse to have reached the end of the path and come to the head of the partial parse list is the most probable (MDL) parse. By continuing to iterate, all legal parses are generated in ascending order of description length.

Although this algorithm is the most obvious and works well for small numbers of regions its cost is too high to be practical.

If there are on average l valid labelings for a region and r rules that expand each labeling, the tree of parse possibilities grows at each node by a factor of lr, so that for an image consisting of n regions, there are lr^n parses. Many of these parses are not legal,

but the cost of the algorithm to find all parses in order is k^n; and for realistic values of k and n this is unmanageable.

Running this algorithm on the synthetic test example in Figure 5.7 with fields2, river, swamp1, and swamp2 removed yields the following result.

```
> (parseImageNS *5-regions* *rules* *region-optical-evidence*)
(((im . image1) (l1 . lake) (f1 . field) (tn . town) (rd . road)
  (f3 . field)) {the ParseTree 27845} 9.6298)
```

The output of the first parse from the parser is the three-tuple implemented as lists. The first part is the binding list that shows that the correct region assignments have been made. The second part is the parse tree and finally the cumulative description length for the parse.

The parser has correctly identified the most probable parse and labeled all regions correctly. Unfortunately running this algorithm with all nine regions fails to terminate in reasonable time. If we had a realistic sized corpus with a large set of rules and an image with a realistic number of regions it would not produce a result in reasonable time.

Images in the SPOT satellite corpus (see Appendix A) have an average of 18 regions; and the MassGIS images have an average of 97 regions, making the network search parse algorithm infeasible. If we relax our requirements for the parse slightly, however, we can achieve a good parse quickly. Below we modify the parse algorithm to find good parses by random sampling using the GRAVA MDL agent architecture.

Implementing the Parser as GRAVA Agents

By relaxing our requirements we are able to recast the parse algorithm within the MDL agent architecture outlined in Chapter 3 and let the Monte-Carlo search algorithm find approximations to the best parse. The algorithm (below) is similar to the one given above but instead of searching in a breadth first manner in order to find the MDL parse first, the algorithm picks rules and region interpretations randomly in proportion to the probability of those rules and the probability of the region interpretations reported by the optical model. When a complete parse has been selected at random in this way it is not guaranteed to be the most probable parse; but if the sequence is repeated often enough, the most probable parse will be generated more often than less probable parses. In this way the most probable parse can be approximated to an arbitrary accuracy by repeatedly generating more legal parses.

Unlike the network search algorithm that extended all paths in parallel in order to find the best path first (breadth first), the Monte-Carlo parser seeks a correct parse with a depth first search. At each branch point in the search it chooses randomly and finally returns the first parse that succeeds. Because the random choices are weighted by the

probabilities of the rules, more likely parses will occur more frequently than less probable parses. Every time the code is run, a new sample (legal parse) is computed and saved. The best current parse is then selected and returned.

We implement the parser using GRAVA agents. Each grammar rule is instantiated as an agent. The statistical patchwork parser agent has a parse rule that it applies to the segmented image. Each rule of the grammar is implemented as a separate model. The structure of the PWGrammar model (shown below) mirrors the structure of a rule (5.13).

```
;;; Model for patchwork grammatical rule.
(defineClass PWGrammar (Model)
  ((nonterminal) ; Nonterminal name
   (context) ; External regions.
   (primary) ; Primary region--or NIL
   (included) ; Included regions.
   (descriptionLength) ; -log2(P(this rule|NT))
   (occurrences)) ; frequency in corpus.
  documentation "Model for a grammar rule")
```

The result of a successful application is a parse tree structure:

```
(defineClass ParseTree (DescriptionElement)
  (...(rule)   (region)   (inclusions)))
```

The parser agent implements the "fit" method by fitting a grammar rule to the segmented image.

```
(defineClass StatisticalPatchworkParser (Agent)  ...)

(define (fit StatisticalPatchworkParser|ag data)
  ;; Described below
  ...)
```

The StatisticalPatchworkParser agent is part of the ImageParserLayer:

```
(defineClass ImageParserLayer (Layer)
  ((parseStateList) ...))
```

The layer interpreter applies the parser agents to the segmented image and resamples as necessary. The layer interpreter manages the parser state list which as before is initialized to:

$$parseStateList \leftarrow (< (), (R_1 \dots R_n), 0.0 >) \tag{5.35}$$

Figure 5.12 shows a schematic view of the ImageParserLayer. The layer interpreter maintains the partial parse state and the binding list and sequences the components of the

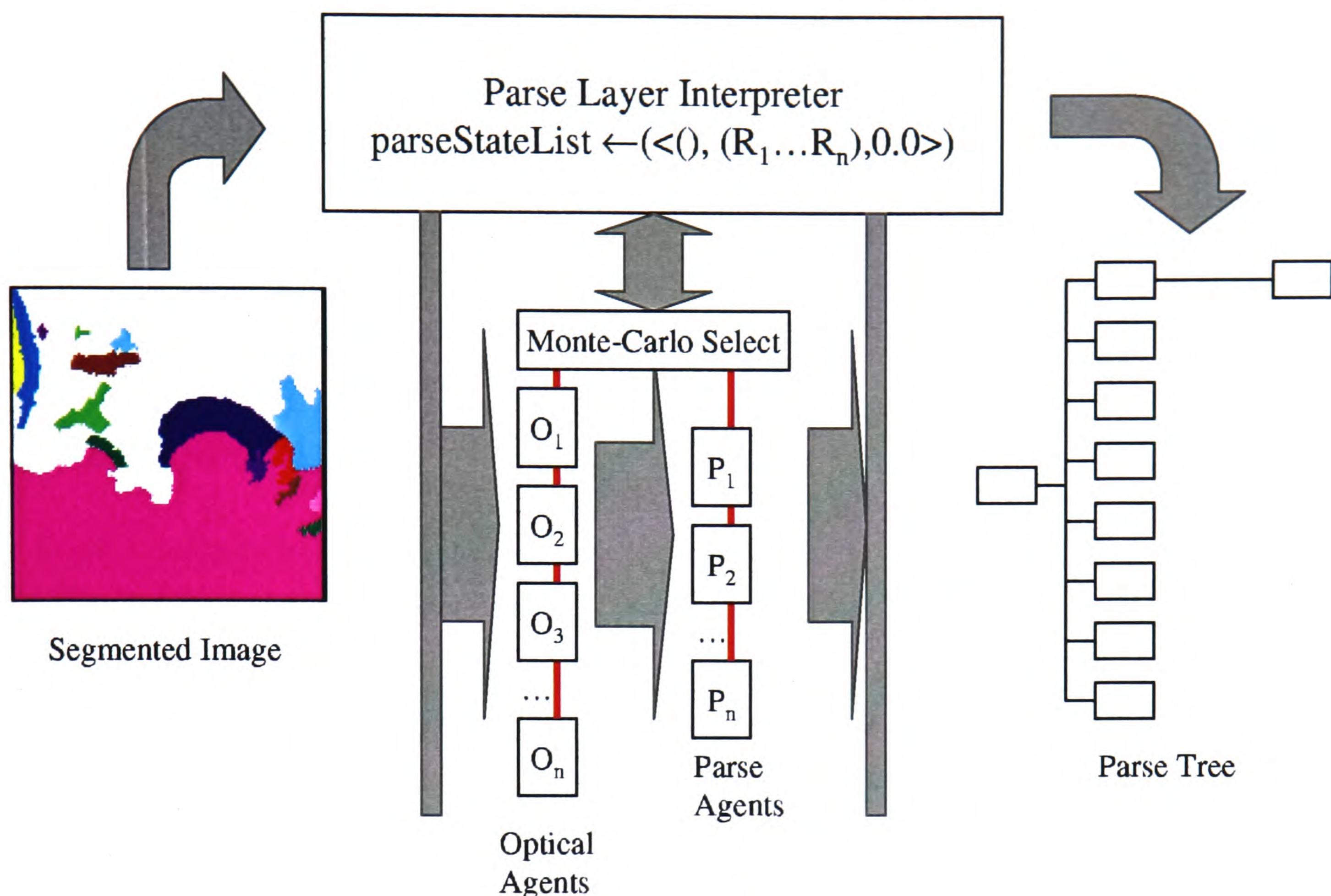


Figure 5.12: Interpret Segmented Image as a Parse Tree

segmentation and partial parse through the agents that make up the layer. The layer has just two kinds of agent—optical interpretation agents and parser agents. The interpreter mediates which agents result will be used in the parse by using "Monte-Carlo-Select". Each agent is controlled by the interpreter to perform its "fit" operation. The agents that satisfactorily "fit" their model to the data announce their result and description length and the Monte-Carlo select algorithm selects which agent's contribution will be selected. The optical agents (O_1, O_2, $\ldots O_n$) attempt to label the regions. The Parse agents (P_1, P_2, $\ldots P_n$) attempt to fit a rule to the primary region or non-terminal.

The StatisticalPatchworkParser agent's "fit" method does the following when applied to a segment.

If the label matches the rule's "primary" region type or, if the rule has a null primary region type (a structured rule), if the label matches one of the included components, check that all of the context (external boundaries) match. Check also that all included regions match. Matching can involve binding previously unbound regions. Finally the description length is calculated. The rule, region, binding list, partial parse, and the description lengths are gathered up into a structure and returned from the rule matching procedure. Monte-Carlo-Select selects one of the agent's contributions, updates the parseStateList and continues to cycle through the regions and non-terminals until the parse is complete. This is essentially the same as step 5 of the network search algorithm.

The interpreter maintains the "best" parse and updates it whenever a sample produces a parse with a smaller description length.

Running this algorithm on the full nine region test case to produce a single sample yields the following results:

```
> (run-layer *parseLayer* *all-regions* *rules* *region-optical-evidence*)
(((l1 . lake) (f1 . field) (im . image1) (rd . river)
  (s2 . swamp) (f3 . road)
  (tn . town) (f2 . river) (rv . field) (s1 . town))
   {the ParseTree 32471} 27.8244)
```

After a single iteration the generated parse contains five incorrect labelings out of 9. F3, a field has been incorrectly labeled as a road; RD, a road, has been incorrectly labeled as a river; F2, a field, has been incorrectly labeled as a river; RV, a river, has been incorrectly labeled as a field; and S1, a swamp, has been incorrectly labeled as a town. A further 100 iterations, however, improves matters considerably.

```
> (dotimes (i 100 (show-result)) (resample *parseLayer*))
(((l1 . lake) (im . image1) (s2 . swamp) (f1 . field) (rd . road)
  (tn . town) (f3 . field) (rv . river) (s1 . swamp) (f2 . town))
   {the ParseTree 29125} 19.3325)
```

The description length of the best parse has decreased from 27.8244 to 19.3325 and there is now only one labeling error (out of nine regions)–field2 has been incorrectly labeled as a town. A further 100 iterations yields the following:

```
> (dotimes (i 100 (show-result)) (resample *parseLayer*))
(((f2 . field) (s2 . swamp) (im . image1) (f1 . field) (l1 . lake)
  (s1 . swamp) (rv . river) (rd . road) (tn . town) (f3 . field))
   {the ParseTree 31473} 18.6531)
```

All regions have been labeled correctly; and the description length has been further reduced to 18.6531. Since the parses are selected randomly, the same problem would have different convergence characteristics if parsed again. In this example we know in advance what the correct labeling is and what the best parse is. In general we cannot know if the best parse has been found. Knowing the correct answer allows us in this case to measure the number of errors and description length at each iteration.

Figure 5.13 (left) shows the average description length based on 100 runs. The right figure similarly shows the average number of parse errors based on 100 runs. Together they show the average convergence characteristics for this example over the first 201 iterations.

This is not the only image that we have parsed. We have parsed many images. This is a typical result.

The Monte-Carlo parse algorithm has a number of benefits and failings.

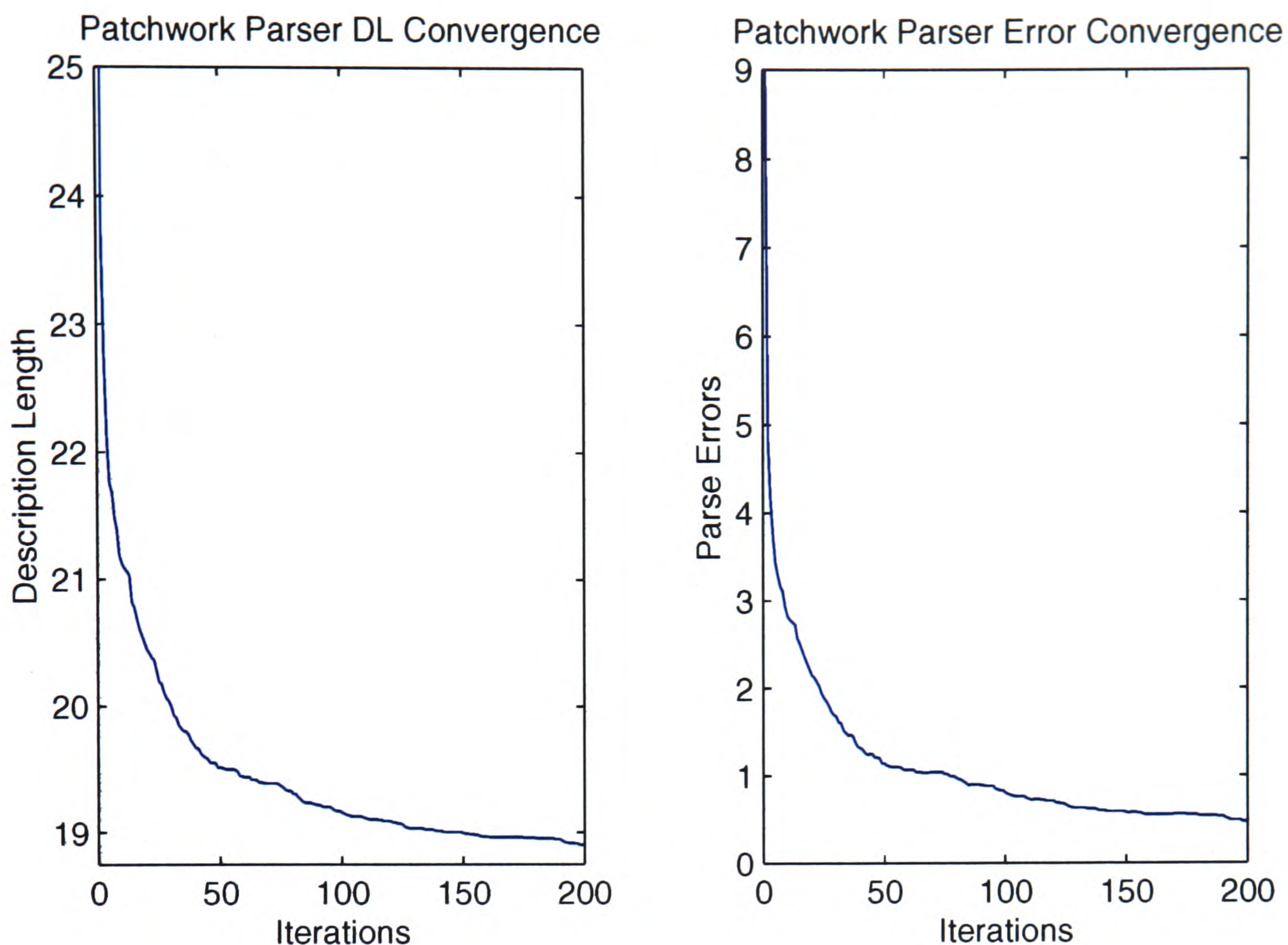


Figure 5.13: Monte-Carlo Parse Convergence

1. *Not all parses are generated.* The network search algorithm guarantees to find all legal parses in ascending order of description length (most likely first). The Monte-Carlo algorithm produces parses at random and may produce the same parse multiple times but may not produce all legal parses for a long time. In particular, we can never be sure that all parses have been generated.

2. *May not yield the most probable parse.* The network search algorithm was guaranteed to find the most probable parse first. All that we can guarantee with the Monte-Carlo parse is that the more iterations we perform the better our approximation to the probability distribution. We can come arbitrarily close to the most probable parse; but we can't actually be certain that we have found it.

3. *A legal parse is achieved very quickly.* The network search algorithm takes a very long time (exponential time) to produce a parse. The Monte-Carlo parser generates legal parses very quickly. This means that in a time constrained situation, it is possible to trade off parse accuracy for time. Despite the complexity of parsing images with many regions the Monte Carlo based parser performs well. The network search algorithm ran for hours on the 9 region test case before we stopped it. The Monte-Carlo algorithm was able to produce good solutions in seconds even though no particular effort was put into optimizing the implementation.

We cannot know in advance how many iterations may be needed in order to get an adequately good parse; however, we can compute the standard error dynamically and continue until it drops below some threshold ϵ.

Traditionally Monte-Carlo methods estimate a PDF by counting occurrences of samples taken at random. In our usage we are not interested in the complete PDF. We are interested only in the most probable result. Since we have description length estimates for all solutions that are sampled it is not strictly necessary to produce enough parses so that we can choose the parse that occurs the most often. Once a parse has been generated we can simply compare its description length against the previous best. This leads to the optimization where the parses can be stored in order of description length rather than in order of frequency.

5.5 Conclusion

There are many parallels between speech/language interpretation and image interpretation. The signal processing, aspects of image interpretation are much more challenging than for speech processing so a significant emphasis has been directed towards low level processing aspects of image analysis. Semantic issues have lagged as a result. It seems likely that many of the approaches to semantics taken by language and speech researchers may be adaptable to use in image interpretation.

Statistical approaches to language and speech understanding have been very successful in recent years. These techniques offer a straightforward approach to semantics (selecting the most likely parse tree) and appear to provide levels of competence close to that of humans for appropriately selected corpora.

We have demonstrated how ideas of statistical parsing can be applied to the parsing of images consisting of patchworks of regions. This approach provides for the mobilization of contextual information and provides a form of filter fusion insofar as contributions from adjacent regions are fused to yield the most probable interpretation. The grammar rules model the relationship between regions that is ignored in segmentation algorithms like region competition.

One of the exciting aspects of the work in natural language has been that, by automatically inducing the parse rules from a hand parsed corpus, the grammar can be taken from the corpus rather than by being specified *a-priori*. This allows the human parsers of the corpus to communicate the grammar implicitly in the way that they parse the corpus.

Since it is our goal to build systems that can interpret images with performance approaching that of human expert photo interpreters, we are concerned with the issues of how to access and represent that human expertise. Knowledge acquisition has been

a major problem in expert system development. The approach taken here allows expert photo-interpreters to annotate a corpus of images and for their expert knowledge to be acquired by the system by extracting the rules and the frequencies from the corpus automatically.

Performance is an issue for any interpretation system. We would like to be able to interpret images and sequences of images rapidly enough to compete with human experts. The search space for parsing real images is huge. Taking rules from a realistic corpus can yield a large collection of rules and the number of segments on a good quality image can also be very large. Consequently brute force parsing of the form described in the network search parser is not realistic.

We have shown, however, how approximate methods can address this issue. In particular we have shown a Monte-Carlo based parser that can parse complex images with a large number of rules with acceptable performance. We cannot ever be sure that we have the best overall parse on an image; but we can get an initial parse very quickly and can subsequently improve the quality of the parse arbitrarily by extending the parse time.

Our approach to smoothing provides good estimates for probabilities that cannot be directly found in the corpus, so that our parser can perform well even when a relatively low percentage of the rules needed to parse an image are actually available in the corpus. Rather than failing to parse an image because of missing rules, we successfully parse the image with a smoothed rule.

It is not difficult to extend the parser to learn new rules. We did not try this due to time limitations but the idea is straightforward. Once a parse has been accepted by the consumer of the parse (no more samples will be taken for the current image), any "smoothed" rules that were used in the parse must be considered good rules because they have been encountered in a real image. The smoothed rules can therefore be explicitly added to the grammar in the same way as the rules extracted from the corpus were added. Thereafter the rule is a first class rule with a higher probability than it had when it was a smoothed rule. Additionally each rule used in a successful parse can be added to the smoothing lattice again. It will not add any new rules since they were already explicitly entered in the grammar but the probabilities will be updated to reflect the visitation of the rule. In these two ways the grammar can continue to grow as the program experiences new images. The layer would need to have "accept" added to the protocol to support this kind of dynamic language learning.

By inspecting the regions of the parse result we can determine how much the parser believes the label assignment as well as what rule was responsible for the assignment. If the average amount of smoothing necessary to parse an image increases, those parts of the image responsible (locally) for the (globally) poor parse probability can be examined

and used to suggest adjustments to the evidence generation and segmentation framework that may improve the parse performance.

We have demonstrated two parse algorithms on synthetic data. In chapter 6 we develop the mechanism necessary to extract optical models and contexts from the corpus which allows us finally to segment and parse real images.

Chapter 6

Interpreting & Clustering the Corpora

6.1 Introduction

In the preceding chapters we have seen several interpretation problems solved with GRAVA implementations. A GRAVA program solves interpretation problems by having agents *fit* models to the data and by using an MDL criterion to choose between competing models.

The statistical models of language described in Chapter 5 depend upon a corpus to provide the language grammar, the frequency of use of the individual grammar rules, and other frequency information such as word frequency and frequency of part-of-speech sequences. Our system has similar needs, from collecting outline sequences for the base segmenter, to inducing picture grammars for the parser.

The requirements of the parser described in Chapter 5 can be divided into two main categories: optical interpretation models that label the region based upon the region's shape and contents; and picture language models.

The goal of the system described in this thesis is to imitate human performance in the task of segmenting, parsing, and labeling aerial images. More specifically, the goal of interpreting the corpus described in this chapter is, as with the other interpretation problems described in this thesis, to build a structural description suitable for the consumer of the description. In this case the consumer of the description is the program synthesis system described in Chapter 7. The program synthesis system constructs GRAVA programs, which consist of agents and models, from the description (which we call the specifications). The key idea is that if the specifications accurately describe the behavior of the human expert, and the program is generated from the specifications, then the resulting behavior of the system should imitate the human expert.

Corpus interpretation provides a key link between the MDL agent part of the GRAVA architecture described in Chapter 3, and the self-adaptive extension to it described in Chapter 7, as the models that are induced serve both pieces of mechanism.

6.1.1 The Idea of Contexts

The GRAVA architecture supports two basic mechanisms aimed at producing descriptions that mimic those produced by an expert. These two mechanisms are depicted schematically in Figure 1.4 which is reproduced below for the reader's convenience.

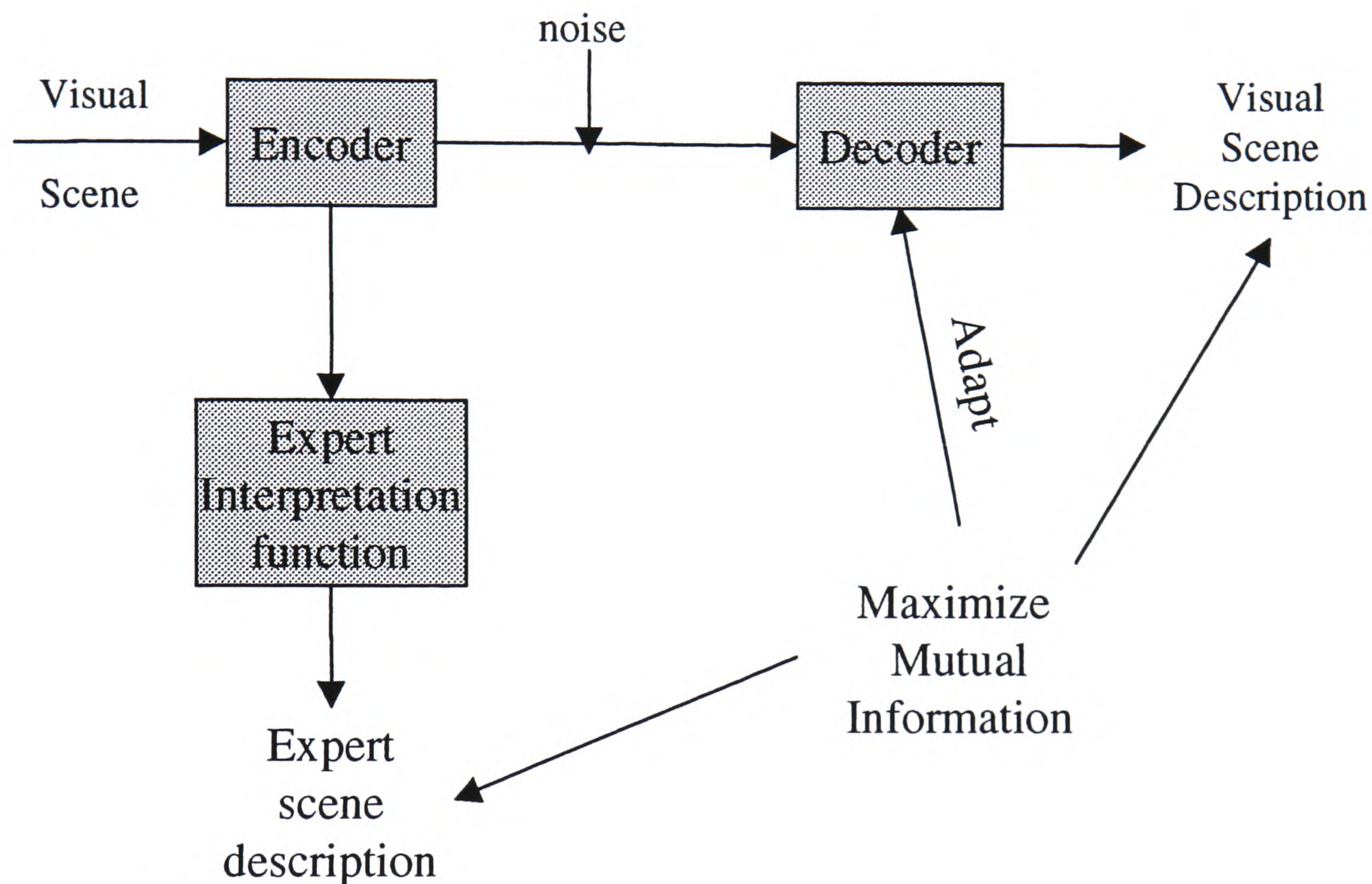


Figure 6.1: Communication Model

Chapters 3, 4, and 5 deal with the MDL agent approach to building good descriptions. Given a set of models, agents that can fit them, access to appropriate approximations for P, and a core interpreter that uses "Monte-Carlo-Select" as the means for choosing between competing agents, it is possible to approximate a global MDL description of the image. That aspect of the architecture was denoted in Figure 1.4 as "Decoder". The second mechanism (which is the topic of Chapter 7) involves self-adaptation of the "Decoder" to maximize the similarity of the produced descriptions with those produced by the human expert that produced the annotated corpora.

A conceptual view of the self-adaptive approach is given in Figure 2.2 which is reproduced below (as Figure 6.2) for convenience. The idea is to adapt the program to a particular "context". In order to achieve this adaptation we build a structural description from the corpus that facilitates dividing the model space into contexts and provide a mechanism for determining when a context is a good fit to the environment of the program.

In speech understanding systems it is common to have separate grammars for specific contexts. For example, when a speech understanding system is waiting for a phone

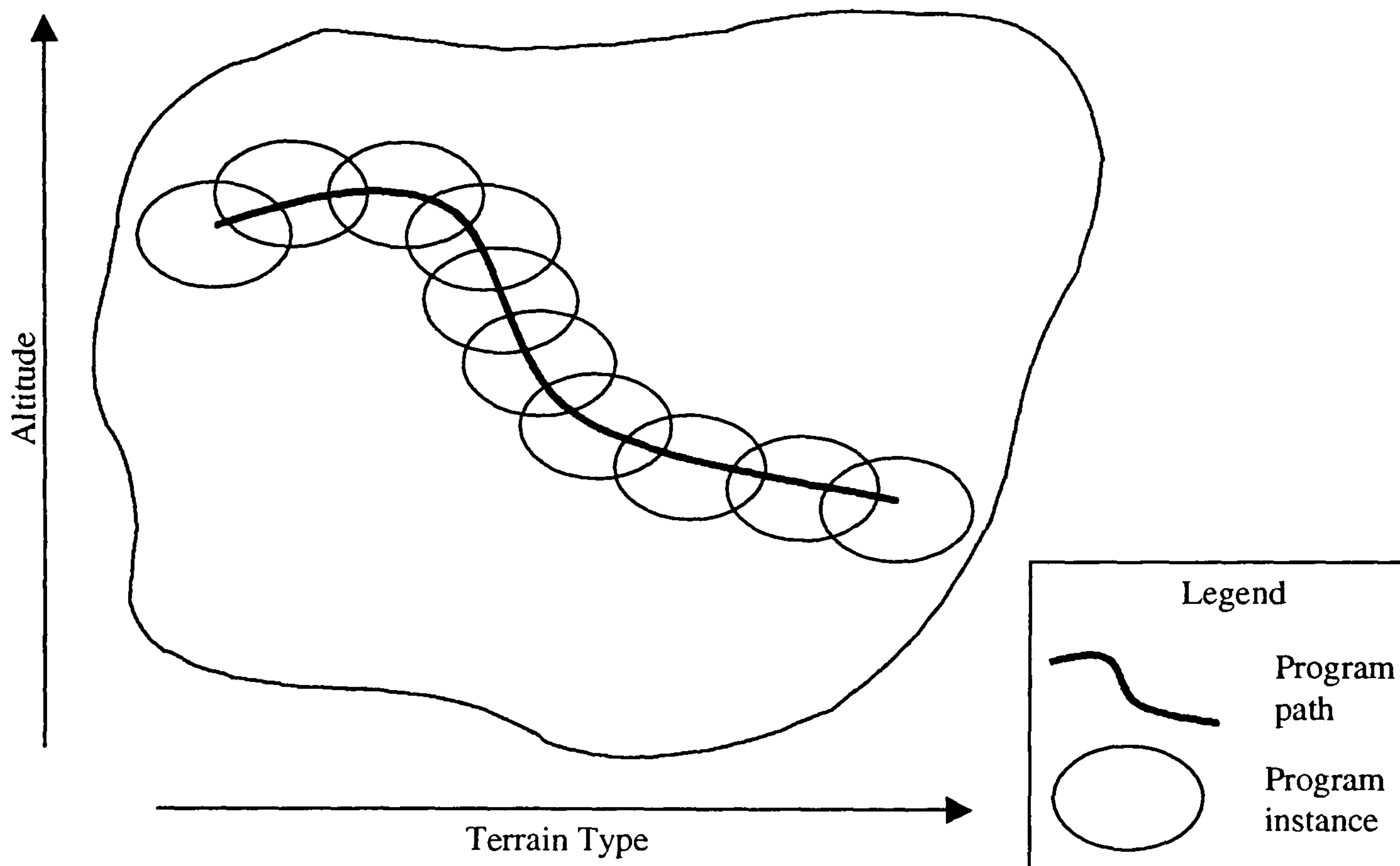


Figure 6.2: Path of a Self Adaptive Program

number, the probability of a word being one of the digits is much greater than it is in other contexts. The main reason for this practice in natural language is that when the vocabulary and the set of parse rules become too large, the HMM methods, that are frequently used in speech processing, become unwieldy. It is also useful, and perhaps necessary, to have such contexts in order to provide enough constraint to make sense of what is often a very noisy signal.

We also note from studies of human perception that we always interpret images within a context that defines our prior expectations about what we expect to see. Psychologists call this "priming".This reaches an extreme form in the case of model-based image analysis, in which programs "hallucinate" (Clowes 1971) one of a small set of models onto images. Typically, the human programmer defines the "context" by providing *a-priori* the (small) set of models that can be hallucinated.

The need for contexts to manage the diversity of the world is no less important for image understanding. AI has long understood the importance of contexts. In 1975 Minsky introduced the notion of *frames* (Minsky 1975) which was essentially an approach to contexts. Frames have been used extensively in AI research, especially for natural language. Riseman's Schemas (Draper *et al.* 1988) was a similar idea specifically for

Computer Vision.

Different image types are comprised of different kinds of regions, different colors and textures, and different parse rules. Rather than making one huge grammar that includes all textures and region types, it is better to have grammars, and optical models tailored to the context because tailored contexts provide greater accuracy and constraint.

A premise of the self-adaptive approach is that it should be possible, at runtime, to synthesize context specific systems, to determine the need to change context and to self-adapt the program so that the correct context is applied to the image that is being interpreted.

Contexts occur for a variety of reasons, at different levels of processing, and in different parts of the corpus. Given a set of images it is generally not possible to divide the images into separate piles with each pile representing a different context. Contexts for different aspects of the problem can be composed in a variety of ways. The explosion of possible combinations of contexts is one reason why the self-adaptive approach is attractive. That is, rather than generating all possible combinations of contexts in advance—and then having a "big switch" to choose which to use—it is better to generate the particular combination of contexts on demand.

To better understand the idea of contexts, consider the case of optical model contexts and language model contexts.

Figure 6.3 shows four multi-spectral color SPOT images from the color corpus[1] that demonstrate different contexts. Images (1) and (2) are similar in content (mostly farmland and small towns) but the colors and textures of the regions are very different. In fact, the images are taken under different imaging conditions. In the case of these two images, the major difference is with the optical models, since, grammatically, the two are rather similar. In images (3) and (4) the nature of the terrain is very different. Image (3) shows part of a major city whereas image (4) shows a rural setting with only small villages. The grammar that is suitable for parsing images 3 and 4 is quite different. Attempting to interpret any of these images with the wrong collection of optical or grammatical models may be expected produce a poor result especially since, as we explained in Chapters 4 and 5, knowledge weak segmentation algorithms often give poor results. In this case, the reason for the differences between image (1) and image (2) were changes in the SPOT[2] technology used to image them.

Separating contexts in this way suggests (for example) that grammar learned using one sensor implementation may be usable even when the sensor technology is changed, so long as new optical models are available for the new sensor. One of our original goals was to be able to build vision systems that continue to work well despite changes in the

[1] The corpora are described in Appendix A

[2] The SPOT satellite imaging technology is described briefly in Appendix A.

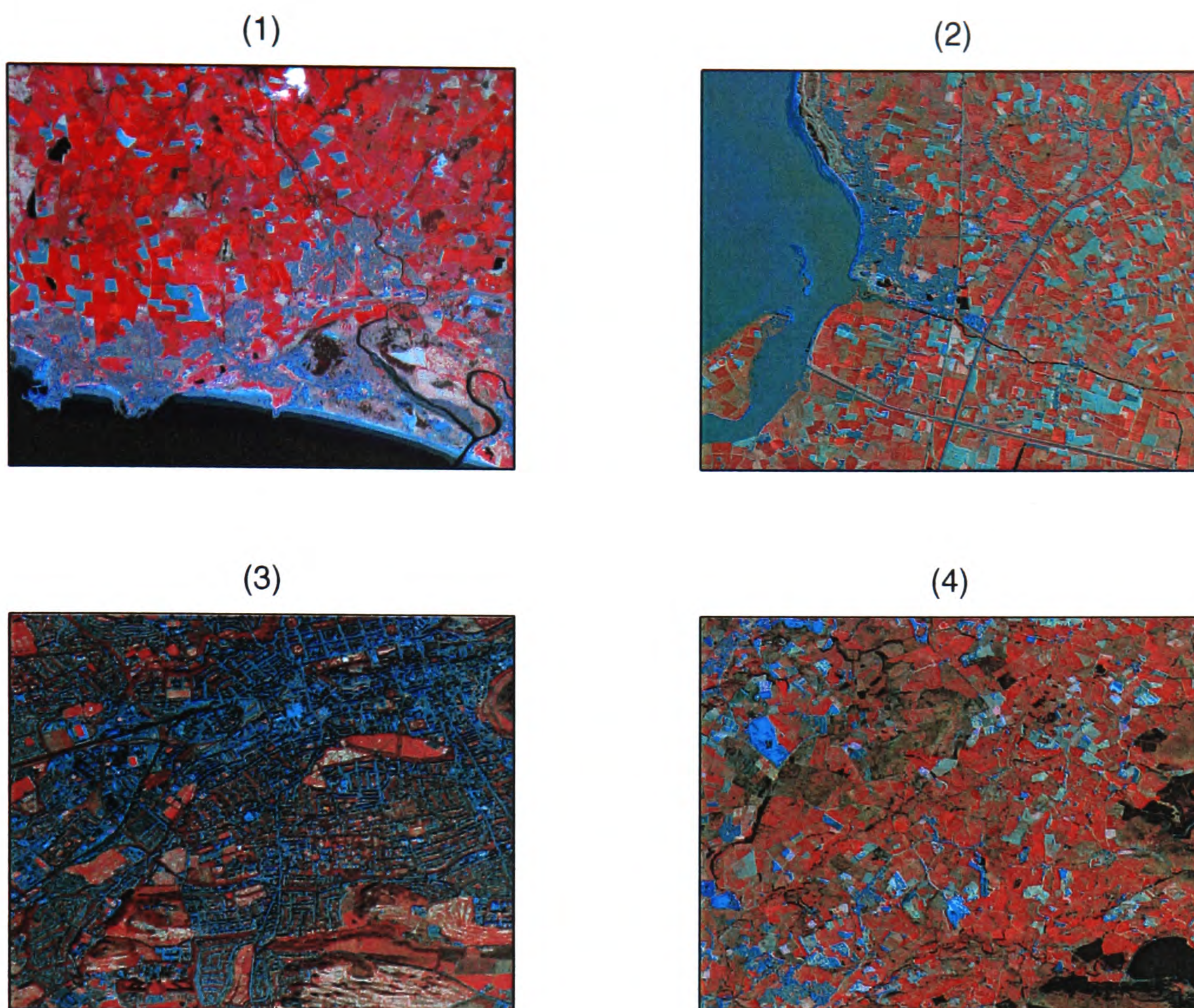


Figure 6.3: Image Contexts

sensor and image preprocessing environment. For that reason, the separation of imaging contexts from language contexts is an appealing idea.

Even without changing the sensor technology, different optical contexts may be called for, for example, variations in optical characteristics due to changes in weather conditions or season. When the seasons change, the optical characteristics of fields and trees vary dramatically but the language of the terrain changes little grammatically because it is defined by the man-made structures and natural constraints of the terrain. London is not a city one day and a mountainous region the next.

In the case of the picture grammar, we want to produce automatically batches of rules that constitute image grammar contexts. These batches of rules are models of the context in question. It is natural to view the task of constructing such batches of rules, within the GRAVA architecture, as interpreting the corpus as specifications of grammatical contexts. Similarly, in the case of optical models, we want to produce batches of optical models that can be grouped into contexts.

Most practical natural language systems have carefully constructed grammars for specific, carefully-designed contexts. However, the real world is so large and complex that if we wish to build image understanding systems that can operate reliably in a wide variety of situations it is necessary that we be able to learn contexts automatically from

the data rather than designing them manually.

In this chapter, we outline a description of the corpora suitable for serving the needs of the segmenter and parser, and which is suitable to serve as a specification for the program synthesizer described in Chapter 7. This chapter deals with what would usually be considered *training*. Our approach to the corpora, in line with the view taken throughout the thesis, is that the corpora is *interpreted*, that the MDL description is best for this purpose, and that the description is formed by fitting models to the data.

This chapter is organized as follows. Section 6.2 provides a discussion of related prior work. Section 6.3 describes a novel MDL clustering algorithm that supports the description of the corpus as models and provides a reasoned approach to measuring the comparative *fit* of a model. Section 6.4 develops the optical meta-models used for the color corpus. Section 6.5 develops the language meta-models.

6.2 Prior Work

In this thesis we attempt to mimic human performance in interpretation of aerial images. The idea of imitating human performance is not new. The basic idea underlies the work on statistical language learning already described in Chapter 5. The idea of imitation learning is also of interest to roboticists. Training a robot to perform actions can be a difficult and tedious task. A natural form of interaction with a robot is to teach the robot what to do by example. Approaches to training robots to perform actions can be roughly categorized into:

1. Learning by being explicitly programmed—programmed learning;
2. Learning by giving rewards and penalties—reinforcement learning; and
3. Learning by example—imitative learning.

The benefits in any of these tasks, whether it be training a robot motion controller or learning a language, can be summarized as follows:

1. Explicit programming is a complex task that is both expensive and error prone. It is certainly not a user level activity—and thus not suitable for many applications. It is also hard to get right. Manually producing the correct grammar for a language is an error prone activity. The approaches to language learning outlined in Chapter 5 have the advantage that the grammar matches the actual usage rather than someone's interpretation of what the grammar ought to be.

   For example, it has been found that specific sets of sentences forming a corpus such as the Wall street journal and children's books have very specific—and different—grammars.

2. Reinforcement learning is a time consuming activity. For certain scenarios it can be dangerous. A mobile robot, for example, might damage itself in the process of reinforcement learning before the skill is learned.

3. Imitative learning has the advantage that learning can be fast, can accurately reflect the real world, and can be safer than other methods. In imitative learning the learner "interprets" what it *sees* and then produces a plan to duplicate the behavior based on its own capabilities. Out of the three approaches, imitative learning is the only one in which the learning system, in some sense, *knows* what it is doing. In explicit programming, the programmer *knows* what is being programmed but the robot merely executes the program—without understanding. In reinforcement learning the learner knows that some things are good, and others bad, but has no idea why—other than experience from training—and has no notion of what it is doing. By contrast, imitative learning involves internalizing, through observation, a semantic model of behavior that the learner must mobilize into a plan of action. In the sense that the learner has such an internalized plan of action it *knows* what it is doing.

Imitative learning belongs to a general approach known as behavior-based methods. As with much work in A.I. the behavior-based approach draws upon human experience. Humans can and do learn by imitation and so it seems natural as a way of interacting with certain kinds of intelligent objects (such as robots). The fact that humans can achieve learning through imitation suggests that the approach is feasible.

Humans are among a small class of animals that learn by imitation (Galef 1998). Chimps and dolphins are other animals that depend upon this form of learning. Most animals do not seem to possess this form of learning just as most animals appear not to exhibit empathy and appear unable to recognize themselves in a mirror. This suggests that this form of learning is *high level* and a recent addition in evolutionary terms.

Imitative learning in robotics has allowed robots to dance the Macarena (Mataric 2000a), and learn the path through a maze by watching another robot navigate a maze (Hayes & Demiris 1994).

Mataric (Mataric 2000b) implements imitative learning for humanoid robots by matching movements of the demonstrator with robot controller primitives responsible for performing movement in the robot. The model of the robot's capabilities drive the attention of the visual routines to attend to parts of the example that correspond to parts of the robot controller capabilities. A sequence of actions on the part of the demonstrator are therefore mapped onto a sequence of robot controller commands. Such examples are learned by a learning module. The robot can then execute the learned sequences of robot controller commands in order to mimic the observed behavior.

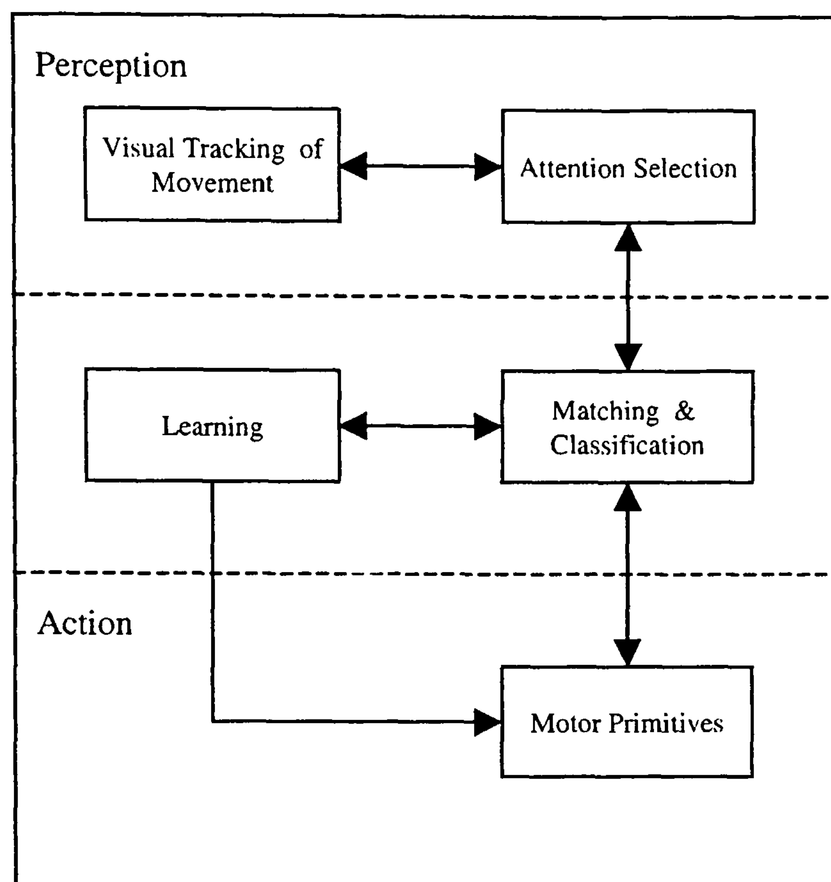


Figure 6.4: Maja Mataric's Imitative Model

Figure 6.4 shows Mataric's information processing flow model for learning movement from observation. The primitive motor models are used as contexts with which the observed movements can be *interpreted.* The observed movements are matched against the motor primitive so that the resulting learned behavior is learned in terms of the robot's capabilities. The robots capabilities may be very different from those of the human demonstrator but the same behavior is learned even though the underlying operations may be very different.

The approach taken in this thesis, while not involving robot motion, is similar in that:

1. it involves learning from example. The program attempts to mimic the performance of the expert photo-interpreter;

2. it learns by matching what the expert did—the annotations—with its set of primitive operations. The primitive operations in this case are the various agents;

3. the agent's capabilities determine what can be learned in the same way that the primitive operations determine what motion sequences can be learned for a robot controller.

GRAVA has a reasoned approach to how the matching of agents to observed phenomena occurs. This is by trying to *fit* models to the data. We also have a reasoned approach for preferring one agent over another—MDL.

We treat learning and performance of skills as a two-stage process. Skill in the context of our experimental program refers to the ability to segment, parse and label an image.

The first stage involves *interpreting* the data in the examples (corpora) as a specification for a behavior. The first stage is the topic of this chapter. The second stage involves synthesizing a program that can perform the task in what is believed to be the environment. That is the topic of Chapter 7. Our approach extends a reasoned MDL approach to interpretation problems to interpreting example data. The separation of corpus interpretation (this chapter) from program synthesis (Chapter 7) allows changes in the environment to be accommodated at runtime by re-synthesis of the program.

The purpose of providing a corpus of positive examples is to *suggest* plausible values that may be encountered during the runtime of the program. If the corpus is *representative* of what is encountered at runtime the program should be able to reason about the data that it encounters as being within the statistical neighborhood described by the examples in the corpus. Interpreting the corpus, therefore, is mostly a problem of modeling groups of data points as *predictive* models of the world. By predictive we mean that the models should allow the probability of an event to be calculated by virtue of a statistical model learned from the corpus.

To do this we need a mechanism for grouping collections of points into clusters that will individually be represented as statistical models and which themselves can be grouped into contexts.

Clustering is known to be a hard problem, especially unsupervised clustering. K-means (Moody & Darken 1989) is good if K is known. Even when K is known K-means performs poorly with multiple-population problems. In our application, K isn't known *a-priori*, and can't be. The Kohonen self-organizing feature map (Kohonen 1988) has problems relating to its topographical property that make it unsuitable especially for high dimensional spaces such as those encountered with our application or interpreting corpora.

Given the coding/MDL perspective of interpretation in this thesis, we have been led to develop what appears to be a novel clustering algorithm.

6.3 Principal Component Decomposition

A corpus provides multiple positive examples of a structure that we wish to model. The structures in question have one or more dimensions, and the corpus provides examples of the structure that enable us to model the location within the appropriate multidimensional space. One way of doing this is to model the structures as a PDF. The natures of the structures may be very different. In this chapter, we develop structures that correspond to region labeling and picture grammar. While the structures may be very

different the essential nature of a corpus is the same: positive examples of structures in a multi-dimensional space.

Chapter 3 introduced an example of a principal component analysis model (PCA). PCA is similar to the goal of corpus interpretation in that it involves building a model from a collection of multidimensional positive examples of the modeled entity. Briefly, the approach is as follows: a mean structure is produced from the training set. This produces a point in the multidimensional space. The instances of the modeled object (such as a face) fall near to the mean point. Some dimensions vary a lot and must be parameterized in the model while others vary very little and can be considered constant in the model. The key idea of principal component analysis is to find the few principal components and to ignore the rest thereby producing a model that has far fewer dimensions than the space in question.

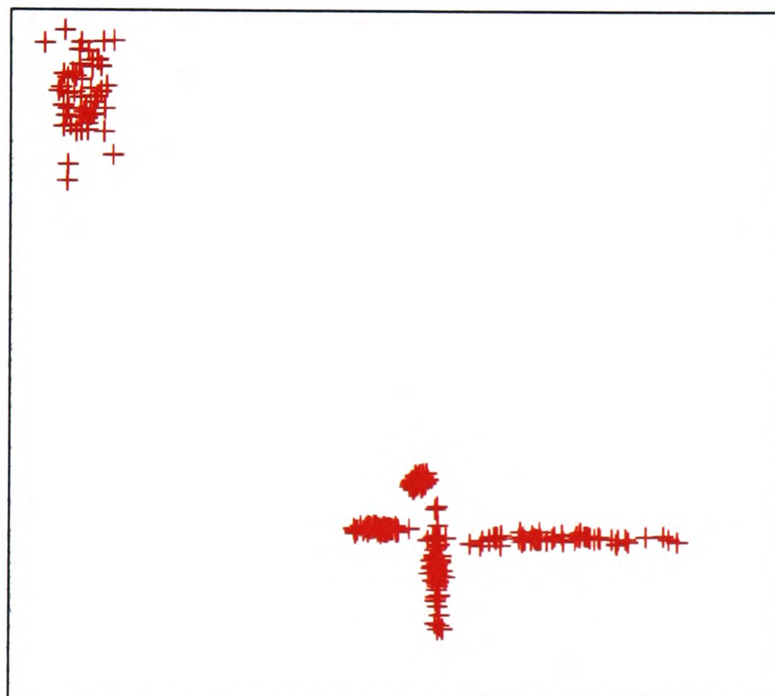


Figure 6.5: The Need for Decomposition

Consider two dimensional[3] space and the collection of positive examples shown in Figure 6.5. Given this set of data points it is clearly possible to find a mean point and the principal and secondary eigenvectors. Unfortunately the resulting model is unusably crude since most of the points that it generates are not suggested by the data. The problem gets successively worse for higher dimensional spaces. Data like this could result, if, for example, data was collected for a PCA face model but instead of limiting the data collection to humans it included dogs, cats, monkeys, and elephants as well. The points that interpolate human and elephant in such a data set are of no interest to us since such creatures do not exist in the world. Of more relevance to our program is that the data could result from aerial images taken during different seasons, or images

[3]We provide two dimensional examples in this chapter because they can be graphically illustrated on a two dimensional medium. In practice the number of dimensions is far larger than two.

obtained using different sensors. The data in this example clearly suggests five different models—not one with statistical variations.

Principal component decomposition is the interpretation of a set of data points, such as the example in Figure 6.5, into the component collections (five in this example) by analyzing the principal components of the interpretation space. After the space has been divided into separate clusters conventional PCA may be applied to produce the models.

The algorithm builds upon two earlier works. The first is a classification program developed by Wallace. Wallace's (Wallace 1990) program (SNOB) worked by finding a minimum message length (MML) description of a set of points. The second is the practice of using principal component analysis (Jackson 1991) to reduce the dimensionality of high dimensional problems so that the separate populations can be modeled. Our algorithm applies principal component analysis recursively in order to separate the collection into successively smaller clusters. At each point the criterion for separating a population is that it reduces the global description length of the original population.

Below we present a novel algorithm that produces such principal component decompositions.

A Statistical Model for Clusters

Given a set D of m n-dimensional points. we can interpret the points in this space as being:

1. Unrelated points.
2. All members of a single model.
3. Grouped into a number of models.

A premise of the GRAVA architecture is that knowledge of the world in the form of models can be used to produce better descriptions of an image. A good description of the world in the GRAVA architecture is one that has a minimum description length. A model allows a shorter description length if the model reduces the amount of uncertainty about the values of features in the image. The best interpretation of the data points that constitute the corpus therefore is the interpretation that reduces the uncertainty about where the data points appear in the multidimensional space.

The entropy of the collection of data points in the corpus is given by:

$$H = -\sum_{i=1}^{m} P(D_i) log_2 P(D_i) \tag{6.1}$$

The lower bound MDL of a description that represents all of the points in D is given by:

$$DL = -\sum_{i+1}^{m} log_2 P(D_i) \tag{6.2}$$

In order to compute this theoretical description length, it is necessary to know the PDF for points in D. A corpus doesn't specify every possible point in the space. A corpus provides a collection of *representative* points in the space. The job of interpreting the corpus involves modeling the PDF. There are many choices for modeling a PDF. One model that is simple, predictive, and which often pertains to naturally occurring distributions is the Gaussian.

The description of a Gaussian model consists of a mean and variance of the distribution $< \bar{\mu}, \bar{\sigma}^2 >$. For a set of points the Gaussian model can be fitted simply by computing the (n-dimensional) mean $\bar{\mu}$ and the variance $\bar{\sigma}^2$. Given this characterization, for any point d we can compute the probability $P(d)$ as follows:

$$P(d) = \prod_{i=1}^{n} \left[erf\left(\frac{(pos(d_i) - \mu_i + \epsilon/2)}{\sigma_i} \right) - erf\left(\frac{(pos(d_i) - \mu_i - \epsilon/2)}{\sigma_i} \right) \right] \tag{6.3}$$

where $pos(d)$ is the position of the point d, ϵ is the position resolution, μ_n is the n-dimensional mean, σ_n^2 is the n-dimensional variance, and erf() is the error function.

The choice of whether to consider the points in the corpus as (1) unrelated individual points, (2) all members of the same model, or (3) divided into groups each of which is modeled, is to select the choice that yields the minimum description length.

The interpretation task can therefore be characterized as dividing the data points in D into k proper subsets $C_{i,k}$ such that:

$$D = \bigcup_{i=1}^{k} C_k \tag{6.4}$$

The MDL is

$$arg \min_{C_{1,n}} \sum_{i=1}^{n} \left\{ \left(\sum_{d \in C_i} -log_2 P(d|C_i) - log_2 P(d \in C_i) \right) + ddl(C_i) \right\} \tag{6.5}$$

where $ddl(C_i)$ is the description length of the distribution used to model C_i. The description of a point is divided into two parts. The first part identifies its position in the space $(-log_2 P(d|C_i))$ and the second part identifies to which collection it belongs $(-log_2 P(d \in C_i))$.

The statistical models chosen for C_i determine the size of the point descriptions. In order to specify the position of a point we choose a resolution ϵ to be used uniformly since otherwise a point can have an arbitrary precision and its representation would be arbitrarily large.

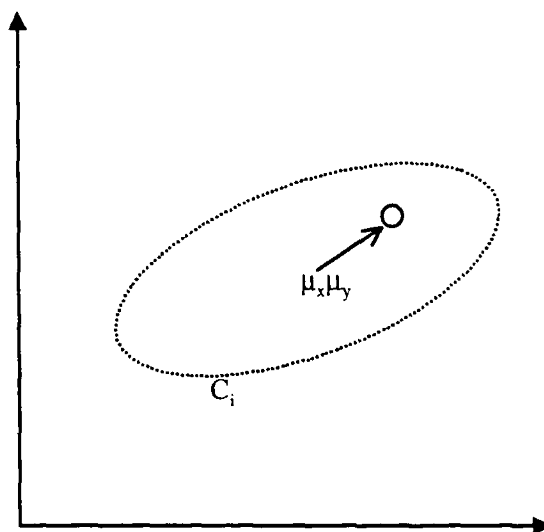


Figure 6.6: Representation of a Point in a Collection

If the representation of a collection includes its mean position μ, the positions of the points in the collection can be described as distances δ from the mean. Figure 6.6 shows the representation of a point within a collection C_i as an n-dimensional mean ($n = 2$ in this example) and a n-dimensional displacement. So any point d can be described as:

$$\bar{\mu} + \bar{\delta} - \frac{\epsilon}{2} \leq d \leq \bar{\mu} + \bar{\delta} + \frac{\epsilon}{2} \tag{6.6}$$

Given the original set of points it is possible to reconstruct the statistical model that was used to represent it. So to communicate the collections, all that is required is the mean position represented to an accuracy of ϵ. The points are represented as a description of which collection they belong to and the offset from the mean: $< C_i, \delta >$.

If all points are considered to be separate collections, each of a single point, the size of their offset will be 0 since all points reside at the collections mean. Since clusters and points have a one to one relationship, and the point description contains no information other than the collection assignment, the representation only requires the positions of the individual points in the collection. Collections that are represented as individual points in this way have no predictive value.

If all points are considered to be members of a single collection the representation of a point doesn't need to identify to which collection it belongs because there is only one collection.

As the data points in a corpus are divided up into smaller collections the description length of the individual points is reduced if the distribution that characterizes the collection is more predictive about the position of its component points than the distribution for the entire corpus was. Any suitable statistical distribution can be chosen for a collection.

Algorithm for Decomposition

Having defined the criteria for an optimal division of the data points into separate models we are left with the task of defining an effective procedure for achieving such a division. To accomplish this we developed an efficient algorithm that approximates a solution to Equation 6.5.

Our algorithm, which we call "principal component decomposition", attempts to divide the data by searching for dividing hyper planes along the eigenvectors of the data. The idea behind the algorithm is that the principal eigenvectors represent the dimensions with the greatest spread. The spread can be caused by a single phenomenon with a large variance, or it can be caused by more than one phenomenon distributed throughout the space. To distinguish these two cases we compute the entropy of the data points as a whole and then we compute the sum of the entropies of the two collections formed by dividing the data points into two collections with a hyper plane perpendicular to the eigenvector. We do this for all possible cut points along the eigenvector. If all sums of divided collections yield a higher description length than the original combined collection the collection is not divided, otherwise the collection is divided at the place that yields the minimum description length.

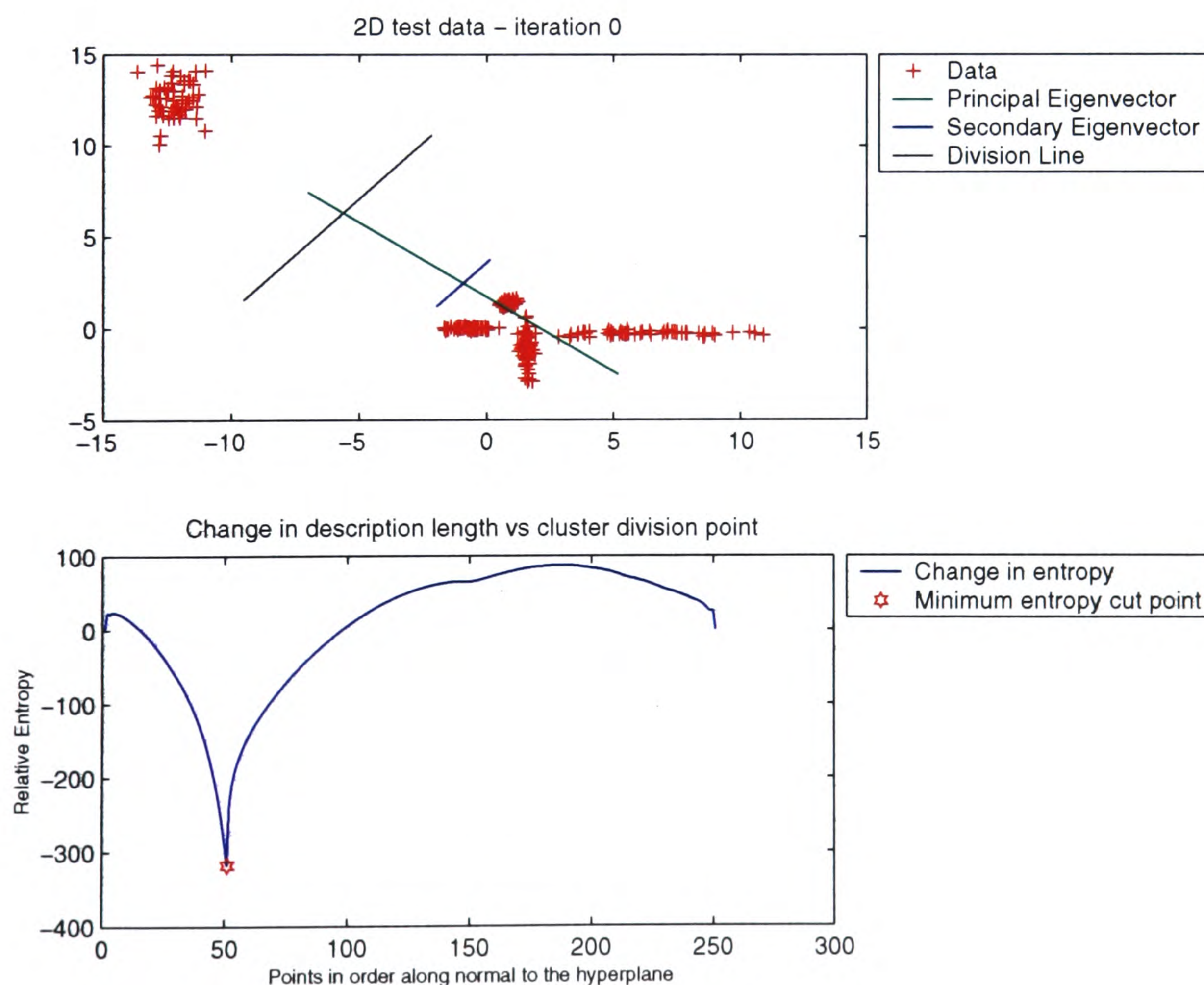


Figure 6.7: Dividing the Data Points to Reduce Description Length

Figure 6.7 shows the 2-dimensional data introduced earlier. There are two eigenvectors. The lengths of the principal and secondary lines are the square root of the

corresponding eigenvalues[4].

The hyper plane that is used for cutting the data is perpendicular to the principal eigenvector[5]. The graph below shows the change in the total description lengths resulting from cutting the collection at any point along the eigenvector.

When the change is greater than zero, cutting makes the description length larger. In this case the total description length is significantly reduced by cutting the collection at the point where the "division" line is drawn. This point can be seen as the minimum point in the entropy curve.

This procedure is repeated for each eigenvector of the collection starting from the eigenvector that corresponds to the largest eigenvalue until either a division occurs or until all eigenvectors have been tried. Once a collection has been split the algorithm is applied to each of the newly divided collections. Eventually there are no collections of points that split. The algorithm consists of two parts CHOP and MERGE.

CHOP looks for places to divide a collection of data points into two collections by finding a dividing hyperplane. CHOP thus produces two collections that have the property that if collection C_0 is divided into C_1 and C_2, $C_0 = \cup\{C_1 C_2\}$, and $DL(C_0) > DL(C_1)+DL(C_2)$. MERGE finds two collections of data points (say C_1 and C_2) that have the property that $DL(\cup\{C_1 C_2\}) < DL(C_1) + DL(C_2)$. If the collection of data points is non-convex CHOP can cause some points to become separated from the collection that they naturally belong to. MERGE re-associates points severed in this way with their natural collection. The advantage of this approach is that it is possible to construct non-convex collections of data points.

First we describe the algorithm for $CHOP(S)$ that chops the collection into separate collections.

$CHOP(S)$:

1. S is a set of n-dimensional data points. Let $\bar{m}$ be the mean and C be the co-variance matrix.

2. Let $v_1 \ldots v_n$ and $\lambda_1 \ldots \lambda_n$ be the eigenvectors and corresponding eigenvalues, respectively, sorted into decreasing order of eigenvalue.

3. For each eigenvector v_i starting with v_1 (the one with the largest eigenvalue—the principal eigenvector), search for the best place to cut the data points into two collections as follows:

   (a) Establish the cutting hyper plane. The cutting hyper plane is the plane that

[4] The eigenvectors are computed from a co-variance matrix so the eigenvalues are variances and the square root of the eigenvalues are standard deviations.

[5] A 2-dimensional hyper plane is a line.

is perpendicular to the eigenvector v_i. We arbitrarily choose the hyper plane that passes through the mean $\bar{m}$.

$$\begin{aligned} n &= \bar{m}^T v_i \\ \bar{r} v_i &= n \end{aligned} \tag{6.7}$$

where $\bar{r}$ is a point specified as a row matrix.

This is the perpendicular form of the equation of a hyper plane. This representation is convenient because it permits fast calculation of the distance of a point from the hyper plane. For any point d the distance from the plane in equation 6.7 is given by $n - dv_i$.

(b) Sort the points in S in order of distance from the cutting hyper plane. Since the hyper plane cuts through the mean, approximately half of the points will be on one side of the hyper plane, with the rest of the other side. Approximately half of the points, therefore, will have a negative distance from the plane. The distance is not the absolute distance from the plane, it is how far to move along the normal to the hyperplane to reach the plane in the direction of v_i.

(c) Let A be the sorted list of data points.

(d) Let B be an empty list.

(e) Let the $cutPoint = 0$ and $position = 0$

(f) Let $minDL = DL(A)$ the description length of the entire set of data points.

(g) Now we simulate sliding the cutting plane along the eigenvector from one end of the set of data points to the other, by taking points one at a time from A, putting them into B, and computing the description length of the two collections as follows:

For each point d_j in A do:

i. Remove d_j from A.

ii. Add d_j to B.

iii. Increment the position ($position = position + 1$).

iv. Compute the new description length as $newDL = DL(A) + DL(B)$.

v. If $newDL < minDL$ set $minDL = newDL$ and $cutPoint = position$.

(h) If $cutPoint > 0$ divide the data points S into two collections S_1, and S_2 at the position indicated by $cutPoint$. Then recursively apply $CHOP$ to both of the sub-collections to see if further chopping can be performed. Finally return the complete list of chopped collections:

$return\ append(CHOP(S_1), CHOP(S_2))$

4. At this point, all of the eigenvectors of S have been searched for chop points, and none have been found. The data points cannot be represented with a smaller description length by chopping along an eigenvector so return the list of collections as the single collection S:

   $return\ list(S)$

The nature of the way the collections are divided up results in some groups of data points being divided unnecessarily. This can be seen in the second and third iteration for this example data.

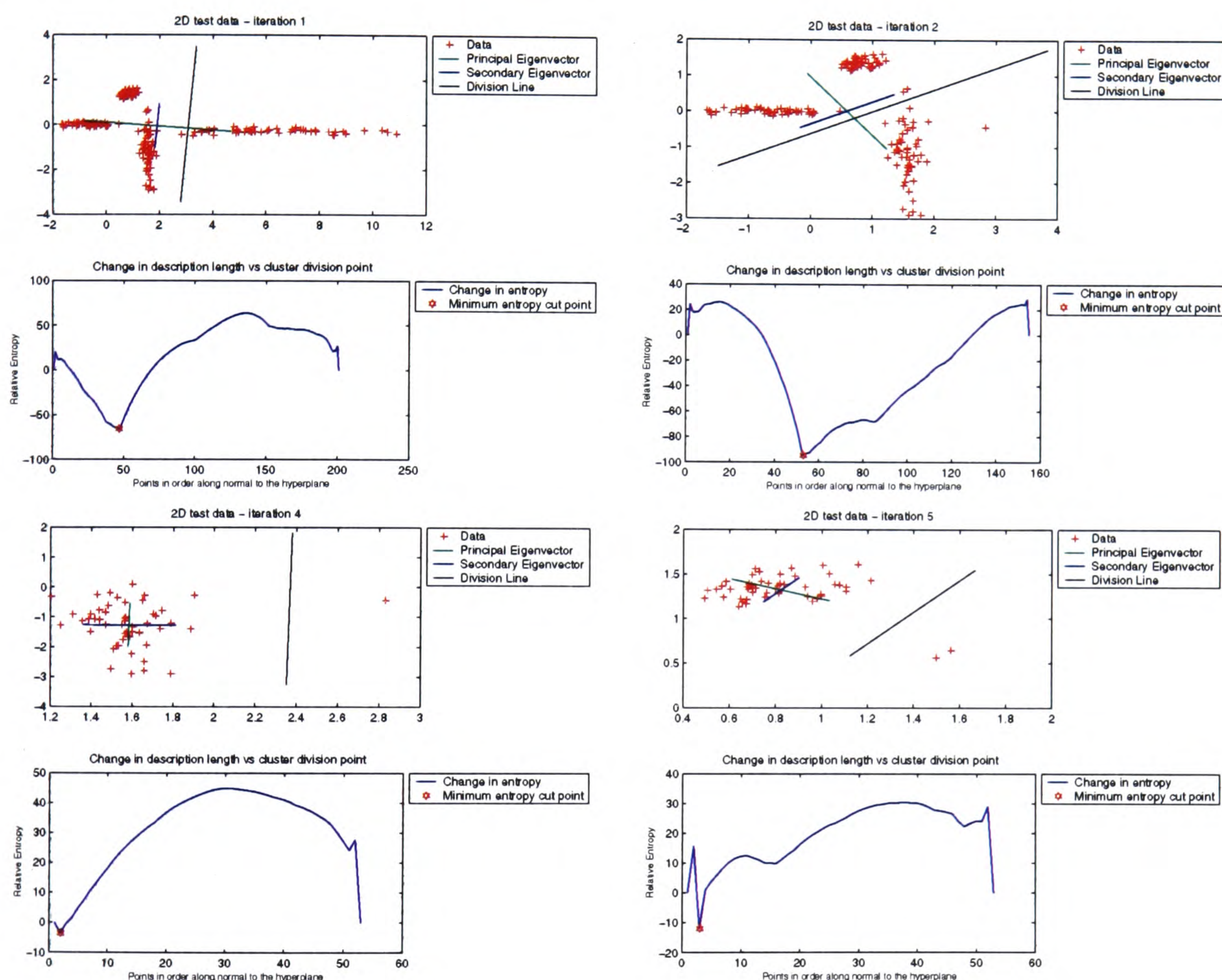


Figure 6.8: Accidentally Severed Points

In the second iteration (Figure 6.8 top left) one point is chopped off the right most collection. In the third iteration (Figure 6.8 top right) two points are chopped off. Later, in the fifth and sixth iteration, the one point on the left, and the two points on the right, are completely severed so as to be small disembodied collections of points (one point and two points respectively in this example). These accidentally severed fragments are corrected in phase two of the algorithm—the merge phase.

In phase two of the algorithm pairs of collections of points are checked to see if the description length would be reduced if they were merged. If the description length would be reduced by merging them they are merged. This is repeated until no more useful merges can be found.

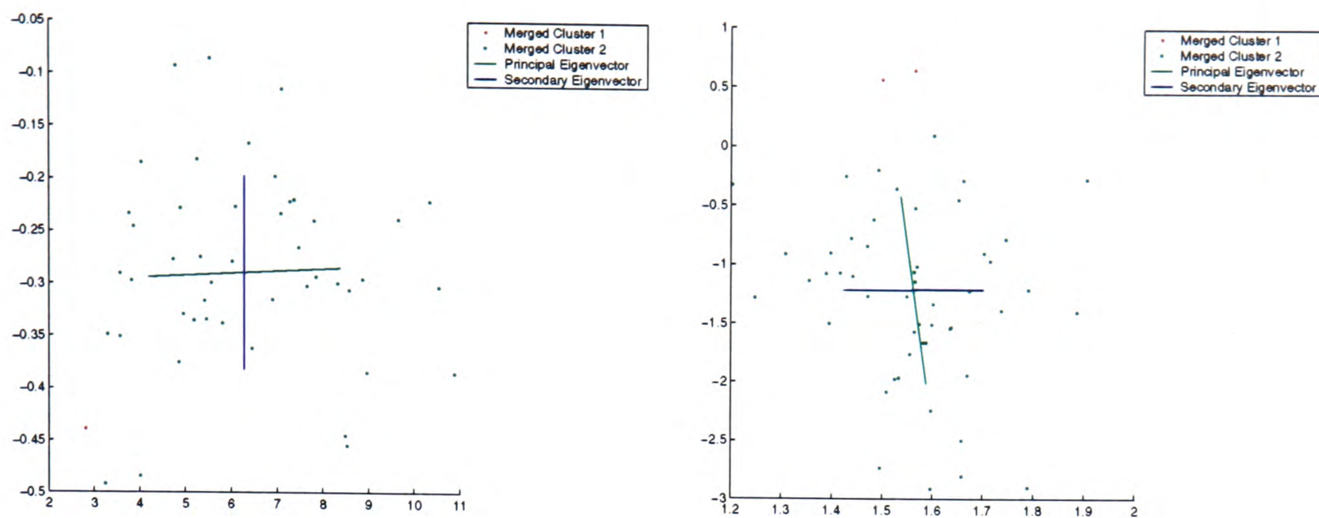


Figure 6.9: Merging Collections

Figure 6.9 shows the merge iterations that repair the severed points described above. In these two cases merging reduced the description length. After 16 iterations the division of the data points in to separate collections is complete.

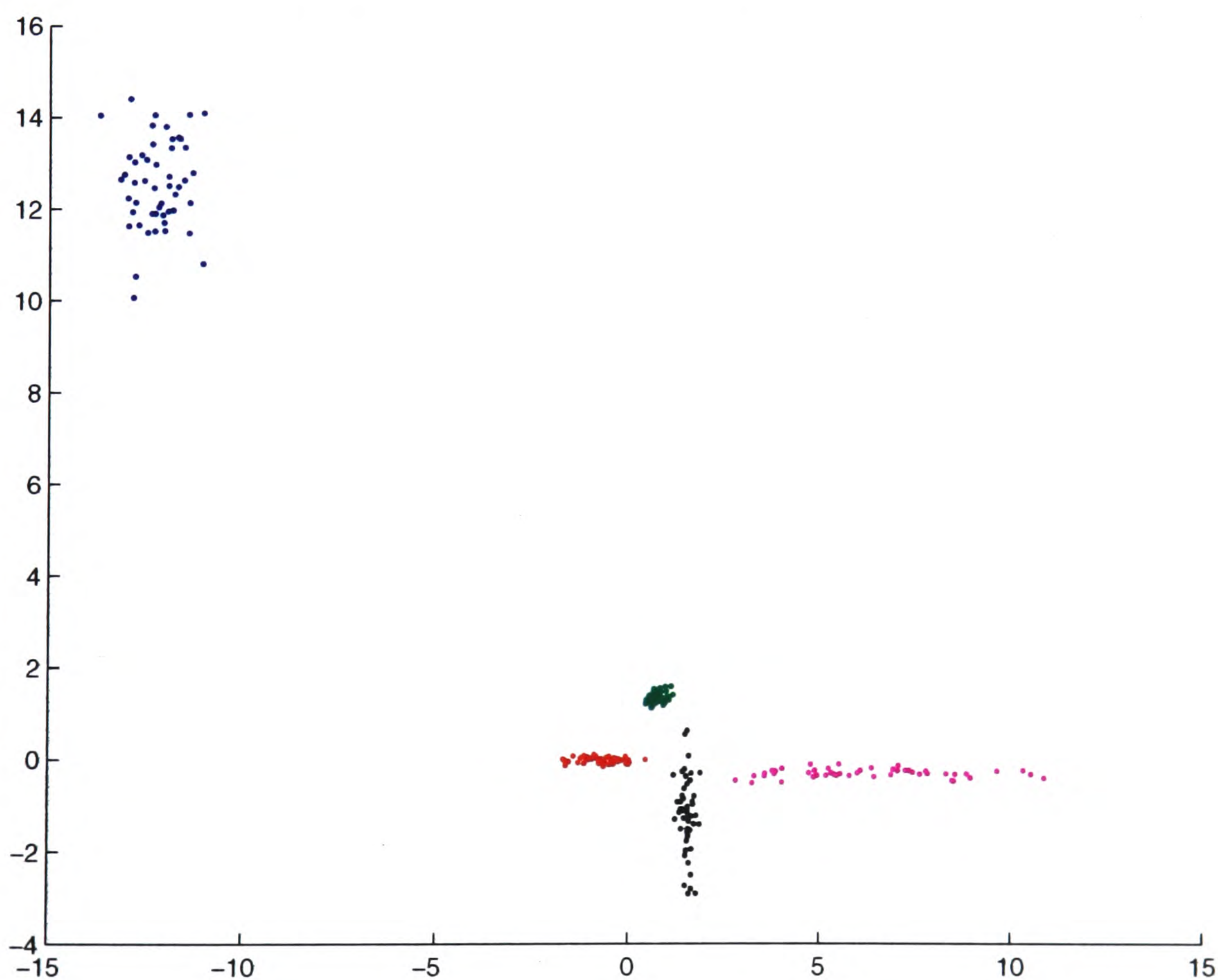


Figure 6.10: Example Result of Principal Component Decomposition

Figure 6.10 shows the final result of decomposition.

The PCD algorithm described above has a number of interesting characteristics:

1. PCD produces a structural description of the data points that is an approximation to a global MDL description of the points.

2. Each remaining collection of points can be represented efficiently by the statistical

model chosen for it since if the collection could not be represented well it would have been divided.

3. Each collection is a good candidate for PCA modeling because of (2). If the data points were for faces of dogs or humans, as discussed earlier, we may end up with a good PCA model for human faces and a good PCA model for dog faces rather than one general model for faces.

4. The algorithm can be implemented efficiently and can produce good decompositions very quickly. The number of "chop" and "merge" operations that are performed in producing a decomposition is very small compared to the number of points.

5. The algorithm can produce non-convex collections.

The final point (5) is an interesting feature of the algorithm that is not obvious from the example given above. Non-convex collections cannot be disentangled by using the "chop" operation alone but inclusion of the "merge" operation allows two convex collections to be joined so as to produce a non-convex merged collection.

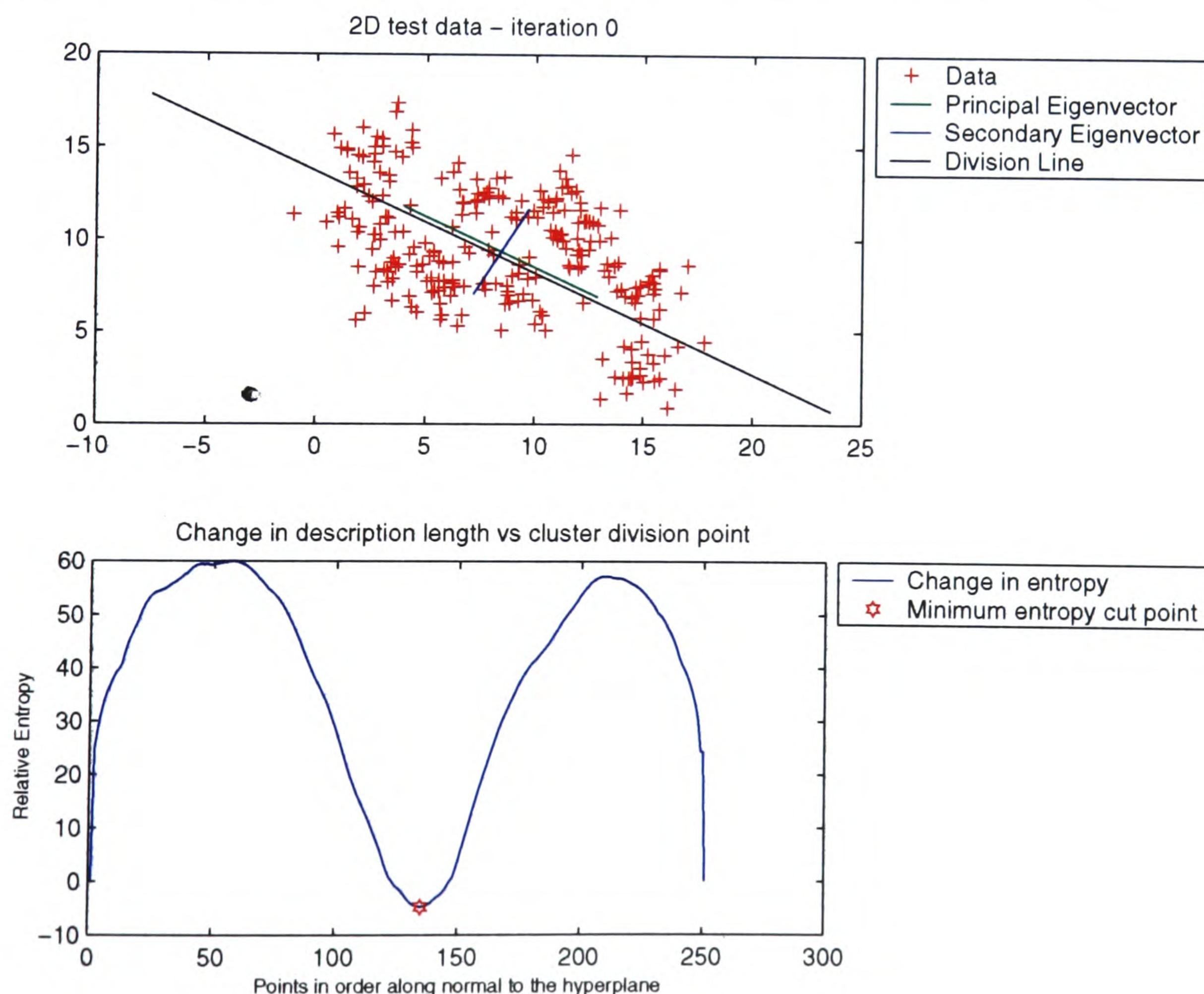


Figure 6.11: Intertwined Non-Convex Shapes

To demonstrate this capability we generated a set of data points by picking points randomly along two interlocking 'C' shapes in a ying yang configuration. Even though

the data is quite dense and intertwined the algorithm manages to "chop" it apart and then "merge" the severed parts back together. Figure 6.11 shows the first iteration on the data.

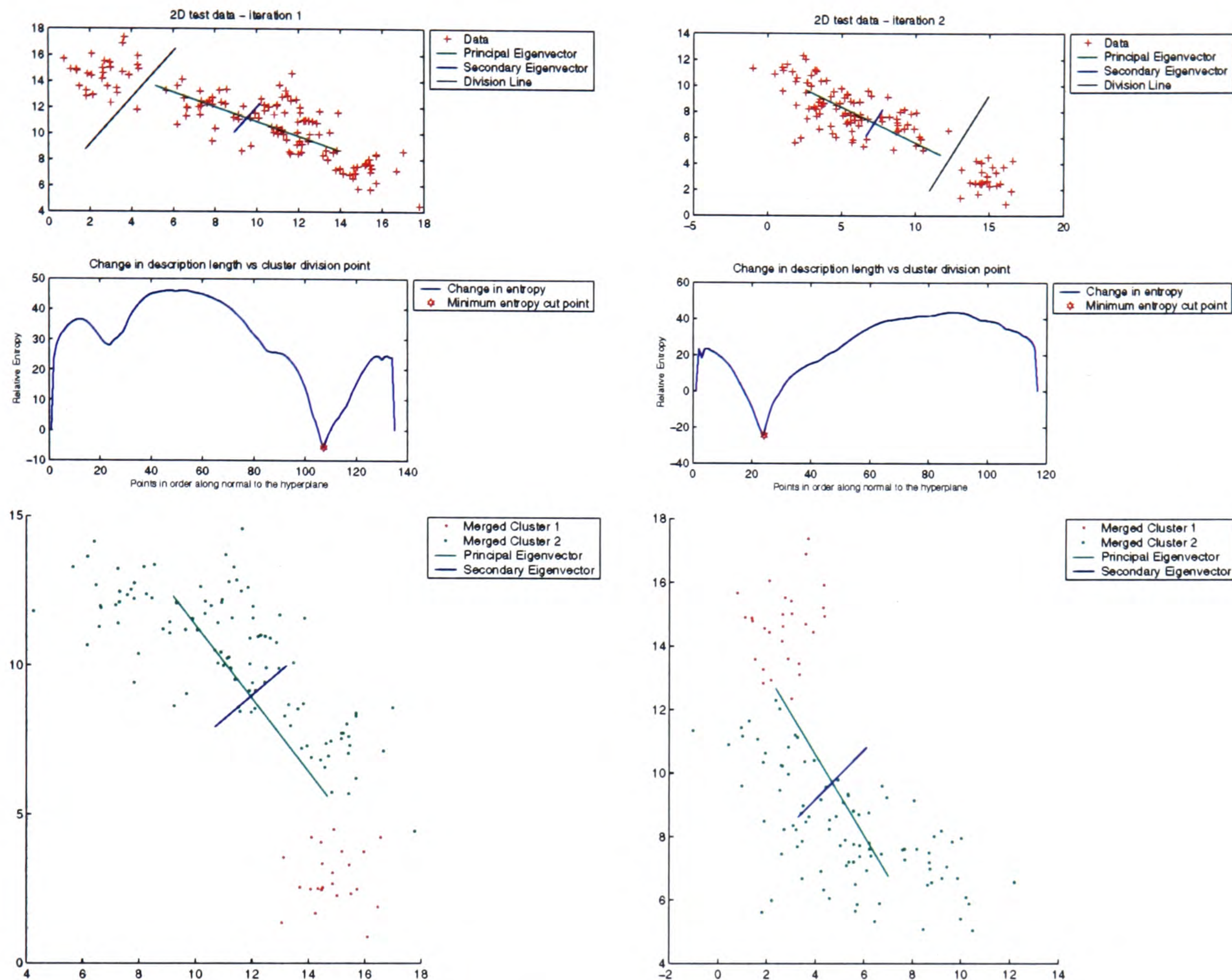


Figure 6.12: Curved example data

The second and third iterations chop the data down further and later iterations merge the severed portions back into their rightful places as shown in Figure 6.12.

The final decomposition of the data is shown in Figure 6.13.

This appears to be a novel algorithm for finding "components" of data in multidimensional space that is different from Kohonen maps (Kohonen 1988). The exact relation will be a subject of future research.

6.4 Interpreting an Image Corpus as Optical Models

The purpose of optical models, as described in Chapter 5, is to provide a way of estimating $P(l_i|r_j)$. The human expert provides for each region r_i, of the corpus, a label l_i.

Regions are generated by the segmenter at runtime. Therefore, the information available to the optical model is the description of the region produced by the segmenter, which may include whatever representation the segmenter provided but will at least include:

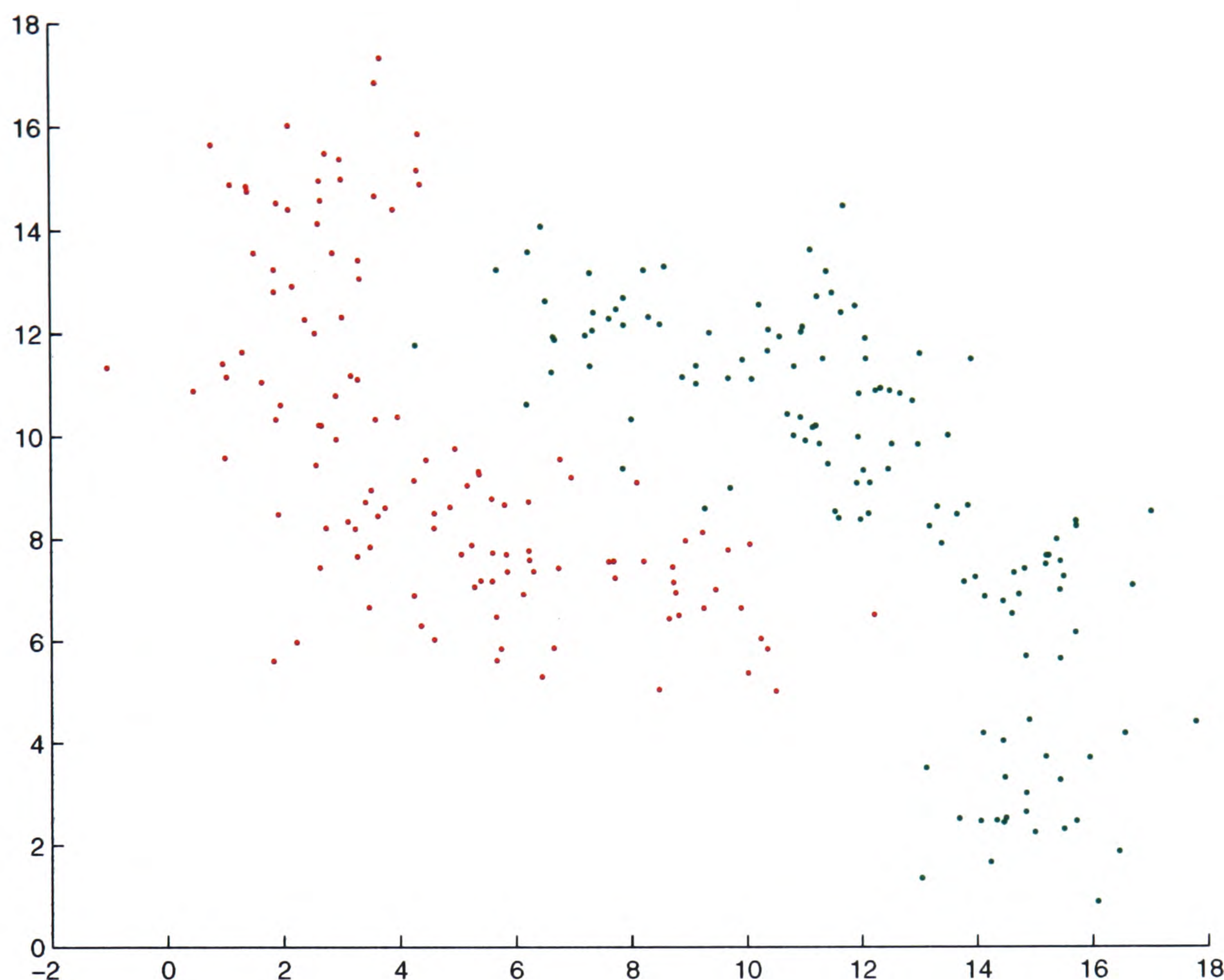


Figure 6.13: Final Decomposition of Example Non-Convex Data

1. The region size—the number of pixels in the region.
2. The region boundary or shape model.
3. The region contents.

Interpreting the optical components of the corpora involves building a description of the image region shape and contents. Shape is clearly important in the domain of aerial image understanding because certain things are best disambiguated by shape. For example river, lake, and sea, are easily distinguished by size and shape even though the pixels they contain may be similar. Nevertheless we focused on region size and contents in our optical models (Due to time constraints).

A corpus consists of a number of annotated images. As we indicated above in reference to Figure 6.3, the images in the corpus may be formed in a variety of imaging situation, for example due to changes in things like sensor, weather, and season.

Figure 6.14 shows the interpretation of corpus data for the label "river". We limited the dimensionality of the space to the red and blue channels so that the results could be easily shown in two dimensions. The data shows examples of river that have a blue component of between 0 and 160 while the red component has a range between 0 and 120. The red component, however, as discovered by the principal component decomposition algorithm, is best described as two separate models.

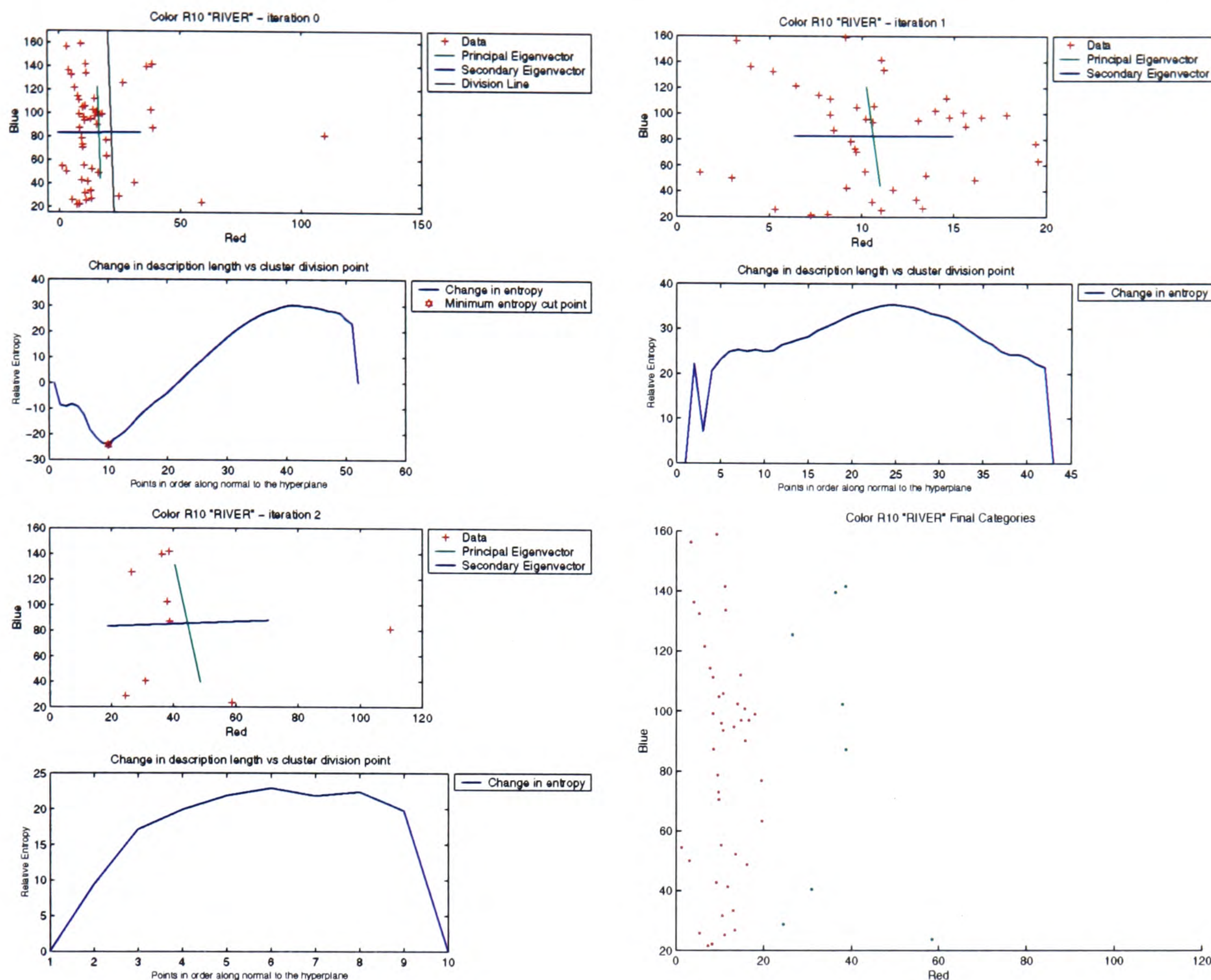


Figure 6.14: Description for River Using Red and Blue

Once the data for each label has been decomposed into separate models we are left with a number of models for each label type. Now given an unlabeled region from a segmentation we wish to build hypotheses about the correct labeling of the region.

A model $model_i$ produces a description length for any data point that is provided to it. That is, for any data point d, it produces a description length equal to $-log_2 P(d|model_i)$, so we have access to the quantity $P(d|model_i)$, but what we want is $P(model_i|d)$.

Applying Bayes' theorem we get:

$$P(d|model_i) = \frac{P(d)P(model_i|d)}{P(model_i)} \tag{6.8}$$

so

$$P(model_i|d) = \frac{P(d|model_i)P(model_i)}{P(d)} \tag{6.9}$$

$P(d|model_i)$ is provided by the models and $P(model_i)$ is easily estimated from the corpus by counting the frequency of occurrences of the models in the corpus.

$P(d)$ depends upon the probabilistic landscape that is contributed to by all of the models derived from the corpus. If all label types are accounted for in the corpus, and if the corpus is representative of the images being interpreted, d must belong to one of the label/model pairs and so:

$$P(d) = \sum_{j=1}^{n} P(d|model_j)P(model_j) \quad (6.10)$$

and therefore:

$$P(model_i|d) = \frac{P(d|model_i)P(model_i)}{\sum_{j=1}^{n} P(d|model_j)P(model_j)} \quad (6.11)$$

The models were all derived from data points labeled the same way so identifying a model uniquely identifies a label. We can therefore sum all model contributions that are for the same label to produce the probability of a label given the region $P(label_k|d)$.

$$P(label_k|d) = \sum_{i=1}^{n} \begin{cases} P(model_i|d) & \text{if the label of } model_i \text{ is } label_k, \\ 0 & \text{otherwise.} \end{cases} \quad (6.12)$$

Once a match has been made which has led to a global solution the point can be added to the corpus as a "learned" point thereby enriching the corpus. From time to time the corpus can be re-categorized in order to produce better models[6].

6.5 Learning Contexts from Images

In a corpus, whether it is of sentences or images, certain groups of entries may form interesting sub contexts of the entire corpus. The Brown corpus (Francis & Kucera 1982), a popular corpus for natural language research, consists of documents in the following areas:

1. Press, reportage (political, sports, financial, etc.).
2. Press, editorial (including letters to the editor).
3. Press, reviews.
4. Religion (books, periodicals, tracts).
5. Skills and hobbies (books, periodicals).
6. Popular lore (books, periodicals).
7. Belles Lettres, biography, memoirs, etc.
8. Misc. (government, industry, college documents).
9. Learned (natural science, humanities, political science).

[6]Just as in Chapter 5, we haven't implemented this form of learning. We discuss the idea further in Chapter 8—further work

Each topic area and format (letter, books, etc.) brings with it a different style of usage of the language, different word counts and so on. We have described these as *contexts* in the preceding.

Our image corpora are less complete in their coverage as they cover only places within the UK in the case of the color corpus, and an area surrounding Boston Massachusetts USA in the case of the monochrome corpus. Nevertheless, especially in the color corpus, many interesting contexts are covered, including terrain contexts varying from urban to rural and from inland to coastal. Additionally the images are drawn from two different sets of SPOT images that used different imaging parameters. The different terrain contexts result in different grammatical contexts and label frequencies. An obvious example is that the "sea" label occurs more frequently in coastal frames than in inland frames. Similarly parse rules involving regions labeled "sea" only occur in coastal contexts. Different imaging modes give rise to different sets of optical characteristics.

One way of extracting the contexts is to label the contexts manually. There is however, no reason to believe that a human expert would know what the contexts were. One reason to automate context induction is that there may be contexts that can be found automatically that are not obvious to the human expert. The rules and labels were provided in the corpus in the form of annotations because they provide semantic markers necessary for the interpretation of the image. The human expert does understand what the regions in the image represent well enough to provide a "label" and similarly the human expert knows about the relationship between adjacent parts well enough to provide grammatical rules—just as the linguist can parse examples sentences in a corpus.

A letter to the editor contains many sentences and an image contains many regions. By comparing the sets of rules that occur in a single document or image with sets from other documents or images we can learn about similarities between certain subsets.

There are many ways in which images may appear to be similar:

1. The images may contain regions with similar optical characteristics—optical contexts.

2. The images may contain similarly labeled regions—lexical contexts.

3. The images may contain similar grammatical constructions—language contexts.

We build context models by collecting for each image n-dimensional points in the three categories. For optical contexts we are concerned with the descriptions of the regions.

Testing Procedure

In order to test our interpretation system, we divide the corpus into two sets—a training set, and a test set. To do this we have a function "allButN" which randomly chooses the training images from the corpus and returns a list of training images and a list of test images. "allButN" can be invoked arbitrarily many times producing each time a different training set and test set from the same corpus. In order to describe the behavior of the system we have taken one particular result of "allButN" which divides the color corpus as follows:

Training Set

S-UK-04-BELFA-C-1	S-UK-04-BELFA-C-2	S-UK-04-BELFA-C-3	S-UK-04-BELFA-C-4
S-UK-04-BELFA-C-5	S-UK-04-BELFA-C-6	S-UK-05-BENEV-C-1	S-UK-06-BODML-C-1
S-UK-06-BODML-C-2	S-UK-06-BODML-C-4	S-UK-07-BRIGH-C-1	S-UK-07-BRIGH-C-2
S-UK-07-BRIGH-C-3	S-UK-07-BRIGH-C-4	S-UK-07-BRIGH-C-5	S-UK-12-EDINB-C-1
S-UK-12-EDINB-C-2	S-UK-12-EDINB-C-3	S-UK-12-EDINB-C-4	S-UK-12-EDINB-C-6
S-UK-12-EDINB-C-7	S-UK-14-FOLKS-C-1	S-UK-14-FOLKS-C-2	S-UK-14-FOLKS-C-3
S-UK-17-HADRI-C-1	S-UK-17-HADRI-C-2	S-UK-17-HADRI-C-3	S-UK-17-HADRI-C-5
S-UK-17-HADRI-C-6	S-UK-20-ISMAN-C-1	S-UK-20-ISMAN-C-2	S-UK-21-JERSY-C-1
S-UK-21-JERSY-C-2	S-UK-23-KINGS-C-1	S-UK-23-KINGS-C-3	S-UK-24-LAKES-C-1
S-UK-24-LAKES-C-2	S-UK-24-LAKES-C-3	S-UK-24-LAKES-C-4	S-UK-24-LAKES-C-5
S-UK-25-LANDS-C-1	S-UK-25-LANDS-C-2	S-UK-28-MENDI-C-1	S-UK-28-MENDI-C-2
S-UK-28-MENDI-C-3	S-UK-28-MENDI-C-4	S-UK-29-MERSF-C-1	S-UK-29-MERSF-C-2
S-UK-29-MERSF-C-3	S-UK-29-MERSF-C-4	S-UK-29-MERSF-C-5	S-UK-31-MIDDL-C-1
S-UK-31-MIDDL-C-2	S-UK-31-MIDDL-C-3	S-UK-31-MIDDL-C-4	S-UK-31-MIDDL-C-5
S-UK-32-MILBR-C-1	S-UK-35-PEMBR-C-1	S-UK-35-PEMBR-C-2	S-UK-35-PEMBR-C-3
S-UK-35-PEMBR-C-4	S-UK-37-PETER-C-1	S-UK-39-SANDW-C-1	S-UK-39-SANDW-C-2
S-UK-39-SANDW-C-3	S-UK-39-SANDW-C-4	S-UK-39-SANDW-C-5	S-UK-39-SANDW-C-6
S-UK-39-SANDW-C-8	S-UK-40-SHEPP-C-1	S-UK-40-SHEPP-C-2	S-UK-40-SHEPP-C-3
S-UK-40-SHEPP-C-4	S-UK-43-SOUTH-C-1	S-UK-43-SOUTH-C-2	S-UK-43-SOUTH-C-3
S-UK-43-SOUTH-C-4	S-UK-43-SOUTH-C-5	S-UK-43-SOUTH-C-6	S-UK-45-THETF-C-1
S-UK-45-THETF-C-2	S-UK-45-THETF-C-3	S-UK-45-THETF-C-4	S-UK-46-WEALD-C-1
S-W-26-GLAS-C-01	S-W-26-GLAS-C-02	S-W-26-GLAS-C-03	S-W-26-GLAS-C-04
S-W-26-GLAS-C-05	S-W-26-GLAS-C-06	S-W-26-GLAS-C-07	S-W-26-GLAS-C-08
S-W-26-GLAS-C-09	S-W-26-GLAS-C-10	S-W-28-GOWE-C-1	S-W-28-GOWE-C-2
S-W-28-GOWE-C-3	S-W-28-GOWE-C-4	S-W-28-GOWE-C-5	

Test Set

S-UK-06-BODML-C-3	S-UK-12-EDINB-C-5	S-UK-24-LAKES-C-6	S-UK-39-SANDW-C-7
S-W-26-GLAS-C-11	S-UK-23-KINGS-C-2		

The results of interpreting the training set described below are for the training set shown above but the results are typical of a randomly chosen set of training images for the color corpus.

Region Label Contexts

A region context is a context in which images have similar regions in similar proportions in the image. For example images of coastal towns are similar in the kinds of region that they contain.

From the labeled training set, for each region we know the size of the region and the expert labeling of the region. On a per image basis we compute the number of pixels of each label

that the image has. There are 42 different kinds of label, so each image can be represented as a point in the 42 dimensional space defined by the 42 different labels. The images are interpreted as region label contexts by finding the MDL description of the points using the PCD clustering algorithm described above.

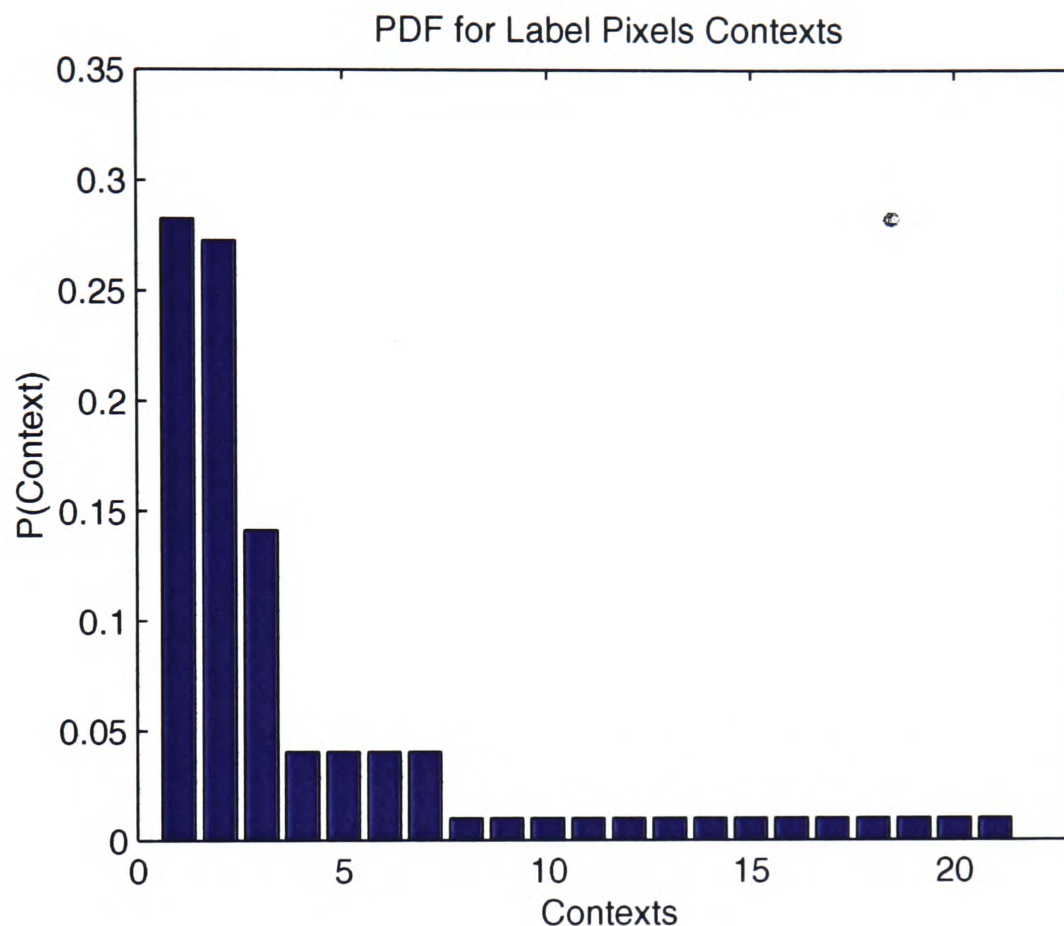


Figure 6.15: PDF for Label Pixels Contexts

Figure 6.15 shows the division of the training set into region label contexts as a PDF. The PDF is the probability $P(RLC_n)$, the prior probability that an image belongs to the region label context RLC_n.

Even though the images of the color corpus are reasonably homogeneous there is some useful division of the images into separate contexts.

The label pixels image descriptions are divided into 21 contexts as follows:
Context 1

S-UK-17-HADRI-C-6	S-W-26-GLAS-C-07	S-UK-23-KINGS-C-1	S-UK-12-EDINB-C-6
S-UK-43-SOUTH-C-6	S-UK-29-MERSF-C-5	S-UK-39-SANDW-C-8	S-UK-39-SANDW-C-6
S-UK-07-BRIGH-C-2	S-UK-12-EDINB-C-7	S-UK-43-SOUTH-C-4	S-UK-29-MERSF-C-4
S-W-28-GOWE-C-3	S-UK-29-MERSF-C-3	S-W-26-GLAS-C-10	S-UK-23-KINGS-C-3
S-UK-04-BELFA-C-4	S-W-26-GLAS-C-08	S-UK-39-SANDW-C-4	S-UK-07-BRIGH-C-3
S-UK-07-BRIGH-C-1	S-UK-39-SANDW-C-2	S-UK-39-SANDW-C-5	S-UK-43-SOUTH-C-2
S-UK-04-BELFA-C-5	S-W-26-GLAS-C-01	S-UK-39-SANDW-C-3	S-UK-31-MIDDL-C-3

Context 2

S-UK-07-BRIGH-C-4	S-UK-07-BRIGH-C-5	S-UK-14-FOLKS-C-1	S-UK-28-MENDI-C-1
S-UK-45-THETF-C-3	S-UK-24-LAKES-C-5	S-UK-17-HADRI-C-3	S-UK-24-LAKES-C-1
S-UK-28-MENDI-C-4	S-W-26-GLAS-C-05	S-UK-45-THETF-C-4	S-UK-28-MENDI-C-3
S-UK-17-HADRI-C-1	S-UK-45-THETF-C-2	S-UK-40-SHEPP-C-1	S-UK-35-PEMBR-C-4

S-UK-24-LAKES-C-2	S-UK-40-SHEPP-C-3	S-UK-35-PEMBR-C-2	S-UK-06-BODML-C-4
S-UK-35-PEMBR-C-3	S-UK-12-EDINB-C-3	S-UK-12-EDINB-C-1	S-UK-04-BELFA-C-6
S-UK-24-LAKES-C-3	S-UK-04-BELFA-C-1	S-W-28-GOWE-C-4	

Context 3

S-W-26-GLAS-C-02	S-UK-40-SHEPP-C-2	S-UK-06-BODML-C-1	S-UK-35-PEMBR-C-1
S-UK-06-BODML-C-2	S-W-28-GOWE-C-1	S-UK-31-MIDDL-C-5	S-UK-40-SHEPP-C-4
S-UK-21-JERSY-C-1	S-W-28-GOWE-C-5	S-UK-21-JERSY-C-2	S-UK-14-FOLKS-C-2
S-UK-20-ISMAN-C-1	S-UK-25-LANDS-C-2		

Context 4

S-UK-25-LANDS-C-1	S-W-26-GLAS-C-03	S-UK-43-SOUTH-C-3	S-UK-43-SOUTH-C-5

Context 5

S-UK-20-ISMAN-C-2	S-UK-29-MERSF-C-2	S-UK-39-SANDW-C-1	S-UK-04-BELFA-C-3

Context 6

S-UK-17-HADRI-C-2	S-UK-31-MIDDL-C-2	S-UK-46-WEALD-C-1	S-UK-31-MIDDL-C-1

Context 7

S-W-26-GLAS-C-06	S-UK-12-EDINB-C-4	S-UK-29-MERSF-C-1	S-W-28-GOWE-C-2

Single Image Contexts 8-21

S-UK-31-MIDDL-C-4	S-W-26-GLAS-C-04	S-UK-04-BELFA-C-2	S-UK-14-FOLKS-C-3
S-UK-43-SOUTH-C-1	S-UK-24-LAKES-C-4	S-UK-32-MILBR-C-1	S-UK-45-THETF-C-1
S-W-26-GLAS-C-09	S-UK-05-BENEV-C-1	S-UK-12-EDINB-C-2	S-UK-17-HADRI-C-5
S-UK-28-MENDI-C-2	S-UK-37-PETER-C-1		

As an example, figure 6.16 shows the four images of context 7. The four images are clearly similar in their content but their optical characteristics are not. In particular image "s-w-26-glas-c-04" has very different optical characteristics from the other images.

Optical Contexts

Interpreting the images as optical contexts is similar to producing label contexts but instead of associating regions with their label the regions are associated with the model that the region belongs to. Recall that a single label may result in many optical models. Each region in the corpus belongs to precisely one optical model.

Optical contexts are derived in two steps:

1. Optical models for all region types are generated as outlined in section 6.4.

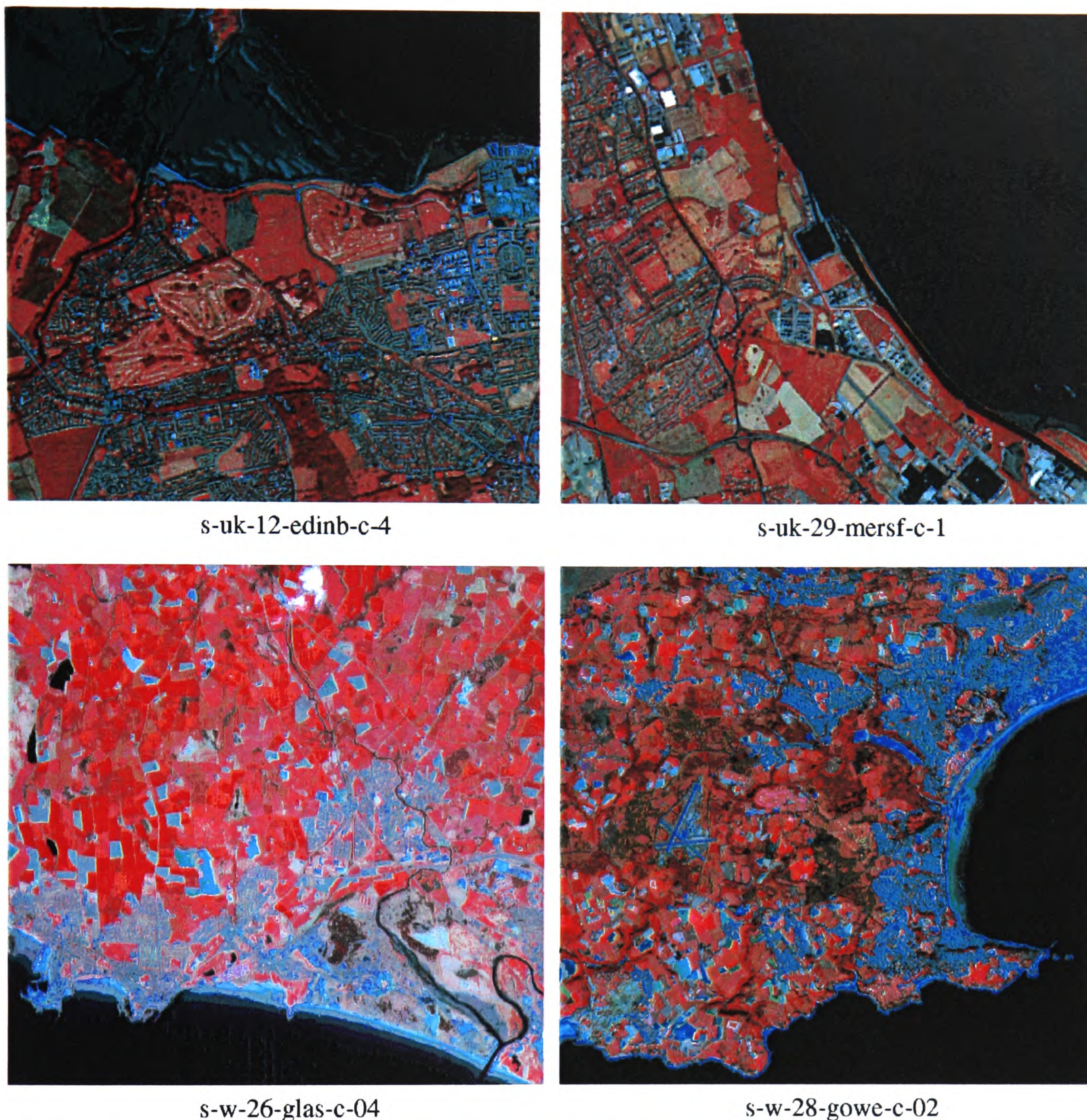


Figure 6.16: The 4 Images of Label Pixels Context 7

2. Each region of each image in the training set belongs to one of the optical models (there may be many optical models for a single label). If there are n optical models we can define the images as points in that n dimensional space. For each image in the training set count how many pixels are assigned to each optical model. For each image, the resulting n dimensional point represents the optical characteristics of the image. The collection of all such points from the training set are grouped in to "contexts" using the MDL PCD algorithm described above.

For the color training set described in this thesis, the above procedure produces 13 contexts. Figure 6.17 shows the division of the training set into optical contexts. The PDF shown provides estimates of the prior probability $P(OpticalContext)$.

By counting the frequency of occurrences of optical models in each context we can obtain

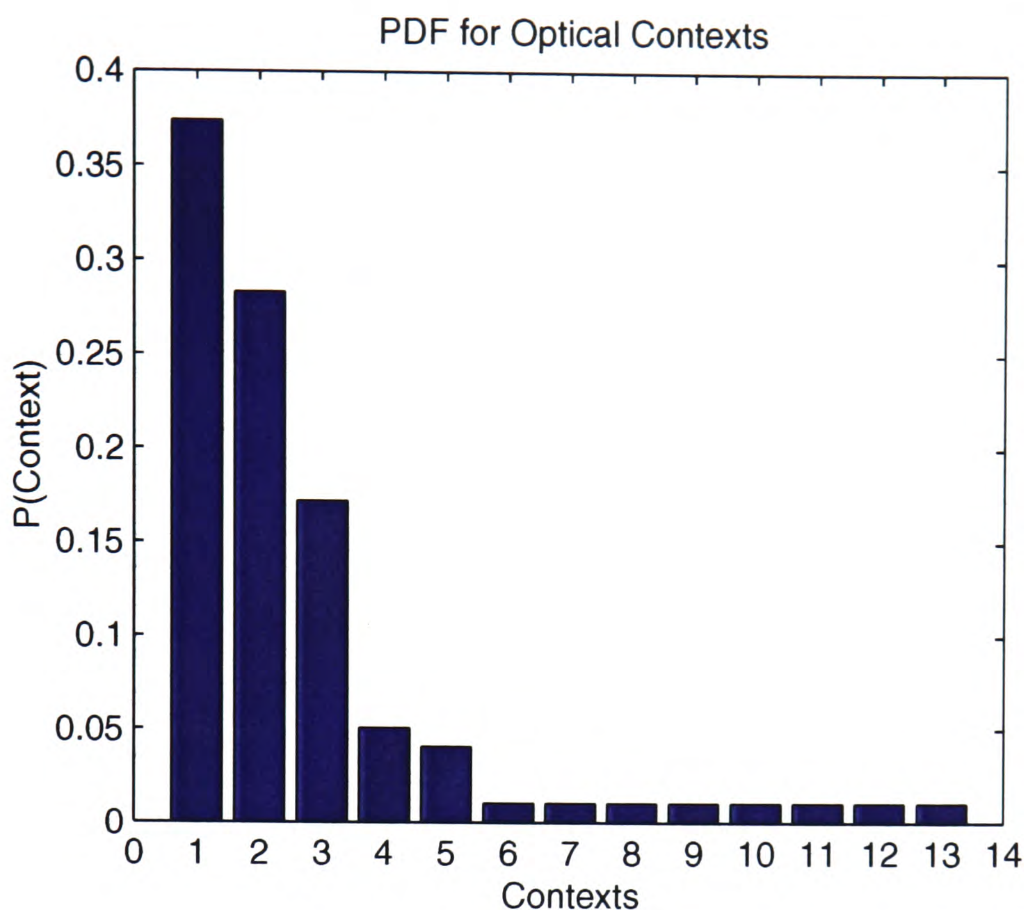


Figure 6.17: PDF for Optical Contexts

$P(OpticalModel|OpticalContext)$.

6.5.1 Summary of the Description of the Corpus

In this chapter we have discussed the interpretation of the corpus as a collection of statistical models found by an MDL clustering algorithm. We have specifically developed:

1. Optical models based on an arbitrary collection of image planes.
2. Optical contexts for grouping the optical models.
3. Region label contexts.

We additionally collect grammatical rules as described in Chapter 5 although we don't divide them into clusters because the limited size of our corpus (105 images in the color corpus) is insufficient for that purpose.

Finally, we collect region outline sequences as described in Chapter 4.

The description is produced by a collection of agents that fit models to the data. Most of these agents fit statistical models by using the PCD clustering algorithm developed in this chapter.

The description generated from the corpus serves as a specification for how we want the aerial image interpretation program to behave. Chapters 4, 5, and 6 have developed the models and agents for interpreting images and the corpus. What remains is to build the mechanism that ties these agents together in the form of a program. We refer to that as the program (or

circuit) synthesis problem. In Chapter 8 we bring together the pieces developed in the preceding chapters and show how we can use the specification (description) developed in this chapter as the basis for synthesis of a program, and how reflection allows the program to self-adapt to changing contexts.

6.6 Conclusion

We have developed a novel algorithm for the decomposition of complex models into collections of simpler models. This forms the backbone mechanism for interpreting the corpora. The algorithm has some important features:

1. The algorithm supports non-convex shapes.
2. The algorithm uses the MDL criteria for interpretation upon which the GRAVA architecture depends.
3. The algorithm doesn't over fit. The algorithm doesn't try to circumscribe a set of data points. It simply tries to separate collections of points by finding the best place to cut.
4. The algorithm is fast because it only searches for cut points along eigenvector dimensions.
5. The algorithm has performed very well on our data: the high dimensionality of the data makes visual presentation of the space almost impossible but the results of clustering the images into contexts appears to produce clusters of images that are similar in either the optical or language domains.
6. The algorithm is non-parametric which removes one more barrier to automation.

We did not use the algorithm described here to separate the rules into clusters because our corpora had too few images to provide a reasonable set of parse rules. If a larger corpus had been available the same method could have been used.

By using PCD we were able to provide good optical models for region labeling and we were able to divide the corpus into contexts in two dimensions: optical contexts, and label contexts.

The result of running these algorithms over the corpora is a description of the corpora that includes:

1. Optical contexts that have similar optical characteristics.
2. Optical models that for a given optical context can support region labeling.
3. Label contexts that support parse contexts.
4. Grammar models (rules).

The real contribution of this chapter is the separation of the step of interpreting the corpus, from the larger notion of "learning from a corpus". "Learning from the corpus" is then completed by the second stage: synthesizing a program from the resulting description.

The principal component decomposition algorithm was developed because it captured our descriptive needs directly. We want to be able to choose collections of data points that can be described by an MDL representation of the data as statistical models. The statistical models are required because we wish the corpus to be predictive of other data points that may be encountered. Our algorithm has precisely those characteristics.

The description produced by the processes described in this chapter fairly defines the space of examples represented by the corpus. In Chapter 8 we describe how the description generated from the corpus is itself *interpreted* as a specification for an image interpretation program, how its performance is measured at runtime, and how, when necessary, the program is re-synthesized.

Chapter 7

Reflection and Self-Adaptation

7.1 Introduction

From Chapters 3–6 we have the pieces ready to assemble into an image understanding system.

First however, we return to the guiding idea that we set out to explore in this thesis.

1. Vision (and Robotics) systems lack robustness. They don't know what they are doing, especially when things change appreciably (i.e. in situations where technologies such as neural nets are ineffective).
2. Reflective architectures—an idea from AI—offers an approach to building programs that can reason about their own computational processes and to make changes to them.
3. Reflection seems like an approach to making software self-adaptive.

The purpose of the reflective architecture is to allow the image interpretation program to be aware of its own computational state and to make changes to it as necessary in order to achieve its goal.

However, much of work on reflective architectures has been supportive of human programmer adaptation of languages and architectures rather than self-adaptation of the program by itself.

This chapter makes a necessary detour from development of a vision system by discussing the final—and most critical—piece of the puzzle. We

1. introduce the idea of reflection;
2. give some detail of how to use reflection as a means of implementing self-adaptation; and
3. introduce "program synthesis" as a key component in our reflective architecture that permits GRAVA programs to self-adapt by using the mechanisms of reflection. To do this we consider the program to be an *interpretation* of a specification for the behavior of the program. The compiler or program synthesis module is simply the interpreter of the specification. We find that we can use the MDL agent architecture developed in Chapter 3 as the basis for implementing such a compiler. Specifically, by viewing compilation as theorem proving, we show how to make reflection generate modules for use by another program. By introducing the concepts of "Fidelity and Quality" of an agent, we are able to extend reflective architectures to deal with uncertain information, characteristic of that manipulated by vision systems.

The next chapter applies the novel reflective architecture developed in this chapter to the problem domain of aerial image understanding.

Given the preceding chapters the approach to self-adaptation taken in this project is easily described:

1. The *behavior* of a human expert is measured in the form of statistical models by constructing a corpus.
2. The human behavior which covers several different imaging scenarios is broken down into contexts. Contexts exist for different levels of the interpretation problem. Each context defines an expectation for the computational stage that it covers. Contexts are like frames and schemas; but because the contexts are gathered from the data automatically using clustering techniques developed in Chapter 6 it is not necessary to define them by hand. Another important benefit of developing contexts as statistical models is that a straightforward procedure exists for measuring the closeness of fit of an input and output to those expected by the context. We can therefore automatically produce not only the contexts themselves but pre-tests of the applicability of a context to the data upon which it is operating.
3. Given a model of the world, we can predict the context that we are operating in. A program to interpret the image can be generated from that context. This is done by *compiling* the context into a program by selecting the appropriate compilation agents.
4. The program that results from compiling a context can easily know the following things:
   (a) What part of the specification gave rise to its components.
   (b) Which agents were involved in the creation of its components.
   (c) Which models were applied by those agents in creating its components.
   (d) How well suited the current program is to dealing with the current input.
5. The division of knowledge into agents that perform basic image interpretation tasks and agents that construct programs from specifications is represented by different reflective levels.

Reflection normally permits introspection and modification, but it is a novel idea to automate the decision process necessary to decide what changes to make to the computational state. In this chapter we develop support for determining the need for program modification and the mechanism for making the changes (which we refer to as program synthesis).

7.1.1 Interpretation Problems

The problem of self-adaptive software is to respond to changing situations by re-synthesizing the program that is running. To do this we reify the software development process.

Layers of Interpretation: An Example

A key idea in our formulation of our reflective architecture is that problems can often be described in terms of interconnected layers of interpretation forming a hierarchy of interpretation problems. A simple and familiar example of such a layered view is the process of how large software projects are executed.

Large software projects start out with a requirements document. This document says what the program should do but doesn't say how it should be done. Someone *interprets* the requirements document as a software system and produces a set of specifications for the components of the software system that satisfies the requirements. The specifications are then *interpreted* as a program design. The program design specifies the procedures that make up the program that implements the specification. Finally a programmer interprets the program design to produce a body of code. If care is taken to retain back pointers it is possible to trace back from a piece of code to the part of the design that it interpreted. Parts of design should be traceable to the parts of the specification they interpret and parts of the specification should be traceable to the parts of the requirements document that they interpret.

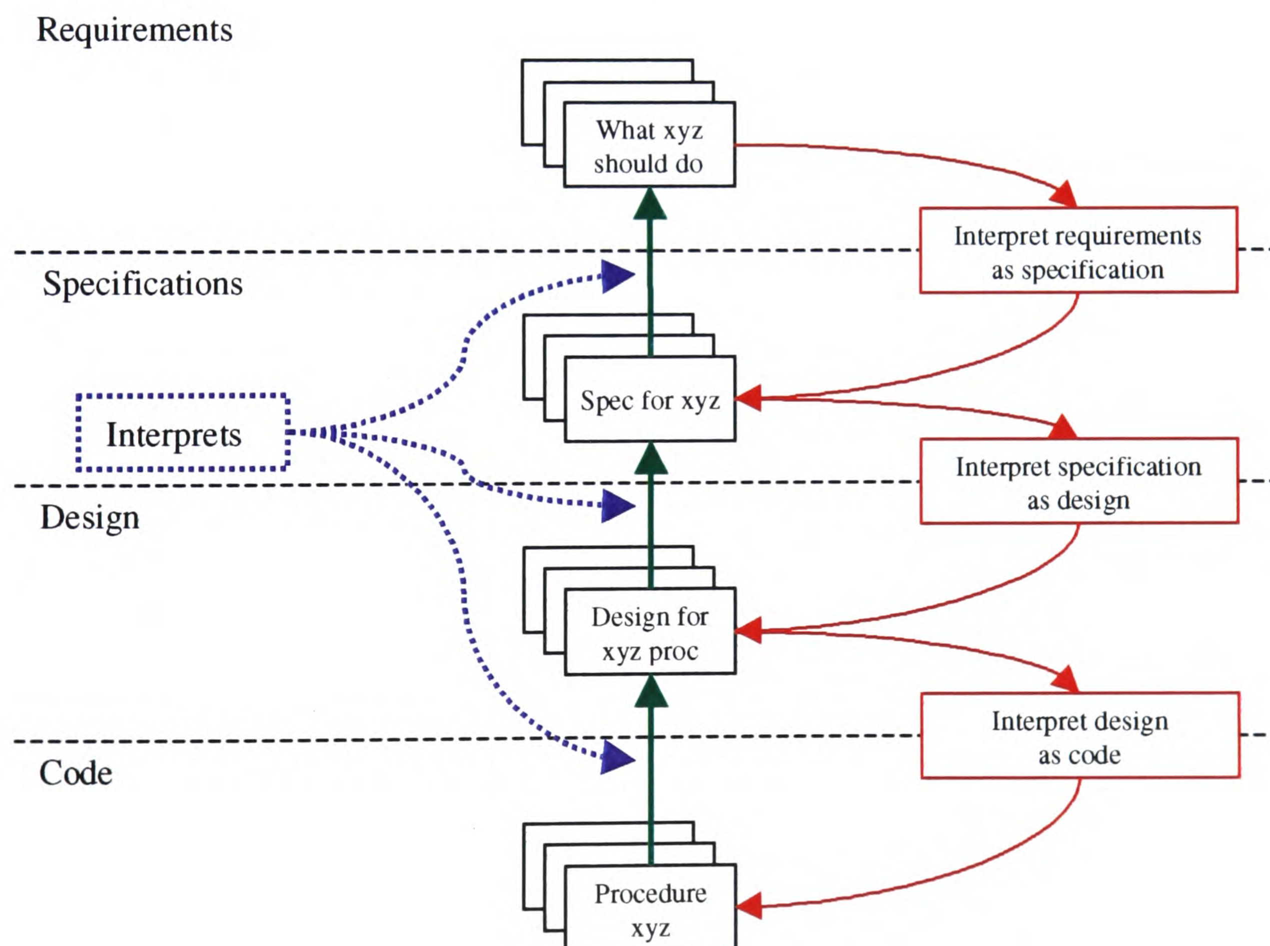


Figure 7.1: Example of the relationship between levels of interpretation

Figure 7.1 shows the relationship between different levels of interpretation in the software development example. Green arrows show "tracability" while the red arrows show how interpretation at each level "generates" the level below.

Very large software projects, especially military software projects, are managed in this way.

When requirements change, as they often do in the lifetime of a software system, it is possible to trace which pieces of the system are affected. In this example, at each level, an input is interpreted to produce an interpretation that is used as the input at a subsequent level. The act of interpretation at each level involves applying models to the input.

Each component of the system "knows" what it is doing to the extent that it knows what part of the level above it implements (interprets).

7.1.2 Overview of the Chapter

In Section 7.2 we provide an overview of reflection techniques. In Section 7.3 we extend the architecture developed in Chapter 3 to support self-adaptation. In Section 7.4 we develop a compiler for agents based on a statistical MDL theorem prover.

7.2 Prior Work

The self-adaptive architecture described in this chapter uses statistical knowledge of its own behavior in order to know when it is not well suited to the problem at hand and then to reason about the same knowledge in order to reconstruct itself in a more appropriate way. We extend the MDL agent architecture that was introduced in Chapter 3 introducing the idea of reflective layers and by introducing a reflectUp operator to move up through the reflective levels. We begin with an overview of the ideas, history, and implementation techniques of computation and procedural reflection.

7.2.1 Reflection

Reflection is a technology that enables programs to be written that can manipulate, to some degree, the computational nature of the programs themselves. Reflection opens up the computational state so that it can be manipulated within the language or system. Wherever it is useful to extend the computational framework beyond what was originally envisioned or to introspect on the state of a computation, reflection is a way of achieving it.

Initially reflection was problematic: it did not appear possible to achieve it efficiently. Besides being slow and hard to use there were (and still are) problems associated with providing formal semantics for reflective languages because reflection provides a way of allowing semantics to be changed dynamically.

Today, ways of implementing reflection efficiently have been developed. Reflection is available in robust industry standard languages (Kiczales, des Rivieres, & Daniel 1993), and has been applied to a wide range of problem domains (Maes & Nardi 1988; Honda & Tokoro 1992; Lamping *et al.* 1992). Providing a formal semantics remains an open research problem, but that lack doesn't interfere with the usefulness of reflection in solving practical problems.

Reflection is a tool in the same sense that differential equations are. If you want to solve a problem that involves the dynamics of a system that changes over time, it is natural to use

differential equations. If you want to address a problem in which you wish to reason about the computational state of the program, reflection is a natural way of achieving it. In both cases, the technology provides a linguistic framework to aid in the problem solution.

Reflection is useful for a vast array of problems. For example, with a reflective language, it is easy to implement class browsers and debuggers. Reflection is often employed as a way of making a system more flexible. It does this by exposing semantics and thereby enabling them to be modified. The idea of being able to reason about the computational nature of a program is a remarkably useful idea.

It should be noted that reflection never provides a capability that couldn't easily be provided in an *ad hoc* way. However, it is a systematic way to provide many different capabilities and that is why it is valuable.

In order to explain what reflection is, I briefly describe how reflection can be implemented.

Smith's Recipe

There are many implementation techniques that have been developed since Smith originally introduced reflection in 1982 (Smith 1982). All current implementation techniques are based on Smith's original general recipe for reflection.

1. There must be an **embedded** semantic account. This embedded account is the *reified semantics.* It usually consists of data structures or functions that define the semantics of the system.

2. The embedded account must be **causally connected** to the semantics of the program. The data structures or functions that represent the semantics must be used by the system in running the program. That way, the computation state of the program can be interrogated (introspection) and by modifying the data-structures or functions (such as by specialization), the semantics of the program can be modified.

3. There needs to be an appropriate **level of detail**. Typically, the entire state of the computation does not need to be interrogated, nor does the entire semantics need to be subject to modification. If the full semantics are both accessible and modifiable, the system is completely open. In solving real problems, a sense of proportion is necessary, and success is achieved by picking what subset of the entire semantics are to be reified. This requires an analysis of the goals of the reflective system in order to determine which pieces of semantics to reify.

The two most common approaches to implementing reflection are:

- Reflective Tower of Interpreters (Smith 1984); and
- Meta-Object Protocols (Maes & Nardi 1988; Kiczales, des Rivieres, & Daniel 1993).

The former approach forms the backbone of much of the theoretical work being done in reflection, while the latter is used widely in systems that must have good performance. I briefly describe how these two approaches work.

Tower of Interpreters

The intuitive notion of a tower of interpreters is that there is an **implicit** infinite tower of interpreters each implementing the level below it.

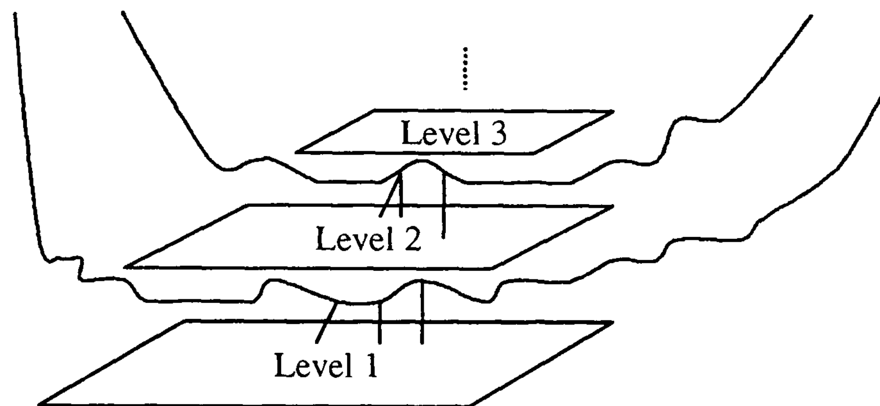


Figure 7.2: Reflective Tower of Interpreters

A tower of interpreters solution is achieved with the following steps.

First, the problem domain is structured as a language. If the reflective entity is already a language, this is trivial. If the system to be made reflective is an architecture, an architecture language is developed to represent the architectural semantics.

Second, a meta-circular interpreter for an *empty* language is developed. The empty language has a context for the program that it is interpreting. The context generally consists of (1) an environment that specifies bindings of the identifiers, and (2) control to specify the point of execution in the program (continuations). The context can contain whatever is necessary to represent the state of computation in the language being defined. The meta-circular interpreter doesn't implement any of the primitives of the language the way a normal meta-circular interpreter would. Instead, it is only able to distinguish between *reflective* procedures, and *normal* procedures. When a normal procedure is encountered, the meta-circular interpreter interprets the invocation of the procedure in the interpreted program context.

A way of defining a reflective procedure is added to the language. In Smith's 3-Lisp, the form **(lambda reflect** [[**args**] **context**] **body)** achieves this purpose.

Finally, the language is populated into the (empty) meta-circular reflective interpreter by defining reflective procedures for all of the primitive operations of the language. In a normal meta-circular interpreter, these procedures would be hardwired into the meta-circular interpreter. By defining them as reflective procedures, they still get evaluated in the context of the interpreter,

but they are available in the environment of the interpreter so that they can be accessed and even changed by user written (lambda reflect ...) procedures. (lambda reflect ...) means that the body will be evaluated in the interpreter's context. The body can therefore reason about the interpreter's context and thus the computational state of the program and can mutate it.

The semantics of the base language is defined by the set of reflective procedures defined for the language.

In principle, arbitrarily many reflective levels can be involved at any point in the program since (lambda reflect ...) procedures can be nested. In practice it is rare to reflect more than a few levels. Since the scheme depends upon having a meta-circular interpreter, it forces the resulting system to be interpreted rather than compiled. The language definition and interpretation semantics can be changed dynamically during interpretation, so it is not possible to freeze the semantics and compile the result in the way that is typically done for languages that don't employ meta-circular interpreters.

The implicit infinite tower of interpreters is similar to the potentially infinite recursive stack for a recursive procedure. In both cases, the infinite nature is conceptual and bounded by practical matters (like available storage). In the case of a recursive procedure, each level allows refinement of the problem space. With a reflective tower of interpreters, each level allows refinement of the language semantics.

Meta-Object Protocols

Because of the desire to have reflective capabilities in production quality languages and systems the meta-object approach to reflection was devised which allows for more efficient implementation.

The meta-object approach uses meta-information in object-oriented class systems in order to achieve the reflection. Here is how it works:

If we are dealing with a language, each construct within the language for which we wish to reflect is defined as a class. For example, in an object oriented language with classes and methods, we would represent a class definition as a class. Similarly, a method definition would be represented as a class. Instances of these classes are called meta-objects.

Next, the semantics of the objects thus represented are defined as methods of the appropriate meta-object classes. The set of these methods defines the protocol for manipulating these objects. This protocol is the meta-object protocol. The meta-object protocol must include all interesting semantics of the object being modeled. For example, in the case of the meta-object for a class definition, the protocol must include methods for the creation and initialization of new instances of the class as well as methods for handling requests that the language allows of instances of the class. In the case of methods, if the language supports method combination, there must be a method that provides the method combination algorithm.

Next we make sure that the compiler uses the meta-object protocol and emits code that uses the meta-object protocol for all manipulations of the objects. This provides the causal

connectedness.

By making sure that access is provided to the meta-objects—such as by providing accessors in the metaclasses—the programmer of the language has a mechanism for introspecting on the state of computation. By modifying the definitions of the meta-objects, the programmer can modify the semantics of the language insofar as it is covered by the meta-objects. Usually this modification is achieved by subclassing the default behavior and then extending it. In principle meta-objects could be provided for every construct in the language making the language completely open for both introspection and extension. In practice, the interesting semantics in object oriented languages are mostly contained in the class system, so usually only the linguistic constructs that implement the class system are represented by meta-objects.

There is another reason why meta-objects are usually not provided for constructs outside of the class system: performance. If trivial constructs like iteration were represented by metaobjects, it would not be possible to open code the compilation of the iteration because the user might want to change the semantics of iteration. The class system differs in that the operations on it are already complex and would generally not be open coded anyway. By indirecting through a meta-object for these operations an insignificant overhead is involved in providing the reflective capability. Class systems in many cases are already implemented using metaclasses, so the overhead for reflection is minimal. It does however require that the meta-objects exist at runtime.

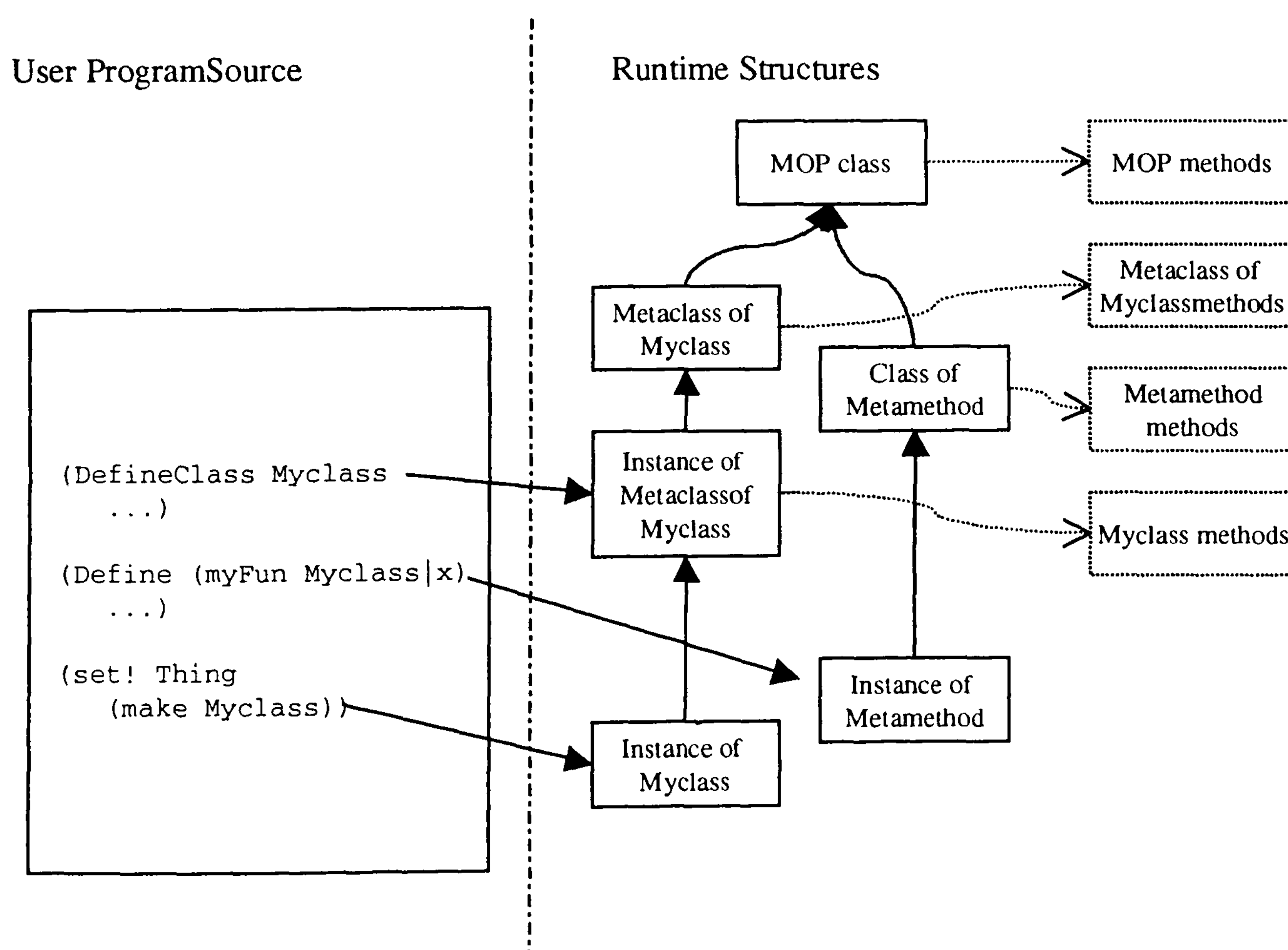


Figure 7.3: Runtime MOP Structures

Figure 7.3 shows a highly simplified view of the runtime structures that support a meta-object protocol implementation. The left depicts a user program in source form. The right depicts runtime objects in solid boxes and methods in dashed boxes.

The program depicted on the left defines a class (Myclass), defines a single method on that class (myFun), and finally creates an instance of Myclass.

The definition of Myclass gives rise to a metaclass that implements the user class Myclass. User-defined methods on the user class Myclass would be methods of the metaclass of Myclass.[1] The Metaclass for Myclass would have inherited methods for the creation of instances and all other maintenance functions associated with instances of the Myclass.

The dashed box "Myclass methods" contains methods that implement the user program. When a user program adds, removes, or modifies methods in that box, he is modifying the semantics of his program.

The semantics that govern the behavior, combination order, and any other semantics of how user defined methods operate are implemented as methods in the dashed box "Metamethod methods". The programmer can modify these methods to change the semantics of the *language* with respect to how user defined methods are manipulated by the language.

The semantics that govern how instances of the class Myclass are manipulated are implemented as methods in the dashed box "Metaclass of Myclass methods". These too may be modified by the programmer.

Given this structure, a programmer can be aware of all operations on objects in his program and can intercept any of these operations and even modify them. For example, it is easy to write metering tools and debuggers using reflection in the language.

If all you want to do is implement a debugger, it is not necessary to have reflection. Instead the debugger can be written outside of the language using special hooks designed in for the debugger by the implementers of the language. Traditionally this is how debuggers are written. The ability to implement metering tools and debuggers within a reflective language are an indication of the added power that reflection brings to a language: it is easy to write programs that reason about their own computation.

If the system being made reflective is an architecture rather than a language, the same approach works. For each object in the architecture, a class is defined that mediates all operations on the architectural object. All semantics are implemented as methods on these classes. The set of such methods defines the meta-object protocol for the architecture. The reflective capabilities in an architectural implementation are exported to the user in the form of class libraries for the meta-objects. For example, a reflective blackboard would have meta-objects for each level of the blackboard, for the objects that are posted to the blackboard, and for the BBKSs. All

[1] The relationship between the methods and classes may involve a less straightforward relationship than indicated on this diagram due to multi-methods. This issue is deliberately ignored here in order to keep the diagram simple.

of the semantics of the blackboard would be implemented as methods on the meta-objects. Reflection could then be used to straightforwardly monitor the state of the system. For example, the algorithm used to implement the control mechanism for deciding which BBKS to activate following a posting to the blackboard would be a method on one or more of the meta-objects and defined in the meta-object protocol. By specializing this activation method, it would be possible to monitor the flow of activations in the system. It would also be possible to change the activation semantics by partially or completely overriding the default behavior.

A blackboard implementation with a fixed built-in activation scheme would be very hard to work around if it wasn't appropriate to its intended task. In a reflective implementation, the semantics would be straightforwardly modified to the intended purpose. In a language, the semantics of the language could be tailored to the users needs often making the task much easier, and the implementation much cleaner.

We can summarize the value that reflection brings as follows:

- Reflection opens up a language or architecture allowing end users to customize the language or architecture to their own purpose. This has the double benefit of making the end user's code clean and making the language or architecture useful to a wider range of users by virtue of the added flexibility. It also has an important evolutionary value: as requirements for the language or architecture change over time, it is easy to modify the semantics of the system because the protocol provides a way of doing it directly. This is the software engineering aspect of reflection. It makes it easier to extend and maintain the system and the user code. This ability is very significant for anyone building a special purpose language or architecture.

- Reflection provides a way of writing programs that have access to and can reason about the computations that they are involved in. This is the higher level reasoning aspect of reflection.

The two kinds of reflection described above [2] are similar in that they both follow "Smith's Recipe". Meta-object reflection is static and tower of interpreter reflection is dynamic in the following sense.

Principally through specialization (subclassing) meta-object reflection allows code to be written that intercepts certain crucial operations and thereby allows introspection. Additionally, within the framework of the protocols, semantics of the language can be modified. Whatever kinds of introspection and modification are to be made however are decided statically by virtue of the specialization that is compiled into the program. Lisp programs, by virtue of a built-in compiler, can construct new subclasses on the fly and compile them at runtime to produce a dynamic effect but such dynamic effects are completely outside of the support of the meta-object

[2] Meta-object protocol based reflection and tower of interpreters based reflection

protocol. Lisp has always been able to manipulate its own structure in that way. That is not what reflection is about. Reflection is about the systematic understanding and manipulation of program semantics.

Tower of interpreter reflection offers a more dynamic form of reflection in that a program can arbitrarily jump between levels using lambda-reflect and have code executed at an arbitrarily meta level.

In the reflective architecture developed below both types of reflection are utilized. A meta-object based architecture allows the semantics of a "reflective level" and even the mechanism for the self-adaptation mechanism to be augmented. A tower of reflective layers of interpretation allows the program to self-adapt at an arbitrary meta level.

7.3 Reflective Interpreter for Self-Adaptation

The techniques for implementing reflection described above have become common in modern languages and architectures. Unlike traditional implementations, in this thesis we use reflection as a way of supporting self-adaptation. There are two principal differences in our use of reflection:

1. We do not wish to open the language up to the human programmer. We open up the program to itself so that by knowing what it knows it can use what it knows to alter itself in order to respond to changes in the real world.

2. We do not wish to change the semantics of the program/language, we wish to change the program itself.

A reflective layer is an object that contains one or more "interpreter". Reflective layers are stacked up such that each layer is the meta-level computation of the layer beneath it. In particular each layer is generated by the layer above it.

So far all of the MDL agent systems that we have discussed in chapters 3, 4, and 5 have been at the same level. A system can have an arbitrary number of levels. The example described in the introduction (Figure 7.1) has four levels. Most systems will have a small number of levels. The test domain developed to interpret aerial images contains three levels:

1. To interpret the corpus as a specification for program behavior.

2. To interpret the specification of program behavior as a program that interprets images.

3. To interpret images.

Each layer can reflect up to the layer above it in order to self-adapt.

```
(defineClass ReflectiveLayer
  ((description) ;; the (input) description for this layer
   (interpreter) ;; the interpreter for the description of this layer
   (knowledge)   ;; a representation of world knowledge at this level
```

```
(higherlayer) ;; the meta-level above this
(lowerlayer)));; the subordinate layer
```

A reflective layer is an object that contains the following objects.

1. *description:* the description that is to be interpreted. In the software development example the requirements level would contain a description of the requirement that is to be interpreted by the layer.
2. *interpreter:* a system consisting of one or more cascaded interpreters such as those described in chapters 3, 4, and 5 that can interpret the description.
3. *knowledge:* a problem dependent representation of what is known about the world as it pertains to the interpretation of the subordinate layer. The knowledge gets updated as the subordinate layer attempts to interpret its description. The knowledge is used in the synthesis of the interpreter for the subordinate layer.
4. *higherlayer:* the superior layer. The layer that produced the interpreter for this layer.
5. *lowerlayer:* the subordinate layer.

The semantics for a layer are determined by the *interpret*, *elaborate*, *adapt* and *execute* methods which we describe in turn below.

Figure 7.4 shows the relationship between reflective layers of the GRAVA architecture.

Reflective Layer "n" contains a description that is to be interpreted as the description for layer "n+1". A program has been synthesized either by the layer "n-1" or by hand if it is the top layer. The program is the interpreter for the description. That interpreter is run. The result of running the interpreter is the most probable interpretation of the description—which forms the new description of the layer "n+1". Layer "n" also contains a compiler. Actually all layers contain a compiler. Unless the layer definition is overridden by specialization the compiler in each layer is identical and provides the implementation as a theorem prover that compiles an interpreter from a description. The compiler runs at the meta level in layer "n" and uses the knowledge of the world at layer "n+1" which resides in level "n". It compiles the description from level "n+1" taking in to account what is known at the time about level "n+1" in the *knowledge* part of layer "n". The compilation of the description is a new interpreter at layer "n+1".

Below we describe the meta-interpreter for layers in GRAVA.

The interpret method is the primary driver of computation in the reflective architecture. The reflective levels are determined by the program designer. In order for the self-adaptive program to "understand" its own computational structure, each layer describes the layer beneath it. In self-adapting, the architecture essentially searches a tree of meta-levels. This is best understood by working through the details of the architecture.

The top level layer is manually constructed by the program designer. It must be because there is no higher level to defer to. That level defines its goal in the form of a description that

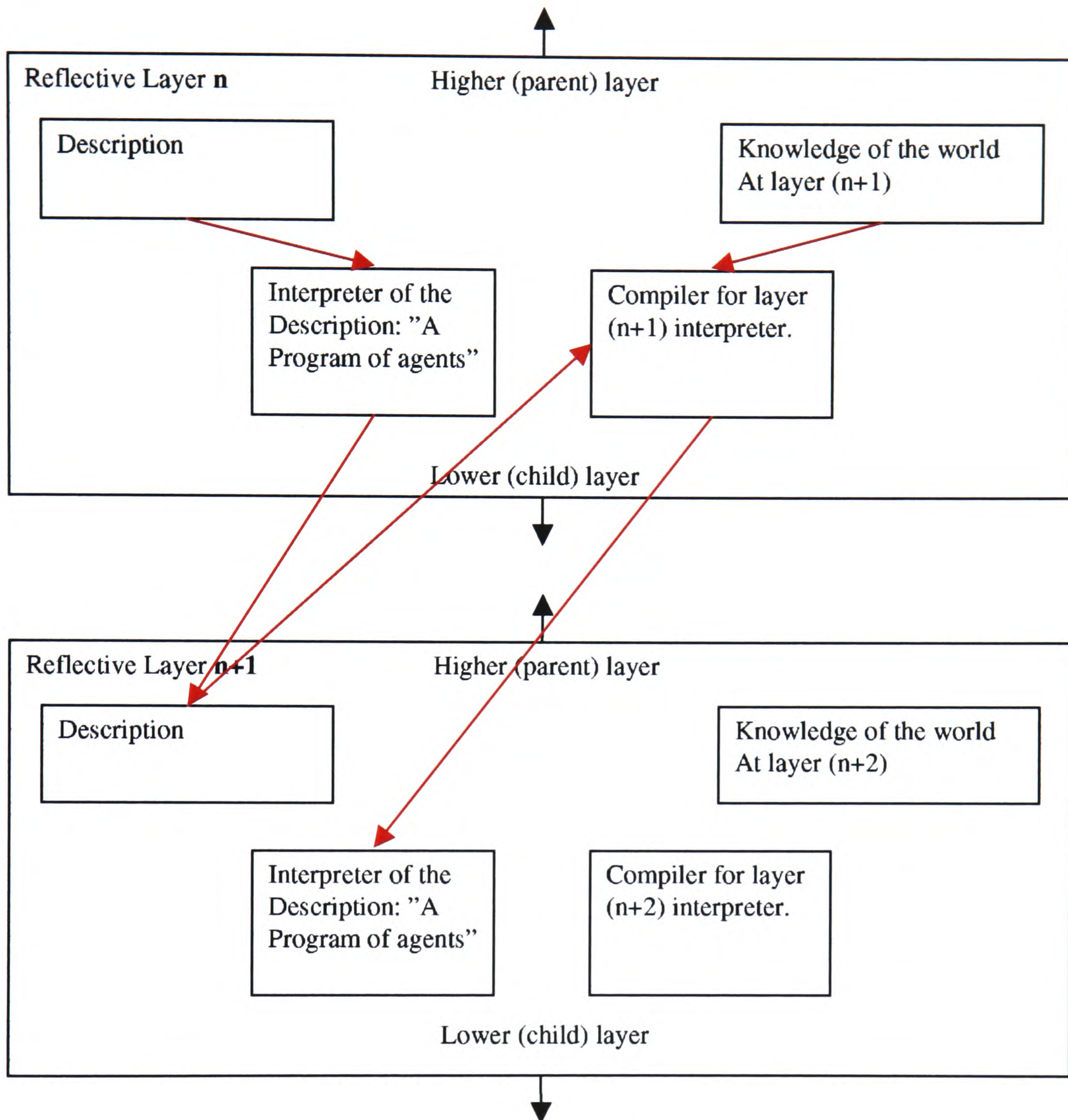


Figure 7.4: Meta-Knowledge and Compilation

must be interpreted. The collection of agents that interpret that description are provided by the human programmer. The program that those agents constitute is charged with the responsibility of producing the reflective layer immediate below. The lower level, once constructed, is then interpreted in order to bring about the desired behavior.

At some point the layers bottom out in a lowest level below which there is no further elaboration of layers. The lowest level layer has a description but no interpreter. The description at the lowest level is the result of the top level application of "interpret".

In the simplest of situations the top level application of "interpret" to the top layer results in the recursive descent of "interpret" through the reflective layers finally yielding a result in the form of an interpretation. Along the way however unexpected situations may arise that cause

the program to need to adapt. Adaptation is handled by taking the following steps:

1. Reflect up to the next higher layer (parent level) with an object that describes the reason for reflecting up. It is necessary to reflect up because the higher level is the level that "understands" what the program was doing. Each level "understands" what the level directly beneath it is doing.

2. The world model (knowledge) that is maintained by the parent level is updated to account for what has been learned about the state of the world from running the lower level to this point.

3. Armed with updated knowledge about the state of the world the lower level is re-synthesized. The lower level is then re-invoked.

Armed with that conceptual overview of the interpret procedure we now explain the default interpret method.

```
1:(define (interpret ReflectiveLayer|layer)
2:  (withSlots (interpreter description lowerlayer) layer
3:    (if (null? interpreter)
4:       description        ;; Return the description
5:       (elaborate layer)) ;; Create and populate the subordinate layer.
6:    (reflectProtect (interpret lowerlayer)
7:      (lambda (layer gripe) (adapt layer gripe)))))

8:(define (reflectionHandler ReflectiveLayer|layer gripe)
9:  (adapt layer gripe))
```

Line 3 checks to see if the layer contains an interpreter. If it does not the result of evaluation is simply the description which is returned in line 4. This occurs when the lowest level has been reached.

If there is an interpreter, the elaborate method is invoked (line 5). "elaborate" (described below) constructs the next lower reflective layer.

"reflectProtect" in line 6 is a macro that hides some of the mechanism involved with handling reflection operations.

(reflectProtect *form handler*) evaluates *form* and returns the result of that evaluation. If during the evaluation of *form* a reflection operation occurs the *handler* is applied to the layer and the gripe object provided by the call to reflectUp. If the handler is not specified in the reflectProtect macro the generic procedure reflectionHandler is used. The invocation of the reflection handler is not within the scope of the reflectProtect so if it calls (reflectUp ...) the reflection operation will be caught at the next higher level. If reflectUp is called and there is no extant reflectProtect the debug is entered. So if the top layer invokes reflectUp the program lands in the debugger.

When the reflection handler has been evaluated the reflectProtect re-evaluates the *form* thereby making a loop. Line 7 is included here to aid in description. It is omitted in the real code allowing the reflectionHandler method to be invoked. The handler takes care of updating the world model based on the information in *gripe* and then adapts the lower layer. The handler therefore attempts to self adapt to accommodate the new knowledge about the state of the world until success is achieved. If the attempt to adapt is finally unable to produce a viable lower level interpreter it invokes reflectUp and causes the meta level interpretation level to attend to the situation.

```
1:(define (elaborate ReflectiveLayer|layer)
2:  (withSlots (lowerlayer) layer
3:    (let ((interpretation (execute layer)) ;; lower level description
4:          (llint (compile layer interpretation)))
5:      (set! lowerlayer ((newLayerConstructor layer)
6:                         higherlayer: layer
7:                         description: interpretation
8:                         interpreter: llint)))))
```

The purpose of the elaborate layer is to build the initial version of the subordinate layer. It does this in three steps:

1. Evaluate the interpreter of the layer in order to "interpret" the layer's description. The interpretation of $layer_n$ is the description of $layer_{n+1}$.

   Line 3 invokes the interpreter for *layer* with *(execute layer)*. This simply runs the MDL agent interpreter function defined for this layer. The result of executing the interpreter is an interpretation in the form of a description.

2. Compile the layer. This involves the collection of appropriate agents to interpret the description of the lower layer.

   Line 4 compiles the new layer's interpreter. The compile procedure is described in section 7.7.1 of this chapter. Layer n contains knowledge of the agents that can be used to interpret the description of layer $n + 1$. The description generated in line 3 is compiled into an interpreter program using knowledge of agents that can interpret that description.

3. A new layer object is instantiated with the interpretation resulting from (1) as the description and the interpreter resulting from *compile* in step (2) as the interpreter. The new layer is wired in to the structure with the bi-directional pointers (lowerlayer and higherlayer).

   In line 5, (newLayerConstructor layer) returns the constructor procedure for the subordinate layer.

The adapt method updates the world state knowledge and then recompiles the interpreter for the lower layer.

```
1:(define (adapt ReflectiveLayer|layer gripe)
2:  (withSlots (updateKnowledge) gripe
3:     (updateKnowledge layer))  ;; update the belief state.
4:  (withSlots (lowerlayer) layer
5:     (withSlots (interpreter) lowerlayer
6:       (set! interpreter (compile layer)))))
```

The representation of world state is problem dependent and is not governed by the reflective architecture. In each layer the world state at the corresponding meta level is maintained in the variable "knowledge". When an interpreter causes adaptation with a reflectUp operation an update procedure is loaded into the "gripe" object. Line 3 invokes the update procedure on the layer to cause the world state representation to be updated.

Line 6 recompiles the interpreter for the lower layer. Because the world state has changed the affected interpreter should be compiled differently than when the interpreter was first elaborated.

```
1:(define (execute ReflectiveLayer|layer)
2:  (withSlots (description interpreter knowledge) layer
3:    (run interpreter description knowledge)))
```

7.4 Program Synthesis

The purpose of the "compile" method described in Section 7.3 is to produce a collection of interpreters connected together that performs the function of interpreting the description that belongs to the layer. In this section, we explore the idea of compilation-as-proof, and then use the idea in our self-adaptive architecture.

"Interpreters" were introduced in Chapter 3 and have been used in subsequent chapters. The interpreter represents a coherent computational unit. It includes code for sequencing agents over the input description and making Monte Carlo samples of the agents' results. An interpreter therefore contains a collection of agents that the interpreter controls. In this section we develop the protocol for interpreters that permits them to be automatically selected, sequenced, and populated with agents by the compiler.

7.4.1 Compilation as Proof

The typical compiler can be thought of as the composition of several proof problems: for example parsing, optimizing and producing machine code. The purpose of this discussion is to draw upon our intuitions about the compilation process and not to carefully model the behavior of a compiler; so although such an exercise could be of interest in its own right, we restrict our discussion here to a single level of the compiler.

If we think of the task of the compiler as proving that the program can be computed by the target machine we can see that the resulting machine codes are the axioms of the proof the leaves of the proof tree.

The knowledge in the compiler can therefore be divided simply into two kinds

1. Knowledge of rules of inference and a procedure for applying them in order to arrive at a proof.
2. Knowledge of the relationship between the source code and the target code in the form of rules.

In this view, the compiler produces a tree-structured proof. The leaves of the proof are blocks of machine code. The machine codes are read off the fringe of the proof tree to produce the target machine language representation.

Consider the problem of interpreting a piece of source code as an assembly language program. Models of how to represent a source program fragment in assembly language can be applied in order to interpret the source code as assembly language.

Using the agent/interpreter architecture developed in this thesis, the interpreter would sequence parts of the language over a collection of agents that can deal with the parts of the language. The interpreter part therefore is a code walker and the agents of the interpreter are the collection of agents that deal with each syntactic construct in the language.

Consider the problem of interpreting the expression **C=A+B**.

Compiling that expression involves proving that the expression is a part of the language. The proof looks like this:

1. C=A+B is an assignment (rule: assign-1) where the location is C and the expression is A+B.
2. C is a location (rule: location-1).
3. A+B is an addition (rule: addition-1) where A is a sub-expression and B is a sub-expression.
4. A is a de-reference (rule: dereference-1).
5. B is a de-reference (rule: dereference-1).

The simplest way of implementing this example is to treat it as a one step problem of interpreting the source expression as a sequence of machine instructions. However, in order to use this example to illustrate the components of the reflective architecture we develop this example here as a two step process. For readability and to avoid getting bogged down in unnecessary details we illustrate the essential components using pseudo code. Figure 7.5 shows the relevant portions of the assignment agent and its associated model.

The assignment agent is connected to two agents A1 and A2 of which it asks about the location of the left hand side (LHS) and the value of the expression. It tells its result to agent A3. It fits the model "store-1" which supports an "emit" method that assembles instructions to an instruction stream. Figures 7.6 and 7.7 show implementation templates for addition and de-reference respectively which are required for this example.

The proof tree is shown in Figure 7.8.

Agent: assign-1	
Consumes:	(A1=(location X) A2=(expression Y))
Produces:	(A3=(expression X=Y))
Fit:	(define (fit assign-1|agent) (withTemporaryRegister (R1) (let ((L1 (ask A1 \`(location ,X))) (E1 (ask A2 \`(expression ,Y ,R1)))) (tell A3 (list L1 E1 (make store-1 val: R1 loc: L1))))))
Model: store-1	
	(define (emit store-1|mod output) (withSlots (val loc) mod (assemble output 'STORE val loc)))

Figure 7.5: Agent and Model for Assignment

Agent: addition-1	
Consumes:	(A1=(expression X) A2=(expression Y))
Produces:	(A4=(expression X+Y) A3=(result))
Fit:	(define (fit addition-1|agent) (withTemporaryRegister (R1) (let ((R2 (ask A3 \`(result))) (E1 (ask A1 \`(expression ,X ,R1))) (E2 (ask A2 \`(expression ,Y ,R2)))) (tell A4 (list R2 E1 E2 (make add-1 val1: R1 val2: R2)))]
Model: add-1	
	(define (emit add-1|mod output) (withSlots (val1 val2) mod (assemble output 'ADD val1 val2)))

Figure 7.6: Agent and Model for Addition

The resulting description that results from running the agents is shown below.

```
((location-of C) ; from assign-1
 ((location-of C)) ; from location-1
 ((register tmp1) ; from addition-1
  (deref-1 tmp1 (location-of A)) ; emits (LOAD tmp1 (location-of A))
  (deref-1 tmp2 (location-of B)) ; emits (LOAD tmp2 (location-of B))
  (add-1 tmp1 tmp2))             ; emits (ADD tmp1 tmp2)
 (store-1 tmp1 (location-of C))));  emits (STORE tmp1 (location-of C))
```

Running through the description in sequence applying the "emit" method on each entry results in the following instructions being assembled:

Agent: dereference-1	
Consumes:	(A1=(location X))
Produces:	(A2=(expression X))
Fit:	(define (fit dereference-1|agent) (let ((R1 (ask A1 `(location ,X)))) (tell A2 (list R1 (make load-1 reg: R1 loc: L1)))))
Model: deref-1	
	(define (emit deref-1|mod output) (withSlots (reg loc) mod (assemble output 'LOAD reg loc)))

Figure 7.7: Agent and Model for Dereference

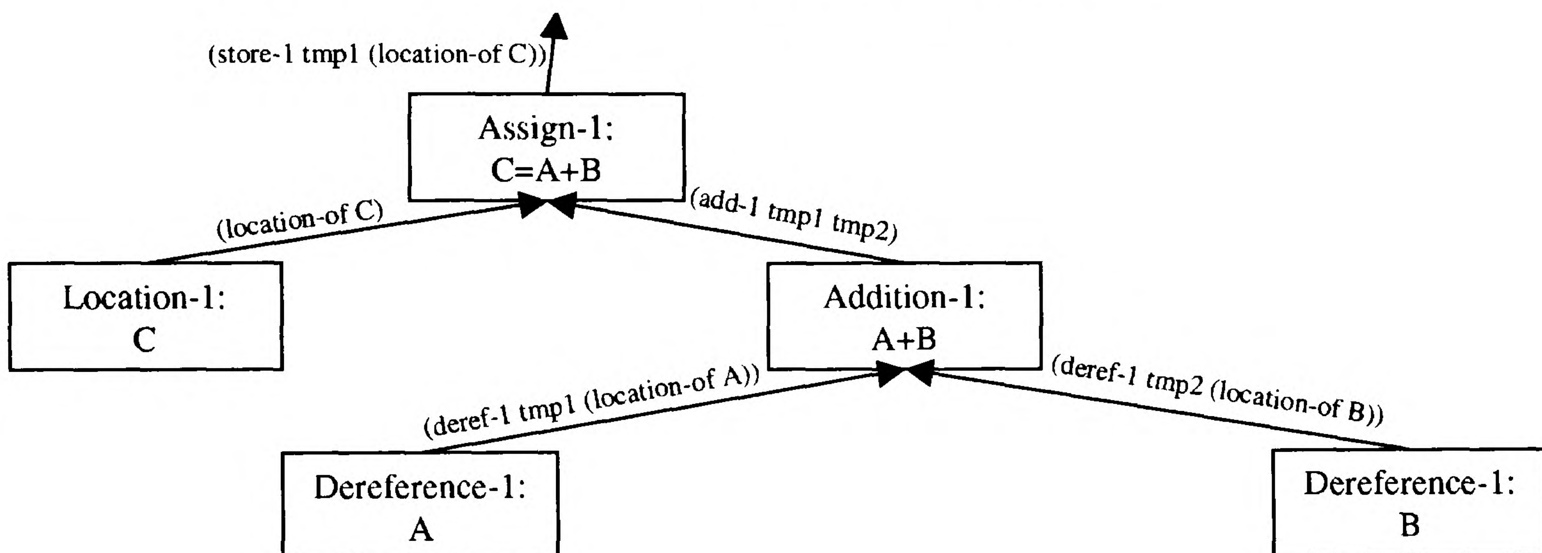


Figure 7.8: Example Proof Tree for Compilation Example

```
(LOAD tmp1 (location-of A))
(LOAD tmp2 (location-of B))
(ADD tmp1 tmp2)
(STORE tmp1 (location-of C))
```

In the example given above we included only a single agent for each operation (assign-1 location-1 addition-1 dereference-1). We could provide arbitrarily many agents for each operation in which case we would have to choose which one to select on the basis of the description length of the description elements.

The number of levels varies from problem to problem. In principle there could be an arbitrarily large number of levels but in practice we expect most problems to have a small number of levels.

The level in the software development example described above that deals with the generation of machine code from a high level language description is of particular interest because our intuition in this domain fits closely with the task of generating code from the corpus.

A compiler is an interpretation program that interprets a high level language source program

and produces a description that draws upon knowledge built in to the compiler about the target machine. Nowhere in the high level source code are the details of the target machine represented. Indeed the code may be compiled with different compilers for different target machines. The compiler embodies various kinds of knowledge essential to producing a good representation of the source:

1. Knowledge of the high level language.
2. Knowledge of certain time and space considerations of certain patterns used in the source program and transformations into more efficient forms.
3. Knowledge of the target machine its instructions, registers, and efficiency considerations.

The compiler may consist of several layers of interpretation problem in which the language is successively translated through intermediate languages until the target level is reached.

The purpose of the above example was to motivate the idea that compilation can usefully be viewed as a theorem proving activity. By adding "produces" and "consumes" to the protocol for agents and interpreters we can treat them as rules of inference and use a theorem prover to connect them up into programs—just as we did above.

In compiling a program into machine code, we generally deal with certainty. The language that is being compiled is not ambiguous and the machine code can be relied upon to perform as expected. Computers are designed to operate reliably and high level languages are designed to be unambiguous.

In the case of image interpretation the source specification may be ambiguous and the rules are not guaranteed to succeed. Instead we have a way of characterizing the likelihood of succeeding. Conventional compilers are a special (deductive) case of a more general (non-deductive) problem.

7.5 Uncertain Information and MDL

The real world doesn't offer us the kind of guarantees that we have managed to build for ourselves in the form of closed world computers. This problem contributes greatly to making programs brittle when they attempt to operate in an unconstrained environment such as the *real world.*

The theorem prover developed below is "relaxed" in that the theorems it produces are guaranteed to be programs that produce the desired effect but only if the program terminates. The program may not terminate because (reflectUp ...) may be invoked prior to completion. Since there may be many agents and interpreters that can be connected up to be a valid program that satisfies the representation from which it was compiled there are many different proofs. We wish to find the compilation/proof that has the highest likelihood of completing without invoking reflectUp.

$$DL(interpreter) = -log_2(1 - P(invokesReflectUp)) \tag{7.1}$$

For each interpreter the description length is determined by Equation 7.1. An individual agent can't invoke reflectUp because in general it won't be the only agent chosen to work on a specific problem. An individual agent may be unsuitable for a particular application and another agent may do better. The interpreter then selects the agent that did best. It is the interpreter object that is in a position to invoke reflectUp, when all chosen agents are unsuitable, and adaptation is required.

We provide three ways for reflectUp to be invoked and cause self-adaptation of the program to occur:

1. An interpreter *pre-test* fails.

2. An interpreter *post-test* fails.

3. No agents are successful in interpreting part of the input that is being interpreted.

We add *pre-test* and *post-test* to the protocol for interpreters along with *produces* and *consumes*.

7.5.1 Protocol for Interpreters

An interpreter is a special kind of computational agent that contains agents which it sequences as described in Chapter 3. To support those activities the interpreters support a protocol for meta-information shown in Figure 7.9.

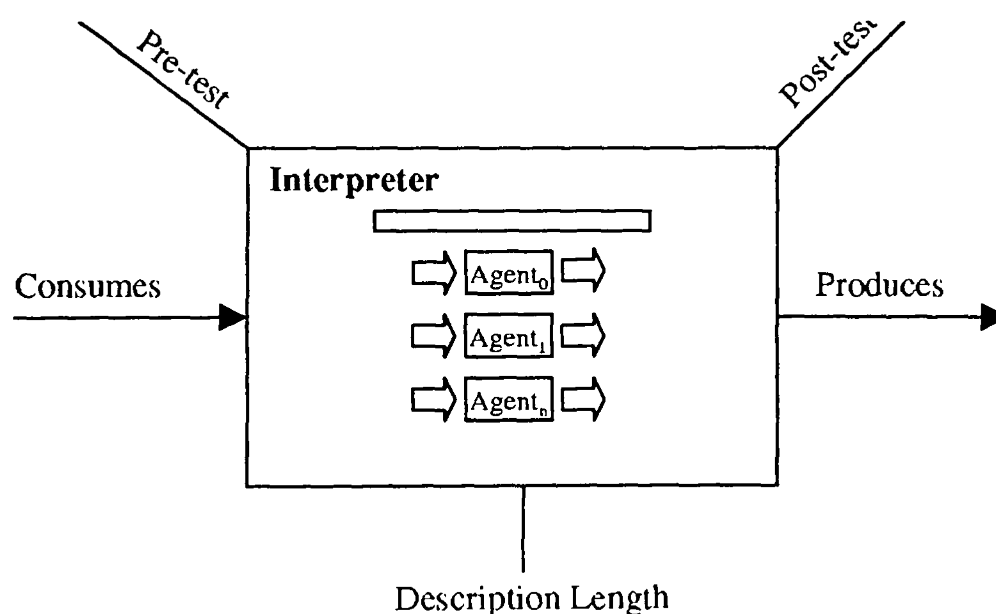


Figure 7.9: Protocol for Interpreter Meta-Information

1. *(pretest anInterpreter anInput)*

   Returns *true* if the input is suitable for the interpreter and *false* otherwise.

2. *(posttest anInterpreter anOutput)*

Returns *true* if the output is acceptable and *false* otherwise.

3. *(descriptionLength anInterpreter anInput)*

Returns the description length of the interpreter. The description length is $-log_2(P(success))$ where $P(success)$ is the probability that the interpreter will successfully interpret the scene.

7.5.2 Protocol for Agents

In order for agents to be selected and connected together by the theorem prover/compiler they must advertise their semantics. The purpose of the compiler is to select appropriate agents and connect them together to form a program. To support those activities the agents support a protocol for meta-information shown in Figure 7.10.

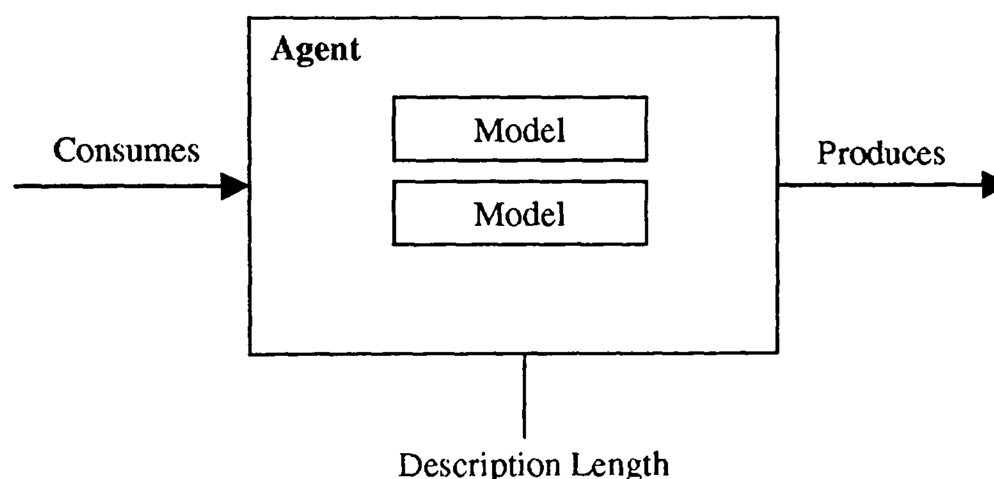


Figure 7.10: Protocol for Agent Meta-Information

1. *(consumes anAgent)*

Returns a list of types that the interpreter expects as input.

2. *(produces anAgent)*

Returns a list of types that the interpreter produces as output.

3. *(descriptionLength anAgent)*

Returns the description length of the agent. The description length is $-log_2(P(correct))$ where $P(correct)$ is the probability that the agent will diagnose the feature in the same way as the specification. The features that appeared in the corpus and were hand annotated are to be diagnosed by agents that have been provided and trained on the corpus. We wish to know how accurately the agent identifies the feature annotated by the expert. We call this measure *Fidelity* as it relates to the fidelity with which an agent mimics the behavior of the

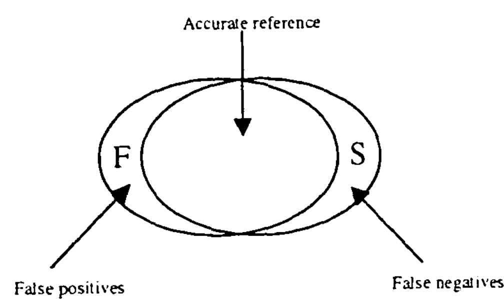
expert. This is the essential part of learning by example in which primitives of the system (the agents) are matched up with the observed behavior of the expert. Maximizing the fidelity maximizes the mutual information between the behavior evident in the observation (the corpus) and the behavior of the agent. The presence of the annotated corpus allows agents to be tested against the data that makes up the corpus in order to compute fidelity values for each agent, as we next discuss.

7.6 Fidelity and Quality

The *fidelity* of an Agent is the probability that it will behave the same way as a human expert as represented by observation over a representative sample (a corpus). If, out of 100 samples, the agent produced the same diagnosis as the human expert (both positive and negative examples) 90 times then its fidelity is 0.9.

Given an observation **S** and a collection of agents that generate diagnoses that correlate well with **S**, the agent selection problem can be construed as finding the agent that produces diagnoses that have the closest correlation with **S** as understood from evidence collected during the training of the agents. If we know the behavior **S**, then we can describe the behavior of the agent in terms of its deviation from **S**. If the Fidelity of an agent is 1.0, the agent always performs the same as the observation and from the standpoint of knowing the correct behavior it is completely predictable. The description length of an agent is:

$$DL(agent) = -log_2(Fidelity(agent, S)) \tag{7.2}$$



S: The set of designations that belong to the rigid kind S.
F: The set of designations produced by agent

Figure 7.11: Fidelity

The situation is depicted in Figure 7.11 for a set of observations **S** and the performance of the trained agent **F**. From the standpoint of agent selection, we wish to select the agent that maximizes the ratio of $F \cap S$ and $F \cup S$.

$$Fidelity(F, S) = \frac{P(F \cap S)}{P(F \cup S)} = \frac{P(F|S)P(S)}{P(F) + P(S) - P(F|S)P(S)} \tag{7.3}$$

This contains prior probabilities that are unavailable but it can be rewritten as:

$$Fidelity(F, S) = \frac{P(F|S)P(S|F)}{P(F|S) + P(S|F) - P(F|S)P(S|F)} \tag{7.4}$$

The magnitude of fidelity for all agents over a given observation are comparable because the magnitude is based on S. Consequently, all agents supporting S can be compared and thus Fidelity is a suitable measure for making agent selections.

Fidelity is a measure of information retained when an observation is represented as a primitive agent operation. Fidelity can be approximated using conditional probabilities that can be acquired during training sessions.

$$Fidelity(F, S) = \frac{P(F \cap S)}{P(F \cup S)} = \frac{I(F' \wedge S')}{h(F', S')} \tag{7.5}$$

S' and F' being random variables corresponding to the events S and F respectively. This correspondence is due to Hu (Hu 1962). This highlights the relationship with mutual information, and the probabilistic construction gives us an algorithm for computing it.

7.6.1 Quality

An agent will often have some basis for assessing its confidence in the particular computation it has performed. In other cases, no such information is available. So far, only cases in which the agent has provided no indication of quality or confidence in their result's accuracy have been dealt with.

The way that an agent arrives at a quality measure is subject to a wide range of possibilities no less limited than the ways that agents themselves are defined and implemented. For example, an agent that recognizes an edge might scale the magnitude produced by a gradient operator and use it as a quality measure - good quality edges yielding high values from the gradient operator; a tank recognizer might combine evidence from a variety of features, and by combining such evidence (such as by Dempster-Shafer (Shafer 1976), or using a Bayesian belief network (Pearl 1988; Mann & Binford 1994)) to arrive at a level of belief that a tank has been recognized.

Any general framework for incorporating quality assessments from a diverse collection of agents must address the issue of quality normalization. An agent's belief in the quality of its produced diagnoses may or may not be accurate. We first state several attributes that any quality normalization process should have.

1. Intuitively, quality information that bears no discernable relationship with observed behavior should be treated identically to cases in which no quality information is provided. Also, an agent that provides some usable quality information should be preferred over an identical agent that fails to provide a quality assessment.

2. Given two agents—identical in every way except that agent A doesn't provide quality information whereas agent B does—agent B should be chosen over agent A unless the measure of quality computed by agent B provides no useful information.

3. There should be no quality information that can *reduce* the value of an agent since the quality measure can always be ignored.

4. It should be possible to compare arbitrary agents A and B when either or both agents provides quality information.

The quality of two agents A and B can be compared without regard to belief functions, using $Fidelity(S, X)$ where a perfect agent yields 1 and an agent that is completely incompatible with the observation (S) yields 0. Fidelity can be extended to include belief in the feature (Bel).

If the quality value Bel is positively correlated with the likelihood that the feature has been identified correctly, the following will hold:

$$A, B \in \mathcal{N}, S \in \mathcal{R} : Bel(A) > Bel(B) \Rightarrow P(S|A) > P(S|B) \tag{7.6}$$

Usually, this will be the case, however negatively correlated belief functions should also work:

$$A, B \in \mathcal{N}, S \in \mathcal{R} : Bel(A) > Bel(B) \Rightarrow P(S|A) < P(S|B) \tag{7.7}$$

Regardless of positive or negative correlation, any useful information about quality should be usable. By correlating evidence about the relationship between a belief function and the likelihood of a good match, the combination of a feature and its belief can be mapped onto a probability assignment $P(S|A)$. Using this value, it is possible to compare agents as before using the goodness value, *Fidelity(A, S)*.

We are not just interested in the correlation of Belief with $P(S|A)$, since a high correlation is of little use if a high value of belief is never (or rarely) generated. We need to have some understanding of the distribution of beliefs as well as the correlation.

If the belief function assigns one of n belief values, such an agent can be thought of as a collection of n agents that each deliver a fixed belief value for the same feature. For an observation S and an agent A, the n agents are known to be disjoint, so

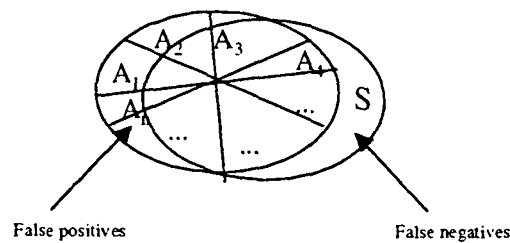
$$P(A|S) = \sum_{i=1}^{n} P(A_i|S) \tag{7.8}$$

where $A_i = A \cap Belief(A) = i$.

The fidelity of each of the n sub-agents can be computed using

$$Fidelity(A_i, S) = \frac{P(A_i|S)P(S|A_i)}{P(A_i|S) + P(S|A_i) - P(A_i|S)P(S|A_i)} \tag{7.9}$$

The combined fidelity can be computed as the weighted mean of the sub-agents fidelity:



S: The set of designations that belong to the rigid kind S.
A_1 - A_n: Designations produced by agent, by belief

Figure 7.12: Division of a Kind into Separate Beliefs

$$Fidelity(A, S) = \frac{\sum_{i=1}^{n} Fidelity(A_i, S) P(S|A_i)}{\sum_{i=1}^{n} P(S|A_i)} \tag{7.10}$$

7.7 Agent Training and Fidelity

To demonstrate the training of agents described above, a prototype was written in Lisp that implements a training engine to detect vertical lines. Three sets of feature detectors are defined below in versions 1, 2 and 3 that allows observations to drive a training session. The task is to present a vertical line (90 degrees) and to measure the Fidelity of the kinds supported by the line recognizer agents. In all cases, due to the latitude of the recognizers, both the 80 degree and the 100 degree recognizers are found to have similar fidelity to the vertical line observation. A test data generator that generates lines at random angles is used to train the VERTICALLINE observation. The probabilities are computed by the method described in the implementation section. Three test cases are demonstrated:

1. Version 1. No belief assignment is provided. Demonstrates that the basic computation of conditional probabilities is working correctly and that Fidelity is computed correctly.
2. Version 2. A belief assignment is provided and the belief assignment does contain information. Demonstrates that the Fidelity measure correctly reflects the information in the belief.
3. Version 3. Demonstrates the intuition that when a belief is provided that contains no useful information, the Fidelity is the same as the case where no belief is provided.

Algorithms and data structures

Figure 7.13 shows the data structures for two values. The first is the observed value VERTICALLINE and the second is the agent computed value *Version1-line80-kind*. This example shows the simple case of values without representations of belief. Whenever an example from the training data is presented, the event count in the VERTICALLINE table of events (left) is

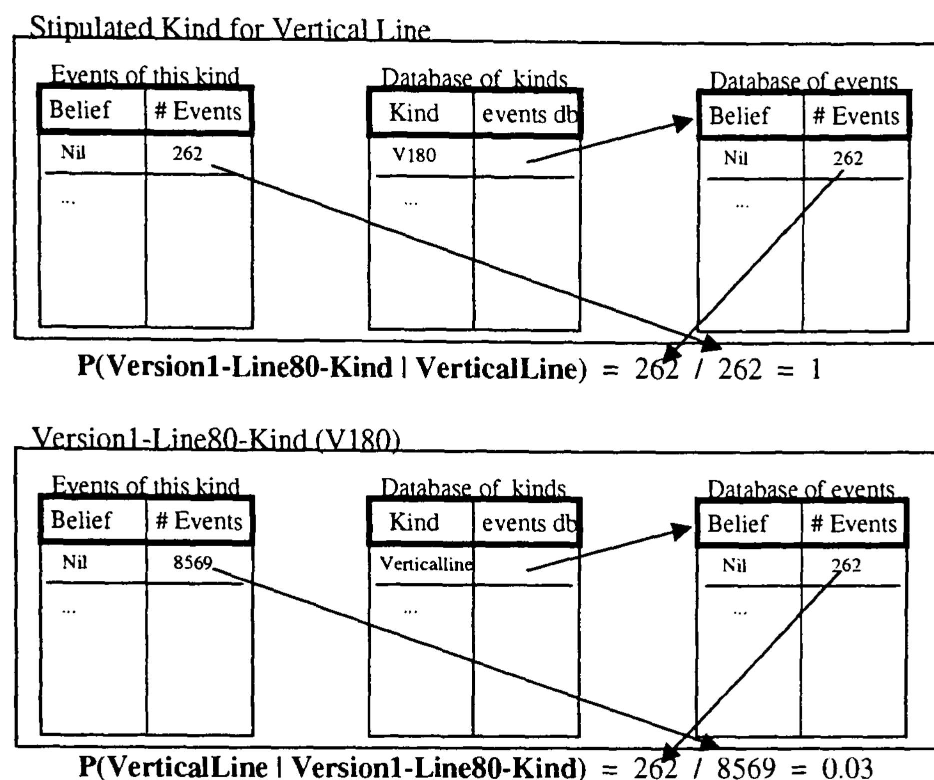


Figure 7.13: Data Structures for Computing Conditional Probabilities

incremented. There have been 262 examples of a presented vertical line in the training data according to this figure. Whenever an agent produces a value, the event count for the kind of the value is incremented. There have been 8569 such events so far in training. If an agent computed value and a presented value are active at the same time, the database of events are updated for the appropriate belief state. A value of 262 such events is recorded for both kinds in this example. The data is updated as training proceeds, and the conditional probabilities can be read out as indicated on the figure. Only data between kinds that have a non-null conjunction are stored.

Version 1

Given an image containing line segments, there are nine feature detectors that detect lines of a fixed angle (0, 20, 40, 60, 80, 100, 120, 140, 160 degrees). The feature detectors detect lines falling within a range of values. For example, the feature detector that detects lines at angle 80 degrees, will detect lines at angles between 65 degrees and 95 degrees (set point $\pm$ 15 degrees).

Version 2

In version 2 of the test, the version 1 tools are left in the toolbox and another 9 recognizers are added. The version 2 recognizers are identical to the version 1 recognizers except that a belief in the quality of the line is given. Quality is defined in terms of closeness to the set point by specifying a belief between 0 and 100. 0 indicates angle plus or minus 15 degrees. 100 indicates perfect alignment.

Version 3

In version 3 another 9 recognizers are added to the nine version 1 recognizers. The version 3 recognizers are identical to the version 2 recognizers except that the belief in the quality of the line is just a random number between 0 and 100. Since the quality estimate contains no useful information, the Fidelity of the version 3 recognizers should be the same as the version 1 recognizers.

Below is the terminal output for a test session in which version 1, version 2 and version 3 tools are all run together over the same data (This run happened to be called experiment 4). The program was run over 50000 randomly generated lines.

```
(experiment4 50000)
Total of 27 tools in database Experiment 4

  lineVersion1-160-tool lineVersion2-160-tool lineVersion3-160-tool
  lineVersion1-140-tool lineVersion2-140-tool lineVersion3-140-tool
  lineVersion1-120-tool lineVersion2-120-tool lineVersion3-120-tool
  lineVersion1-100-tool lineVersion2-100-tool lineVersion3-100-tool
  lineVersion1-80-tool  lineVersion2-80-tool  lineVersion3-80-tool
  lineVersion1-60-tool  lineVersion2-60-tool  lineVersion3-60-tool
  lineVersion1-40-tool  lineVersion2-40-tool  lineVersion3-40-tool
  lineVersion1-20-tool  lineVersion2-20-tool  lineVersion3-20-tool
  lineVersion1-0-tool   lineVersion2-0-tool   lineVersion3-0-tool

Total of 1 kinds in database Experiment 4
  PRESENTEDKIND: VERTICALLINE approximated by 6 computed kinds
    COMPUTEDKIND lineVersion1-80-kind
      P(lineVersion1-80-kind|VERTICALLINE)=         1.0
      P(VERTICALLINE|lineVersion1-80-kind)=         0.0289702087731644
      Fidelity(VERTICALLINE, lineVersion1-80-kind)= 0.0289702087731644
    COMPUTEDKIND lineVersion2-80-kind
      P(lineVersion2-80-kind|VERTICALLINE)=         1.0
      P(VERTICALLINE|lineVersion2-80-kind)=         0.0289702087731644
      Fidelity(VERTICALLINE, lineVersion2-80-kind)= 0.47047619047619
    COMPUTEDKIND lineVersion3-80-kind
      P(lineVersion3-80-kind|VERTICALLINE)=         1.0
      P(VERTICALLINE|lineVersion3-80-kind)=         0.0289702087731644
      Fidelity(VERTICALLINE, lineVersion3-80-kind)= 0.0313345962367738
    COMPUTEDKIND lineVersion1-100-kind
      P(lineVersion1-100-kind|VERTICALLINE)=         1.0
      P(VERTICALLINE|lineVersion1-100-kind)=         0.0290451552210724
      Fidelity(VERTICALLINE, lineVersion1-100-kind)=0.0290451552210724
    COMPUTEDKIND lineVersion2-100-kind
      P(lineVersion2-100-kind|VERTICALLINE)=         1.0
      P(VERTICALLINE|lineVersion2-100-kind)=         0.0290451552210724
      Fidelity(VERTICALLINE, lineVersion2-100-kind)=0.457407407407407
    COMPUTEDKIND lineVersion3-100-kind
      P(lineVersion3-100-kind|VERTICALLINE)=         1.0
      P(VERTICALLINE|lineVersion3-100-kind)=         0.0290451552210724
      Fidelity(VERTICALLINE, lineVersion3-100-kind)=0.0325063292981336
```

Analysis of the version 1 tools data

The results are as expected. $P(lineVersion180kind|VERTICALLINE) = 1.0$ because whenever there is a vertical line, the recognizer with set point = 80 recognizes it. $P(VERTICALLINE|lineVersion180kind) = 0.0318033550792171$ because the recognizer accepts lines ranging from 65 to 95 degrees (30), only one of which is a vertical line. The value of $Fidelity(VERTICALLINE, lineVersion180kind)$ is the same as $P(VERTICALLINE|lineVersion180kind)$ because $P(lineVersion180kind|VERTICALLINE) = 1.0$.

$$Fidelity(F,S) = \frac{P(F|S)P(S|F)}{P(F|S) + P(S|F) - P(F|S)P(S|F)} = P(S|F) \text{ if } P(F|S) = 1. \qquad (7.11)$$

Analysis of the version 2 tools data

This example peaks out at 33 where the 80 recognizer is recognizing 70 or 90, and the 100 operator is recognizing 90 or 110. As expected, the version 2 recognizer is significantly better than the version 1 recognizer. The fidelity jumps to around 0.5. This is because all PRESENTED lines occur when the belief value is 33, but because this state also recognizes lines of angle 70 degrees (in the case of the 80 degree recognizer), and 70 and 90 are equally likely, the Fidelity is 0.5. Clearly the version 2 recognizers will be selected over the version 1 recognizers because of the quality estimate that is provided. The Fidelity improves because the quality measure contains information about the quality even though the correlation between the quality value and the desired line angle is not straightforward (i.e.: it peaks at 33, not at 0 or 100).

Figure 7.14 Shows the results for a run of experiment 4 showing Fidelity values for computed kinds from the three versions. As expected, only the 80 and 100 kinds have non-zero fidelity. 80 and 100 kinds are approximately the same. Version 2 tools are dramatically superior due to the information in the belief. Version 3 tools have no information in the belief since they are randomly computed. As a result they are approximately the same as the version 1 tools that don't have a belief computation.

It is interesting to note a slightly higher fidelity recorded for the version 3 tools than the version 1 tools. It could easily be mistaken for an effect of randomness in the data, but in fact it is a side effect of the way that beliefs are mapped. Since separate conditional probabilities are kept for the different discrete belief values (101 in this case: 0 through 100), more trials are required in order to reach the same confidence in the result. The estimate is initially optimistic since it is triggered by a positive occurrence. As more data is collected, it asymptotically approximates the correct value.

Figure 7.15 shows the convergence of fidelity at fixed intervals over the training period.

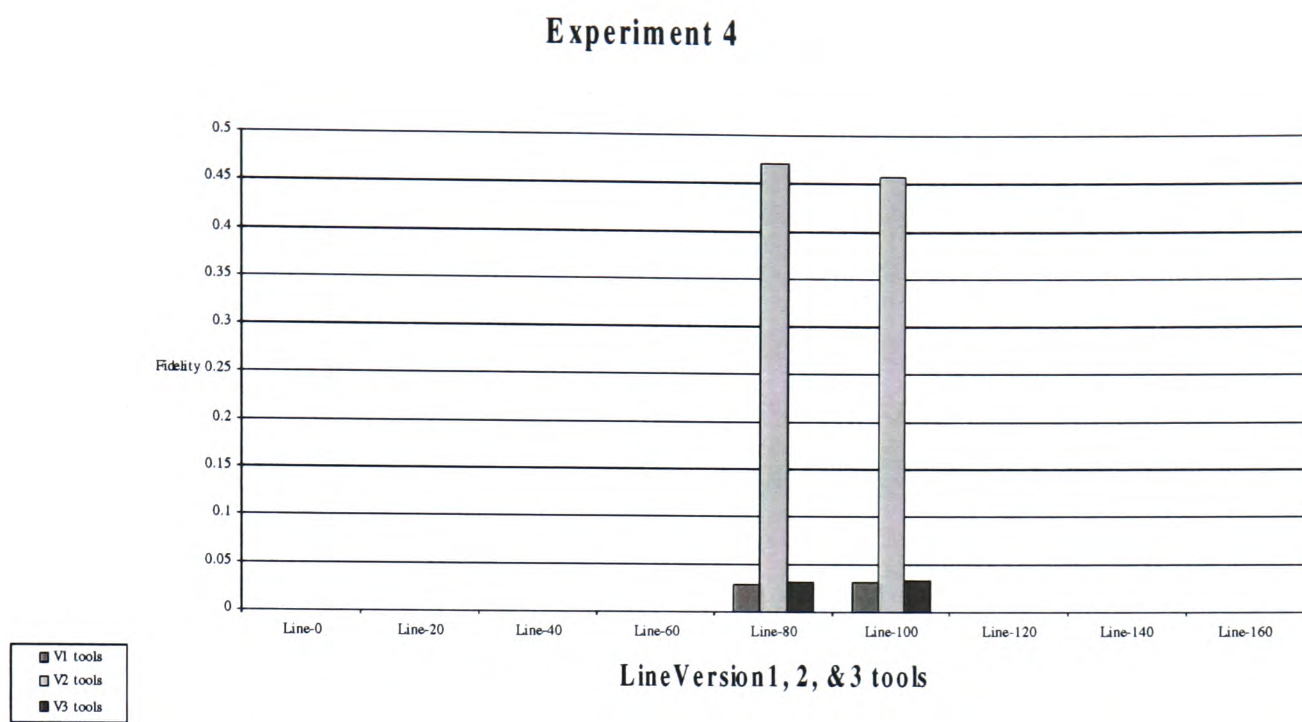


Figure 7.14: Fidelity Results for Version 1, 2 and 3 Tools

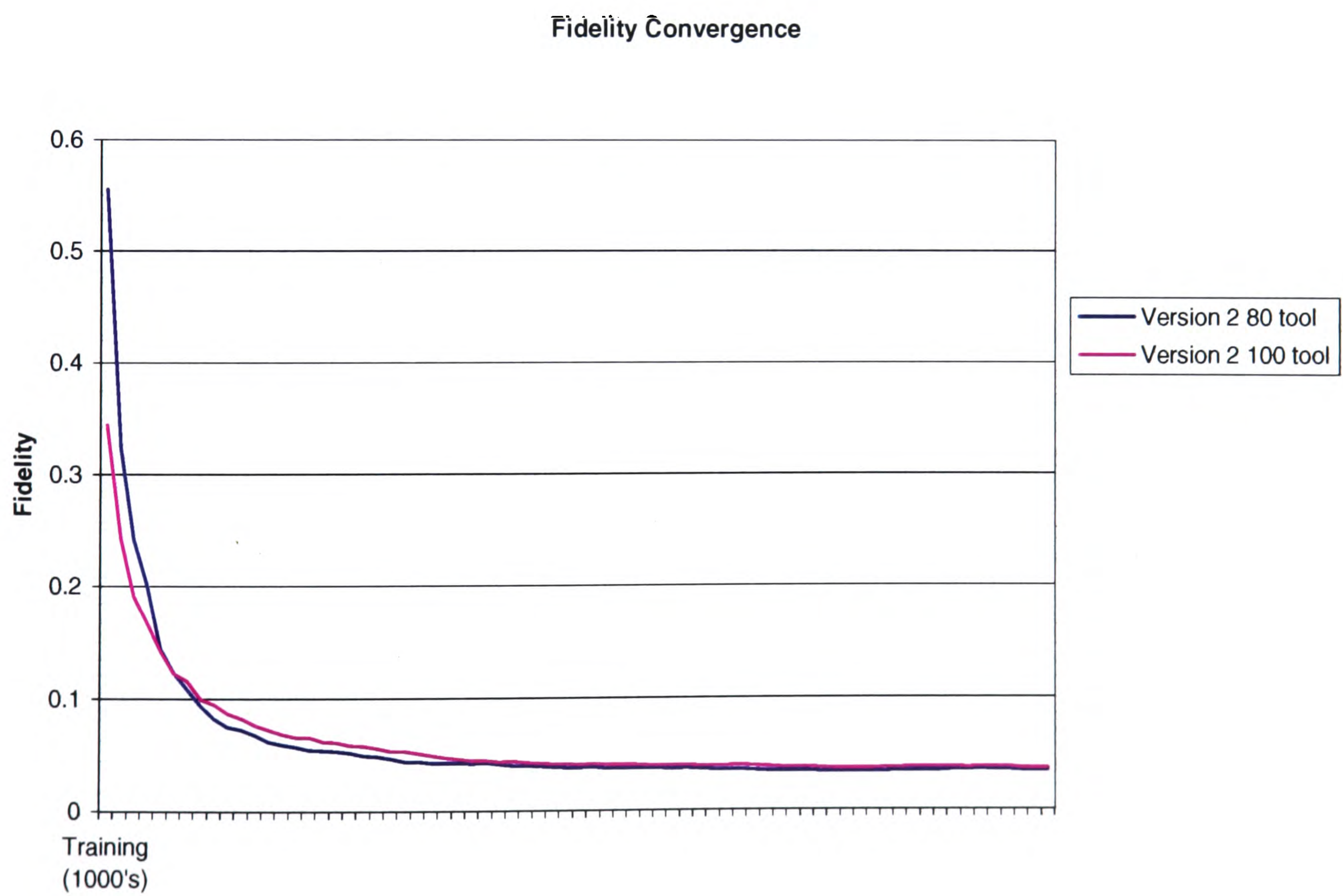


Figure 7.15: Fidelity Convergence

7.7.1 MDL statistical theorem prover for code generation

The purpose of the theorem prover/compiler is to prove that the input representation is a theorem by treating the interpreters that interpret it as rules of inference. If the antecedents of a rule are satisfied, the rule is true—meaning that it will successfully interpret its input. In practice however an interpreter may fail to interpret its input even though its antecedents are true. This is not because the rule is false but because the input does not belong to the context for which the interpreter is designed. When an interpreter detects that it is not matched to its input it performs a reflectUp operation in order to self-adapt to be better suited to the input.

Since there are often multiple possible proofs, and since the proofs are not all equally good, we wish to find the proof that has the greatest likelihood of successfully interpreting its input without having to self-adapt. The proof/program that is most likely to succeed without adapting is the one with the global minimum description length.

An interpreter with *produces* and *consumes* is used as a rule of inference with *consequent* and *antecedent* respectively. A proof consists of a set of rules (interpreters) necessary to prove the parts of the theorem. Each rule may have one or more rules that satisfy its antecedents.

The description length associated with a rule assumes that the rule's antecedents are true.

$$-log_2 P(rule_n = success|antecedents) \tag{7.12}$$

The probability that the entire program (the proof) will complete successfully (without calling reflectUp) therefore is:

$$P(rule_n = success|antecedents)P(antecedents) \tag{7.13}$$

If T is the proof tree the description length for the entire proof is:

$$\begin{aligned} DL(T) = -\, log_2 \prod_{rule_i \in T} P(rule_i) \\ \sum_{rule_i \in T} -log_2 P(rule_i) \\ \sum_{rule_i \in T} DL(rule_i) \end{aligned} \tag{7.14}$$

The algorithm for proving a theorem is structurally similar to the parser described in Chapter 5.

Implementing the Theorem Prover/Compiler as GRAVA Agents

The algorithm picks rules randomly in proportion to the probability of those rules. When a complete proof has been selected at random in this way it is not guaranteed to be the most probable proof; but if the sequence is repeated often enough, the most probable proof will be

generated more often than less probable proofs. In this way the most probable proof can be approximated to an arbitrary accuracy by repeatedly generating more legal proofs.

The Monte-Carlo theorem prover seeks a correct proof with a depth first search. At each branch point in the search it chooses randomly and finally returns the first proof that succeeds. Because the random choices are weighted by the probabilities of the rules, more likely proofs occur more frequently than less probable proofs. Every time the code is run, a new sample (legal proof) is computed and saved. The best current proof is then selected and returned.

We implement the theorem prover using GRAVA agents. Each interpreter is instantiated as an agent. Each interpreter is implemented as a separate model. The structure of the Proof model is shown below.

```
;;; Model for proof rule of inference.
(defineClass Proof (Model)
  ((interpreterName) ; Interpreter name
   (antecedents) ; Antecedents of the rule.
   (descriptionLength)) ; -log2(P(this rule|antecedents))
  documentation "Model for a rule of inference")
```

The result of a successful application is a proof tree structure:

```
(defineClass ProofTree (DescriptionElement)
  (...(rule)   (antecedents)))
```

The theorem prover agent implements the "fit" method by fitting a rule of inference to the unproven form.

```
(defineClass StatisticalTheoremProver (Agent)  ...)

(define (fit StatisticalTheoremProver|ag data)
  ;; Described below
  ...)
```

The StatisticalTheoremProver agent is part of the TheoremProverInterpreter:

```
(defineClass TheoremProverInterpreter (Interpreter)
  ((proofStateList) ...))
```

The original theorem to be proved is a conjunction represented as a list of conjunctive forms $(C_1 \ldots C_n)$. The interpreter applies the theorem prover agents to the unproven forms and resamples as necessary. The interpreter manages the proof state list which is initialized to:

$$proofStateList \leftarrow (<(), (C_1 \ldots C_n), 0.0>) \qquad (7.15)$$

The interpreter maintains the partial proof state and the binding list and sequences the components of the theorem through the agents that make up the theorem prover interpreter.

The interpreter mediates which agent's result will be used in the proof by using "Monte-Carlo-Select". Each agent is controlled by the interpreter to perform its "fit" operation. The agents that satisfactorily "fit" their model to the data announce their result and description length and the Monte Carlo select algorithm selects which agents contribution will be selected.

The StatisticalTheoremProver agent's "fit" method does the following when applied to a form.

If the form matches the rule's "produces" part, the rule, form, binding list, partial proof, and the description lengths are gathered up into a structure and returned from the rule matching procedure. Monte-Carlo-Select selects one of the agents contributions, updates the proofStateList and continues to cycle through the forms and rules until the proof is complete.

Antecedents are appended to the partial proof list. If a form on the partial proof list has already been proven a pointer to the original proof is used instead of proving it again. Because of this the proofs are actually directed acyclic graphs (DAGs) rather than strict trees.

The interpreter maintains the "best" proof and updates it whenever a sample produces a proof with a smaller description length.

7.7.2 Disjunction and Teams

The needs of an agent are implicitly a logical conjunction. For example the agent that fits a grammar rule that requires a "town" and a "river" requires as an antecedent (consumes) river AND town. Disjunctions occur when there are multiple agents/models capable of producing the same interpretation. For example there may be several models that produce the label "town". If there are three visually different kinds of region annotated as "town" in the corpus there will be three distinct models of "town" (see chapter 6). Agent/model combinations that belong to the same context can work together as a "team" to determine whether a region is a "town" or not. Agents that belong to different contexts are incompatible and require a context switch. Representing a collection of agents is less specific than representing a single agent and so in general should require fewer bits. The MDL proof of a rule that consumes river and town then will be the one in which all contributors to river and all contributors to town within the same context are included. If the problem was resource constrained we would have to select which contributors to choose based on their expected utility but since we have not considered resource constraints it is always better to include all compatible (same context) contributors.

7.8 Conclusion

The architecture described in this chapter has the novel attribute that it knows what it knows and it knows how to apply what it knows. As with Smith's (Smith 1984) reflective architecture each reflective layer implements the layer beneath it. In Smith's architecture each level of interpreter implemented the level of interpreter beneath it. In the GRAVA architecture each layer implements

not only the interpreter but also the program beneath it. The inclusion of a compiler in the reflection loop allows the meta-interpreter to respond to the changing state of the environment by recompiling lower layers as appropriate. Each layer contains the knowledge of:

1. What the layer beneath is trying to do.
2. What the current belief state is as to the state of the world as it pertains to the operation of the layer beneath it.

When the belief state changes at some level the code for lower levels is automatically updated. If new agents are added they are automatically used where appropriate.

By using contexts automatically generated by the clustering algorithm developed in Chapter 6 we are able to automatically generate the models for the agents based on evidence *grounded* in the annotated images. Given the automatically generated contexts we are even able to automatically generate pre-tests for the interpreters.

The use of an annotated corpus to provide both contexts and models provides both challenges and opportunities. Even with our simple system the number of models produced is huge. The number of contexts produced is less huge because the training sets were fairly homogeneous. As the size of the system becomes large our ability to debug the resulting programs becomes poor. Since the system is self-maintaining there should of course be no need to debug the system but while the system is being actively developed and changed it becomes hard to find subtle bugs. On the bright side, when the system is working correctly, it is simultaneously possible to represent the complexity of the real world adequately for the purpose of generating reasonable interpretations and not necessary to understand the program complexity that usually makes maintaining such systems a nightmare.

The GRAVA architecture in total consists of the protocol developed in chapter 3 for the MDL agent architecture and the reflective self-adaptive protocol developed in this chapter. The MDL agent architecture developed in Chapter 3 has been used as the implementation basis for all subsequent chapters. The MDL approach guarantees that the interpretation that results will be an approximation to the most probable one.

Chapter 8

Self-Adaptive Aerial Image Interpretation

8.1 Introduction

In this chapter we develop the self-adaptive image interpretation program for the problem domain using the results from Chapter 6 as the specification of program behavior and using the self-adaptive architecture developed in Chapter 7.

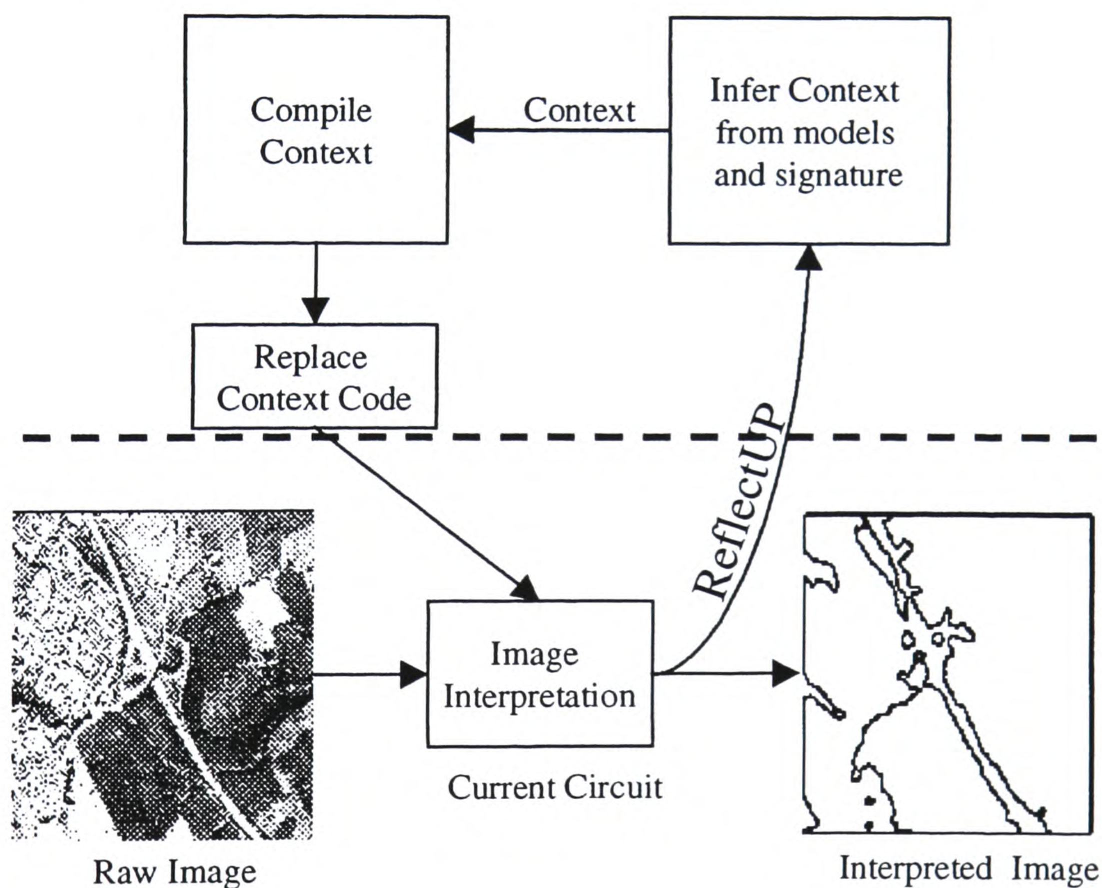


Figure 8.1: Conceptual Schematic of the Self-Adaptive Loop

Figure 8.1 shows a conceptual overview of the self-adaptive process as it pertains to the aerial image interpretation problem domain.

The part above the dashed line represents the meta-level that consists of a collection of agents that compile the specification into the program that is running below the dashed line.

When the image interpretation program detects that the running program is not well matched to the image under consideration the program reflects up to the meta level (above the line) where the new knowledge about the world can be used to: update the world model, select a

more appropriate context, and recompile the portion of the program that corresponds to the new context.

Having developed the reflective architecture (described in Section 7.3) and the program synthesis capabilities described in Section 7.4, all of the components of the thesis are brought together to build interpretations of images from the aerial corpus. This brings together the segmenter developed in Chapter 4, the parser developed in Chapter 5, contexts from Chapter 6, and the self-adaptive architecture developed in Chapter 7.

8.1.1 Overview of the Chapter

In Section 8.4 we describe the overall design of the domain problem demonstration program. We describe how the models of the corpus developed in Chapter 6 are interpreted as specifications for a program that computes interpretations similar to those of the human expert. In Section 8.3 we show some results of using the self-adaptive architecture on the problem domain of aerial image understanding.

8.2 Aerial Interpretation Program Design

The self-adaptive architecture described in Chapter 7 does not necessarily have to use knowledge generated from a corpus. In the demonstration problem domain, however, we have defined the problem as one of mimicking a human expert and we have extracted certain information about the performance of the human photo interpreter from the annotated corpus. The behavior of the human expert is contained in the annotated corpus. The information contained in the annotated corpus is represented in the form of statistical models using the PCD algorithm developed in Chapter 6. The statistical models represent a *specification* of the behavior that our program should produce.

8.2.1 Reflective Layers

There are two complete layers in the problem domain. The top layer interprets the corpus in order to build a description of the corpus and acts as an interpreter for the models of the corpus. The next layer interprets the image.

Figure 8.2 schematically shows the layers of the domain example. The descriptions (boxes D2, D1, and D0) are, from top to bottom, the annotated corpus, the models of expert behavior, and the final interpretation. Each one is an interpretation of the previous in the context of the world knowledge at that layer.

Below, we describe the flow of control from level 3 to level 0. Level 3 is the top of the tower of interpreters and contains nothing but the debugger. Any *reflectUp* operation executed from level 2 will fall into the debugger (unless it is handled by an exception handler in level 3).

Level 2 begins with the annotated corpus (D2). The top level interpreter (I2) is a hand coded collection of agents that interpret the corpus as models. We gave an outline of I2 in

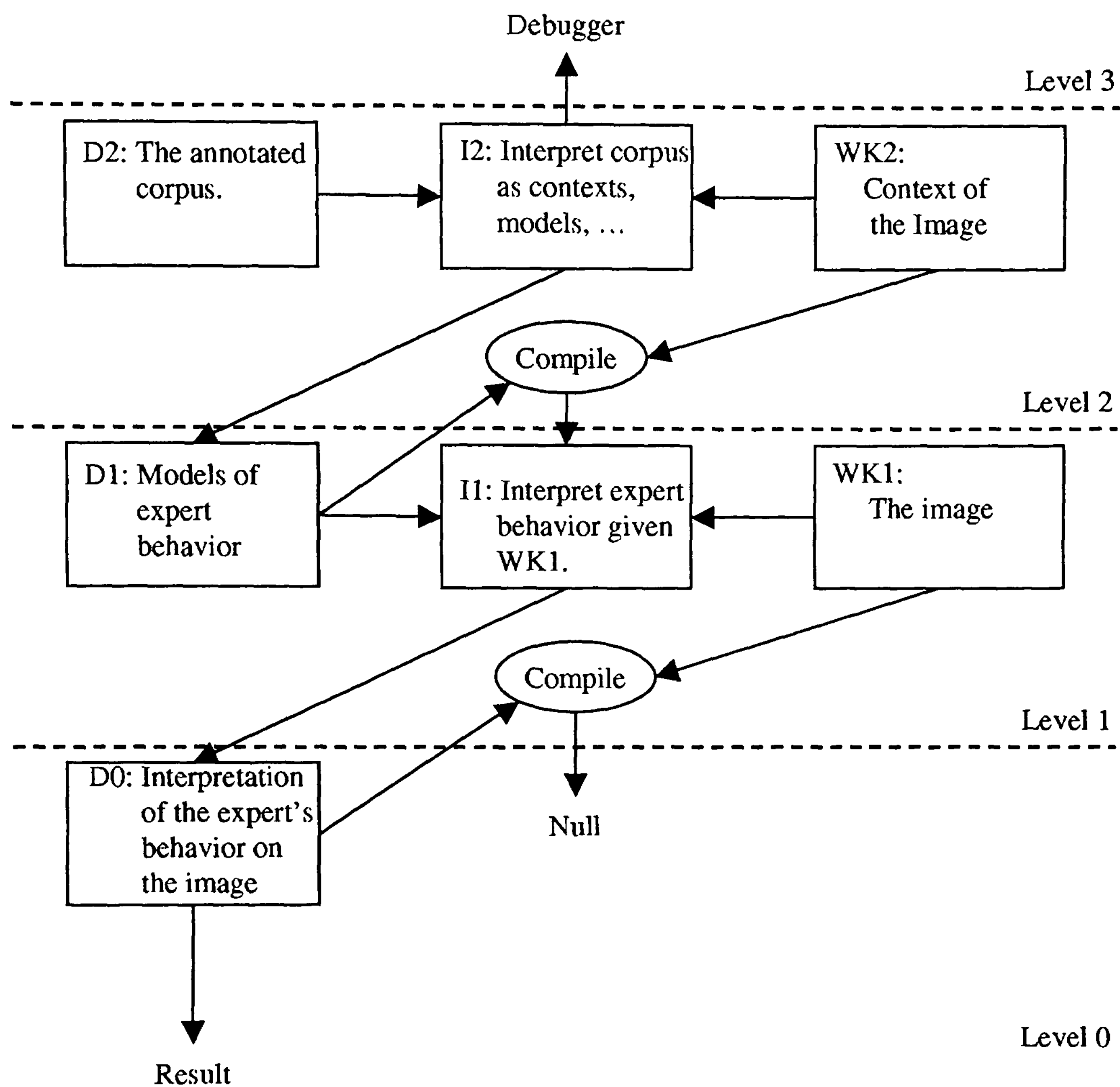


Figure 8.2: Reflective Layers of the Domain Example

chapter 6. It collects statistical models from the data points that constitute the corpus. These include descriptions of the region's contents to initialize seed point agents for the segmenter, chain-link outline models for the segmenter, cluster models of label interpretation, parse rule sets learned from the annotated corpus, and cluster models of images as optical contexts and labeling contexts.

If level 3 existed, I2 would have been generated from level 3; but because it is the top level, the corpus interpretation code is hand implemented. WK2 is the representation of the world knowledge which we discuss below in section 8.2.3. WK2 is what the program knows about the circumstances surrounding the formation of the image which is being interpreted. The image itself is what is known about the world at level 1. WK2 is the program's "belief" about the state of the world as it pertains to the image formation process. The interpreter I2 is a function that takes the description D2 and the world knowledge WK2 as its two arguments (see line 3 of the "execute" method in the meta interpreter description in Chapter 7).

Level 2 is responsible for populating two aspects of level 1: the description, and the inter-

preter. These are generated by the interpreter (I2) and the compiler respectively.

$$\begin{aligned} D1 &= I2(D2, WK2) \\ I1 &= Level2Compile(D1, WK2) \end{aligned} \tag{8.1}$$

The compiler at level 2 finds agents and interpreter objects that can produce the models in D1 in the state of the world which is described in WK2. The resulting interpreter I1 contains all agents that can assist in interpreting the model D1 of the expert behavior. When I1 is executed it will attempt to mimic the behavior of the human expert that produced the annotations.

$$\begin{aligned} D0 &= I1(D1, WK1) \\ I0 &= Level1Compile(D0, WK1) \end{aligned} \tag{8.2}$$

As can be seen from equation-list 8.2 the description D0 is, depending on your point of view, a description of the expert behavior given the image or a description of the image given the behavior of the expert. Because level 1 is the lowest layer defined in the program the compiler doesn't attempt to compile D0 but returns "null" immediately which signals to the self-adaptive meta-interpreter that the computation has terminated and returns D0 as the end result. The knowledge of the world at level 1 (WK1) is simply the image that is being interpreted.

8.2.2 Optical and Labeling Contexts

Our test problem domain works by treating the contexts that we learned from the corpus in Chapter 6 as the contexts that the program adapts between. There are two kinds of context in our data: optical contexts, and labeling contexts. If the corpus had been larger we could support more contexts such as a language context for the parse rules. Because the contexts were quite small (many contexts had only a single representative image) the effect of subsetting the parse rules to those found in a particular context would be insufficiently representative to generate good parses of new images. The size of the corpus limits us to two kinds of context.

8.2.3 Models of World State

The agents in I2 that interpret the corpus construct two context models as described in Chapter 6 for optical contexts and labeling contexts. All optical and labeling models are associated with a context. Only rules that match the current context are matched by the compiler. Two other things are generated from the contexts: the world state (WK2), and pretests for the context which are described below.

The original idea for the problem domain was to have image sequences in which the world state knowledge (WK2) was to have retained state information between frames of the sequence, so if the context in frame n was (say) $C(1,7)C(2,9)$ the frame $n+1$ would start out assuming the same context. For image sequences it is usually the case that contexts persist between frames,

only occasionally changing as the scenery changes. Since we goal reduced the demonstration problem to dealing with single frames, the world state consists of, for each context category (i.e. optical or labeling) a list of contexts along with their probability. Each list is sorted and the head of the list (most probable) is initialized as the current world state. The program begins with the assumption that the image will come from the most likely contexts.

When a pretest fails and a reflectUp operation occurs, the world knowledge is updated by removing the first (current) context from the list so that the next most probable context is selected. The lower level is then recompiled and the lower level is retried. Since the context is removed it cannot be tried again so we are guaranteed that the program does not become unstable—although it can terminate without succeeding in interpreting the image.

The world state knowledge in this program is very simple. A more interesting program would utilize more elaborate world state updating models in which evidence collected by lower level interpreters would be combined into the knowledge base such as by using Bayesian updating (Mann & Binford 1994), Dempster Shafer (Shafer 1976), or some other approach to evidence combination. How the world state is represented and updated is part of the problem domain code. Since our demonstration system is so simple we utilized a trivial evidence combination method.

The hand written agents at Level 2 that interpret the corpus as, among other things, contexts, also generates from the contexts the pre-tests that check for context membership. Below we discuss how the pre-tests are generated from the contexts and how they are used to invoke *reflectUp* and cause self-adaptation.

8.2.4 Generating Code from Contexts

The division of corpus data into contexts allows us to generate separate interpreters for each context. The contexts are represented as PCA models and so, given a point in the n-dimensional space that makes up the context space, we can straightforwardly compute the probability that an input will occur with any given context ($P(input|context_i)$).

A context $context_i$ produces a description length for any data point that is provided to it. That is, for any image d, it produces a description length equal to $-log_2 P(d|context_i)$, so we have access to the quantity $P(d|context_i)$, but what we want is $P(context_i|d)$. That is we want to know the probability that the current context is the right one given an image.

Applying Bayes' theorem we get:

$$P(d|context_i) = \frac{P(d)P(context_i|d)}{P(context_i)} \tag{8.3}$$

so

$$P(context_i|d) = \frac{P(d|context_i)P(context_i)}{P(d)} \tag{8.4}$$

$P(d|context_i)$ is provided by the PCA model of $context_i$. $P(context_i)$ is easily estimated from the corpus by counting the frequency of occurrences of the models in the corpus.

$P(d)$ is less easy to estimate directly. However, we can compute $P(d)$ by noticing that we have access to the exhaustive set of contexts.

$$P(d) = \sum_{j=1}^{n} P(d|context_j)P(context_j) \tag{8.5}$$

and therefore:

$$P(context_i|d) = \frac{P(d|context_i)P(context_i)}{\sum_{j=1}^{n} P(d|context_j)P(context_j)} \tag{8.6}$$

We can use this quantity as a pre-test for an interpreter for each context as follows:

$$pretest : P(context_i|d) < \epsilon \tag{8.7}$$

In our implementation we chose ϵ so as to fail when the input was more than two standard deviations away from the mean.

If the world model doesn't give any indication as to which context to prefer we can select the most likely context.

$$arg \max_{context_i} \frac{P(d|context_i)P(context_i)}{\sum_{j=1}^{n} P(d|context_j)P(context_j)} \tag{8.8}$$

Since in this case we are not interested in the actual probability we can drop the denominator.

$$arg \max_{context_i} P(d|context_i)P(context_i) \tag{8.9}$$

If the context chosen by equation 8.9 fails the pre-test (yields a probability less than ϵ when computed by equation 8.6) the data point (image) cannot be handled by any of the available contexts. When this happens (reflectUp ...) allows the higher meta-level to look for an alternative way of addressing the problem (or cause an error if there is no higher level). When there is no higher level a continuable error handler can be wrapped around the top level to catch the error and to proceed using the most likely of the contexts even though it fails the pre-test. The subsequent result can be marked as *suspect* so that it is clear that there is a low confidence in the result. This can occur when an image is encountered that is not well represented in the corpus. It may also happen when conditions are outside of the operating range of a sensor. For example when a robot wanders into a dark area the images may be unusable—and unrepresented in the corpus.

Selection of values of ϵ in the pre-tests can therefore be used as a way of having confidence in the result of the image analysis. Rather than simply proceeding to interpret the image without realizing that the data is unusable the automatically generated pre-tests give us confidence that

if the pre-tests succeed and the system managed to produce an interpretation that the resulting interpretation will be of a similar quality to what the system usually achieves.

8.3 Results

8.3.1 Test Interpreted Image

Figure 8.3 shows the result of running the complete image interpretation program on one image of the test set (not included in the training set). The image at the top is the raw image data. The image at the bottom left is the ground truth. Note that although the ground truth for this image was not included in the training set the ground truth data exists—as it does for all images in the corpus. The image on the right is the result of running the interpretation program on the raw data. The results are not identical. The interpretation program found a few more regions than the expert annotated in the ground truth. The outlines of the boundaries are not identical either although they are close. The labeling of the regions while appearing in different colors in the figure are actually correct. The colors are allocated in order and since the images have different numbers of regions the color distribution is different.

```
Command: OPEN
08/14/99 13:29:29: Image /aerial/s-uk-23-kings-c-2.tif loaded.
Command: SHOW-PIXELS
Command: SHOW-GROUND-TRUTH-MAP
Command: INTERPRET-IMAGE
08/14/99 13:35:14: Segmenting...
08/14/99 13:47:53: Parsing...
08/14/99 13:51:11: Reflect up switching context
08/14/99 13:51:32: Segmenting...
08/14/99 14:03:41: Parsing...
08/14/99 14:07:06: Reflect up switching context
08/14/99 14:07:26: Segmenting...
08/14/99 14:19:12: Parsing...
08/14/99 14:25:21: Done
Command: SHOW-INTERPRETATION-MAP
Command:
```

The run included two context changes before a good interpretation could be reached.

Figure 8.4 shows a second run. As with the previous example the output is not identical. A couple of regions were not found (towards the bottom) and some of the regions grew to different extents. As before the regions were all labeled correctly.

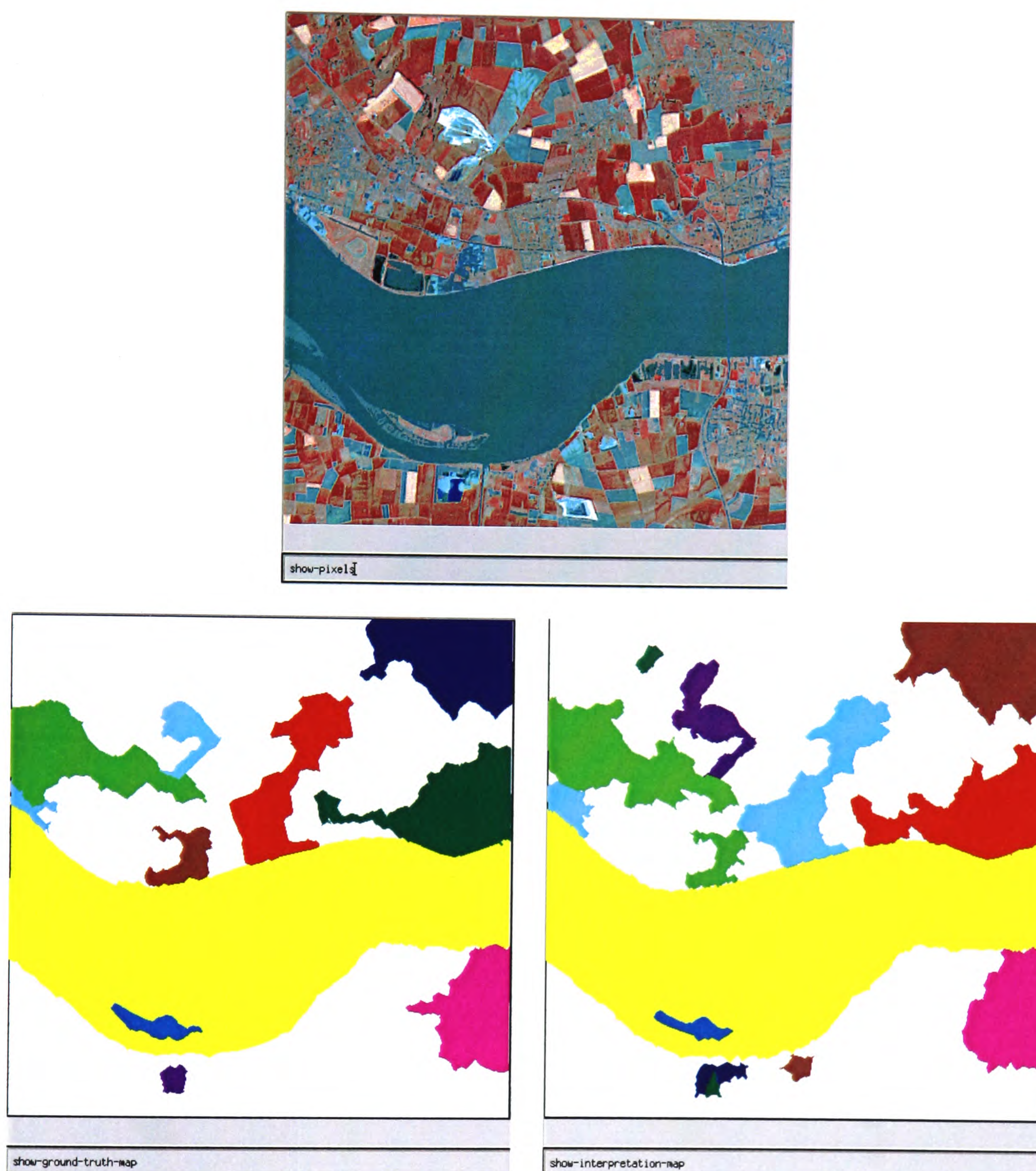


Figure 8.3: Test Run of Image Interpretation

```
Command: OPEN
07/11/01 02:05:20: Image /aerial/s-uk-06-bodml-c-3.tif loaded.
Command: SHOW-PIXELS
Command: SHOW-GROUND-TRUTH-MAP
Command: INTERPRET-IMAGE
07/11/01 02:13:11: Segmenting...
07/11/01 02:28:13: Parsing...
07/11/01 02:34:45: Reflect up switching context
07/11/01 02:35:17: Segmenting...
```

```
07/11/01 02:50:19: Parsing...
07/11/01 02:56:26: Reflect up switching context
07/11/01 02:57:52: Segmenting...
07/11/01 03:13:23: Parsing...
07/11/01 03:22:08: Done
Command: SHOW-INTERPRETATION-MAP
Command:
```

As with the first run the second run included two context changes before a good interpretation could be reached.

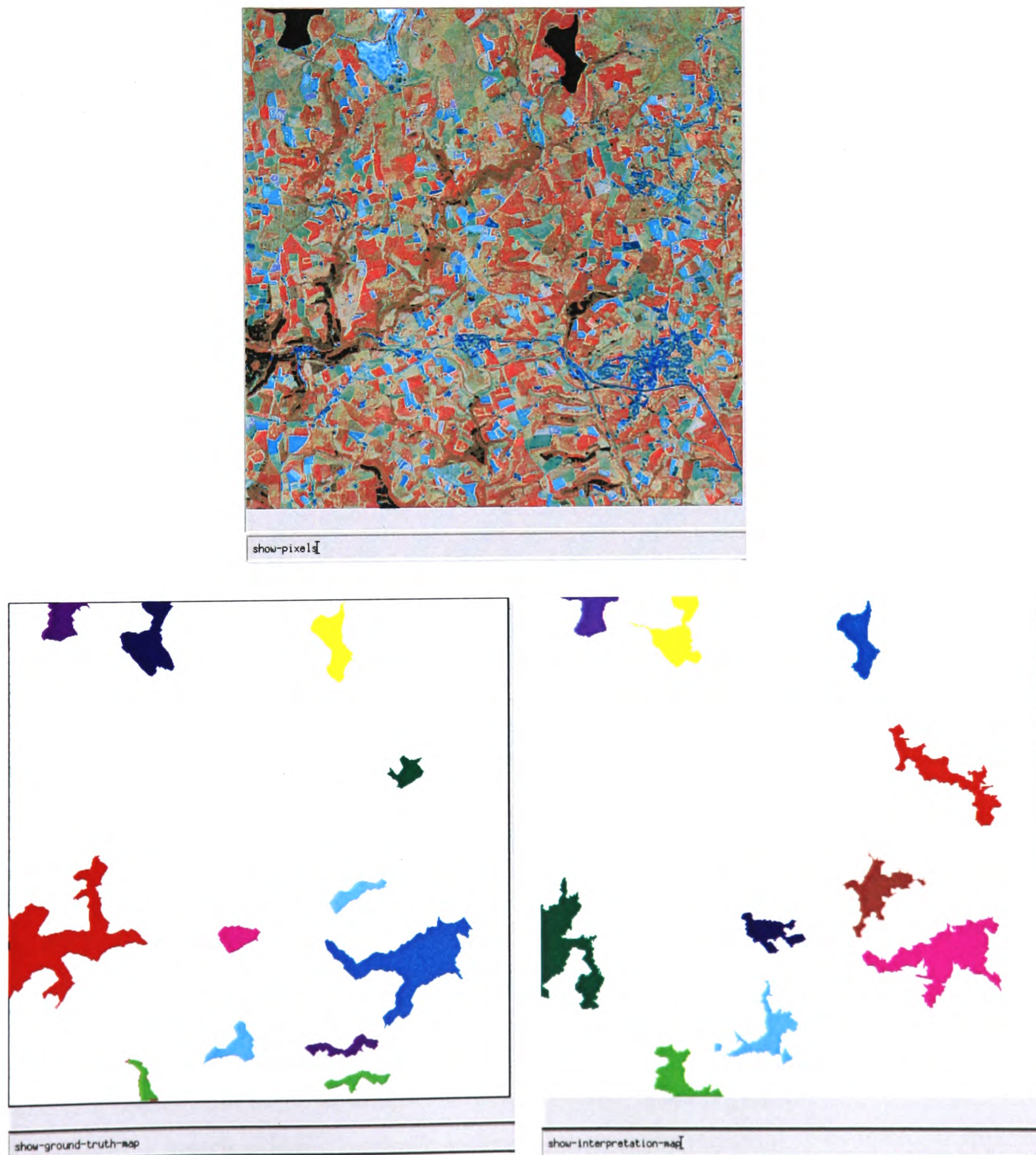


Figure 8.4: Test Run 2 of Image Interpretation

8.4 Conclusion

The program described above is very simple. It uses only two complete reflective layers, its pretests are very simple and automatically generated, the world models and evidence combination methods are also extremely simple. Nevertheless the program brings in to play all aspects of the reflective architecture as well as the segmentation program developed in Chapter 4, the labeler and parser developed in Chapter 5, and the corpus interpretation function developed in Chapter 6. While the program structure described in section is simple the program as a whole is not. The number of models for labeling, parsing and segmenting the images is significant and the contexts allow the results, generated by the system, to be quite similar to those generated by the human photo interpreter.

The test system that we developed produced 14 optical contexts and 21 labeling contexts. These are relatively small numbers. They are small because our system consisted of only two layers and although there were three cascaded interpreters (segmenter, labeler, and parser) there were only two kinds of context used since there was inadequate test data to make grammar contexts. Additionally the images were fairly homogeneous in content and imaging conditions. Even with those limitations the system permitted 14*21=294 unique configurations. Using realistically sized problems the number of stages and contexts would grow significantly and the number of possible configurations would soon be in the thousands. The numbers are only interesting insofar as they provide some indication of the futility of trying to program and manipulate such complexity with hand written code. Grounded self-adaptive approaches such as we have developed in this thesis, in some form, are surely necessary in building robust vision systems that operate in unconstrained environments.

The compelling reasons for the use of self-adaptation and automatic context induction are as follows:

1. Context induction allows signature of images to drive self-adaptation.
2. Contexts support better interpretations for the same reasons that context helps in problems involving frames (Minsky 1975), or schemas (Draper *et al.* 1988).
3. We know when we are operating out of bounds—and should self-adapt.
4. We know when we cannot adapt so as to improve the context match—and should therefore not trust the results.

The reflective layer diagram (Figure 8.2) is similar to the *learn by example* architecture of Materic (Figure 6.4) discussed in Chapter 6. Both attempt to mimic performance by example and both attempt to match existing methods against observed phenomenon. The GRAVA approach however has a coherent model for producing similar behavior based on a combination of "Fidelity" and an MDL compiler/theorem prover that together maximize the mutual information, for the

entire generated program, between the observation and the generated program. The GRAVA approach while limited in layers in the demonstration program discussed in this chapter can easily be extended by adding additional layers. Materic's architecture has no provision for dividing the observations into contexts, or for that matter, switching between contexts.

Chapter 9

Conclusions and Further Work

9.1 Introduction

At the outset of this project, we were interested in applying ideas of reflection and self-adaptation to building more robust image understanding systems.

Vision systems lack robustness for a variety of reasons. Some of those reasons relate to algorithmic inadequacies. However, an equally significant reason for the lack of robustness is that vision programs don't know what they are doing and don't know when they are doing poorly. The world is highly complex and changing but vision algorithms are typically built to handle small subsets of the complexity that the world offers and they assume static conditions. When images are outside of the narrow range that the programs are designed to deal with the results are often bad: performance degrades disgracefully.

This thesis was based on the idea that if vision programs knew what they were doing, and knew when they were doing poorly, then they could adjust their assumptions and thereby do a better job. It seemed that the idea of reflection, developed in AI, offered a way to do this. That approach lead us to utilize reflection in a novel way—to support self-adaptation. Self-adaptation is itself a new idea. This meant that the thesis work had to develop simultaneously on two fronts:

1. developing a set of vision modules to tackle a difficult problem; and

2. to develop reflective architectures in order to build a self-adaptive vision system.

We selected the understanding of aerial images as a problem domain although almost any problem in image understanding would have served our purposes equally. Most of our effort has been spent on the problem domain, building a new segmentation algorithm that could permit higher level semantics to influence the segmentation, and a 2D parser to enable structural descriptions to be generated from a segmented image. Much effort went into the development of image corpus development tools (the image annotation tool described in Appendix B), developing the image corpora, and implementing agents that interpret the corpora.

We were able to get positive results from all phases of the research, including the final step of making the program self-adapt in order to produce an acceptable interpretation of images. However, much, remains to be done. Our original goal was to process a sequence of images in which the terrain changed somewhere in the sequence (such as from urban to suburban to rural) so that at these transition points the program would self-adapt in order to maintain a good

level of interpretation of the image while undergoing transitions that might have resulted in poor performance in conventionally structured programs. In practice we had only enough resources (image data) to process single images that may or may not involve self-adaptation. Our world model consisted of a canonical world assumption. Images from the canonical world are interpreted without adaptation while other images require self-adaptation in order to accommodate the appropriate contexts for interpretation. Much more experience with the self-adaptive architecture developed in this thesis is required in order to fully access the merits of the architecture.

The problem domain involved segmenting, labeling, and parsing aerial images. We had no consumer for the image parse and so it is difficult to access the success or usefulness of the parse for any particular use. The parses that were generated seemed plausible when inspected by hand. The parse process was certainly useful as a mechanism for propagating non-local context in arriving at plausible region labeling. We did not collect enough data to quantify the benefit to labeling that the parser provided but anecdotally there are many cases where otherwise ambiguous regions were correctly labeled.

The size of the corpus, though large by the standards of much vision research, was really too small. The grayscale corpus consisted of 40 images while the color corpus consisted of 105 images. Neither corpus had a lot of different contexts nor did they have a lot of examples of each context. The grayscale corpus included urban, suburban, and rural contexts but the images were otherwise homogeneous both in imaging conditions and even content. We ended up not gathering results for the grayscale corpus. The color corpus included a number of difference imaging modalities and somewhat greater variation in geographical location and image content. There were too few examples of some of the contexts to make separate parse rule contexts and inadequate sequences of images to support the initial goal of handling sequences of consecutive images.

Nevertheless, the data that we were able to develop was sufficient to test all phases of the architecture and the special purpose agents that we developed along the way.

We are encouraged by the success of the architecture within the limitations discussed above. We have succeeded in building a segmentation algorithm that is capable of using higher level semantics in order to produce appropriate segmentations (Chapter 4). We have produced an image parser that produces structural descriptions of the image in terms of labeled regions(Chapter 5), and we have shown how models can be induced from an annotated corpus and used as the basis for the segmentation, labeling, and parsing of an aerial image.

9.1.1 Overview of the Chapter

In section 9.2 we review what was learned in the course of the research. Section 9.3 introduces some specific ideas for further work beyond the need for "more of everything". In Section 9.4 we look forward and attempt to find a context and direction for this new approach to solving complex problems.

9.2 What We Have Learned

9.2.1 Usefulness of Interpretation Problems

In designing the GRAVA architecture, we focussed on a class of problems that we called "Interpretation Problems". Of course, not all problems are interpretation problems, though surprisingly many are. Computer Vision and many of its associated tasks fit well into the view of interpretation problems.

Interpretation problems can be solved by fitting a model to an input so that the input can be interpreted as the resulting parameterized model. That parameterized model is what we refer to as an "interpretation". We have demonstrated that even the task of compiling specifications into code can be viewed as an interpretation problem. GRAVA agents are defined as pieces of code that attempt to *fit* a model to the data which is presented to it.

Many interesting interpretation problems have the property that they have several plausible solutions. Some of these interpretations are more likely to be "right" than others. We are not satisfied by finding the first solution that provides a solution. We wish to find the "best" solution—in the sense of being the most probable solution. We have successfully solved a variety of problems in which Monte Carlo sampling allows us to find the solution that are approximations to the minimum description length.

9.2.2 Benefits of the GRAVA architecture

The GRAVA architecture allows agents to be implemented, in a conventional way, that encapsulate focussed image processing capabilities. The architecture then takes care of managing the manner in which the agents are connected together to form a working program. The architecture is grounded in the real world by virtue of the statistical modeling and the corpus. It might be grounded in other ways too—such as by the use of qualitative reasoning and physical models (see Section 9.3). Instead of building models of the real world by hand that are necessarily naive, we build meta-models, such as PCA face templates, that can be learned from real world data.

Other systems, such as the influential PDP approach to reading, force the abandonment of traditional programming methods in order to gain the benefits of the approach. Our example in Chapter 3 of a simple reading program illustrated that similar results to those achieved in PDP can be achieved in a much more conventional and explicit way.

The overwhelming complexity of the natural world argues for significant complexity in any program that attempts to make sense of it. The approach developed in this thesis enables researchers to continue to develop better algorithms for image processing while allowing the complexities of building a program to deal with the natural world to be learned.

9.2.3 Reuse of Ideas from Computational Linguistics

Many ideas from computational linguistics appear to have counterparts in image understanding. The use of statistical learning techniques in which knowledge (such as grammar and part of speech models) is acquired from a corpus is a well established approach that has yielded surprisingly good results when compared to human performance. In Chapter 5 we showed that grammatical techniques also have a place in image understanding and that, just like in natural language understanding, it is possible to acquire statistical grammars from annotated corpora.

9.2.4 Benefits of MDL

MDL has been used throughout the thesis as the principle by which the best solution is sought. GRAVA has a MonteCarloSelect algorithm that is used by all of the algorithms as a means of approximating the MDL solution. MDL has previously been proposed as a means for finding the best solutions within constrained situations and is the basis for some segmentation algorithms (Zhu & Yuille 1996; Leclerc 1989). Our use of MDL recognizes MDL as a unifying force. In GRAVA, MDL is used as a currency that crosses semantic levels.

9.3 Further Work

The ideas in this thesis require further validation from "more of everything"—more agents with which to interpret the images, larger corpora from which to build more and better contexts, and more experience with the process of self-adaptation.

There were of course many specific ideas that cropped up in the course of the research that could not be explored within the constraints of the project. The more interesting of these ideas are described below.

9.3.1 Resource Contention Real-Time Constraints, and Utility

The compilation of interpreters as described in Chapters 7 and 8 collects all relevant agents for a chosen context and puts them into the interpreter. This means that all agents that are valid within the context and which may contribute to the computation will be available for the interpretation task. We have not taken into consideration the possibility that there may not be sufficient computational resources to run them all. We have concentrated on finding an approximation to the best interpretation at any cost.

In certain practical situations, it may be necessary to further limit the agents available in an interpreter. To do this we would need:

1. some representation of a resource budget,
2. some representation of the cost of running an agent,

3. some way of estimating the extent to which agents overlap in their coverage of the interpretation space, and
4. some way of distributing the resource budget over the entire program.

Making the system real-time is similar to any other resource problem where the resource is time.

Solving resource constrained situations within the framework of the GRAVA architecture is by no means a trivial exercise. Three issues make the problem non-trivial.

1. Agent Overlap: If two agents provide coverage of the same part of the computational space there may be very little added utility in having both agents. Knowing the extent of overlap of agents is itself a non-trivial problem.
2. Cost Distribution: It may be best to front-load the resource usage and then cut back in latter stages, it may be better to do the converse, and it may make sense to distribute the resource usage evenly. Some parts of the problem may have redundant coverage and therefore not suffer from not including some agents whereas other parts may barely have enough agents to cover the space and serious quality degradation may result if any agents are omitted. Some of these issues are problem dependent and some depend upon the distribution of problem coverage. Finding a distribution that maximizes a simple utility function is unlikely to be an adequate solution especially if (1) is unsolved.
3. Tradeoff missing agents versus less Monte-Carlo sampling: The total cost of computation is governed by the number of samples taken by the Monte Carlo sampling at each phase of the interpreter. Reducing the number of samples reduces the resource utilization but also results in a poorer approximation to the best solution. When trying to manage resource utilization therefore there is a choice to be made between taking fewer samples and including fewer agents in each sample. The solution is not obvious.

First thoughts on this problem suggested that a simple utility function would allow the compiler to limit the agents added to an interpreter so as to reduce the cost. However, the above considerations convinced us that the issues of real-time constraints and resource limited computation were significant problems that we did not have the resources to address at this time.

9.3.2 Unsupervised Learning

We have seen several places where it is possible to learn a new model without the need for an expert annotation. Typically these situations occur when smoothing has occurred. Consider the case of the parser. Since not all parse rules are likely to be encountered in the corpus we have a way of generating rules that allow the parse to succeed. Similarly we may have a region for which we don't have a content model that would allow us to label the region so we pick the most likely region given the available parse rules that may consume it. In the present system

these "smoothed" models are not retained and are given low probabilities so that they are not preferred over rules that do occur in the corpus. If the smoothed models were retained as if they had been encountered in the corpus they could participate in future interpretations. Each time the rule is used, whether it is an original model from the corpus or a model learned from a prior smoothing, its frequency of use could be tracked as the program encounters real data. Over time, the system would converge on a set of rules that match what the "real world" suggests it needs.

This view makes smoothing not just a way of dealing with the inevitable lack of completeness of the corpus but a way of learning new models in an unsupervised way.

Having the system continue to accumulate frequency information of existing and smoothed models means that the probabilities continue to converge upon the world as the running program experiences it. In such a view the corpus represents a starting point for a set of models of the real world. Subsequently, the program would converge from that starting point to a set of models and associated description lengths that fit well with the real world.

There are a number of potential problems with this approach.

1. If experience is limited to a particular view of the world, it may discriminate against other possibilities within the same context that were represented in the original corpus in a balanced way.

2. Wrong rules may be learned through smoothing that subsequently are reinforced by other occurrences.

The approach seems plausible because with a multi-stage interpreter there is a sense in which latter stages "validate" smoothing that occurred in earlier stages. This may result in bad smoothed rules not being learned often enough to be a problem.

The idea of a system that continues to converge after it has extracted a starting point from the annotated corpus is appealing. It potentially reduces the size of the corpus necessary to get a good system built. A robust system may be best constructed by inducing an initial set of models from a corpus followed by unsupervised training on representative images until the performance of the system is deemed to be adequate.

This issue is clearly not a trivial one and so we resisted the temptation to explore it.

9.3.3 Picture Language for 3D

The meta-grammar developed in Chapter 5 for parsing aerial images took advantage of the almost two dimensional nature of such images. Only cloud occlusion was considered as a special case. For high altitude aerial images the grammar seems to work well. For low altitude images where 3D effects are more pronounced or for ground-based images, such as from a mobile robot, a true 3D grammar is necessary. The success of parsing images using the statistical parser techniques with a learned grammar suggests that similar success may be achieved with true 3D scenes if the meta-grammar was extended to true 3D.

9.3.4 Other Ways of Estimating Description Length

We have focussed on estimating description length by counting frequencies in the corpus and by training agents against corpus data and computing *Fidelity* as described in Chapter 7. There may be other viable ways of determining description lengths including the use of physical models and qualitative models of the process being modeled.

9.3.5 Image Reconstruction

The approach taken to image interpretation is to treat it as a problem of finding the most probable interpretation. We have used the view of "most probable interpretation" as "minimum description length".

The interpreters apply agents to the data in order to account for the data in the image by *fitting* models to the data. The resulting description of the image is thereby rendered as a structure of parameterized models. The description length of the entire such structure is the quantity that we minimize in order to find the best interpretation.

The resulting description reduces the overall description length of the image from its original representation as a collection of pixels to a semantic representation based on models of image content. In principle, the image could be regenerated from its description and rendered back as a collection of pixels. This observation makes it clear that interpretation really is a form of intelligent image compression. It is compression because it produces a reduced description length representation of the image that can be re-expanded to form an image and it is intelligent because what is retained about the image is determined by the models that the agents *fit*. By supplying an appropriate set of models for the agents to fit we can therefore control what detail of the image the representation retains. Where most image compression algorithms use compression algorithms that attempt to maximize the quality of the expanded image as perceived by the human eye, a semantic compression algorithm such as we outlined above could allow semantically important parts of the image to be retained where a standard compression algorithm might discard seemingly unimportant detail that actually has important semantic import.

9.3.6 Stability and Predictable Behavior

Self-adaptive software is like a control system. The software adapts in an attempt to maintain a set point where the set point is the specification or goal used by the compiler.

People are nervous about self-adaptive software because it is not possible to test in a conventional sense all of the possible adaptations that the software might take.

A paraphrase of an argument that is frequently made regarding self-adaptive software is:

"We don't want a missile flying through the air rewriting its software because we don't know what it will do. We can't test software that we have never seen. How can we be confident that the software will not do something disastrous?"

With the GRAVA architecture, the straight-forward issue of the what the software will do seems to be addressable. The compiler is a theorem prover that compiles a program that meets the goal that it was supplied with. The compiler is not going to synthesize a program that does something different from what it was programmed to do by virtue of its specification. Furthermore, the interpretative stages check at each point whether the interpreter is operating within its operating range. If it is not, it reflects up to cause an adjustment. So where a conventional piece of software might wander outside of its intended region of operation and begin behaving erratically without any idea that anything is wrong, a GRAVA program will:

1. know that something is wrong,
2. attempt to self-adapt to solve the problem, and
3. if self-adaptation fails to find a viable context for operation cause a reflect up to a higher level.

The highest level which we have referred to as the debugger may include a handler that makes the missile self-distruct harmlessly.

Concerns over a GRAVA program adapting into a monstrous and dangerous piece of software appear therefore to be manageable. That the compiler is a theorem prover provides some confidence in the relationship that the resulting program has with its intended behavior. Over time such systems should be much more reliable and worthy of our trust than conventional systems.

There is however a concern that is not so easily dispensed with: unstable behavior.

In our example in Chapter 8, the system self-adapted twice before it was able to interpret the image presented to it. Each adaptation brought the system closer to what was required in order to interpret the image. One adaptation was to adjust the optical context and one was for the labeling context. the system converged until it was able to interpret the image. In an unstable system the program would either get no closer to its set point or would diverge from the set point resulting in an infinite series of self-adaptations.

Design methodologies have been developed for linear control systems and for non-linear control systems. Intuitively it would be nice if these methods could be adapted for self-adaptive software systems. Unfortunately, the techniques cannot be applied straightforwardly because software is highly non-linear. It most cases, it is impossible even to decompose the problem into piecewise approximations to linear.

The design of how world knowledge is updated during an adaptation appears to be the key to controlling whether a self-adaptive program implemented in GRAVA will be stable. What is needed is an approach that allows us to verify that the resulting program cannot get into an unstable cycle of self-adaptation.

We have attempted to come up with a general solution to this problem but thus far have not come up with anything satisfactory. It is clearly a hard problem, an important problem, and for the time being remains open.

This problem belongs to the general class of problems that might be classified as "semantics of self-adaptive software". In order to be confident in the performance of self-adaptive programs semantics of self-adaptive programs may become an important area of research. There are already some attempts being made at early investigations into the semantics of self-adaptive software such as that of Meng (Meng 2000) and Pavlovic (Pavlovic 2000).

9.4 Looking Forward

As self-adaptive software is a new field of exploration we would be remiss if we didn't take time to speculate as to how it might fit in to our existing understanding.

The kind of computation that GRAVA supports is different in some interesting ways from conventional programs. As such, it may be necessary at some point to rethink some of our assumptions about computation and the theory of computer science.

9.4.1 A New Kind of Computation

There are many useful ways of partitioning computation but the dimension that interests us in this thesis is the dimension of predictability of the environment. In the following discussion, a computation that depends upon a completely determined environment will be referred to as a type-1 computation and a computation that does not depend upon the environment to be completely determined is referred to as type-2.

Almost all of computer science developed to date has focussed on type-1 computations. When we try to accommodate environments that are less well determined, the complexity rapidly increases to the point where the programs are difficult to maintain. Programs that perform input/output are simple examples of where this problem is frequently faced. Because the environment is not well determined, a program that has to (say) read in a command string must face the problem of dealing with the full range of possibilities and respond to them appropriately. If we knew that a program would read a well formed command each time it would be easy. In such cases we restrict the problem by defining any input that does not fit the pattern of a correct input as being an error. Even with this simplification I/O programs are often huge and buggy.

Hypothesis: Most interesting computational problems are type-2.

In the previous century we have largely been concerned with the open loop absolute formulation of computing but it represents a small fraction of the types of computation that we are interested in.

Nature provides countless examples of computational systems. They are almost without exception type-2 and they are robust. Now we wish to bring computation into the realm where nature has provided us with so many successful examples and we find that type 1 computation does not scale to the real world where absolutes disappear and guarantees are all but nonexistent.

Programs that have to interpret visual scenes are an interesting example of type-2 computa-

tions. The environment cannot be precisely specified and all scenes are legal. We wish to make sense of all visual scenes that the camera can capture.

Until recently we have built the theory of computation around the notion of type-1 computation and developed notions of program correctness around the assumption that the environment can be accurately and completely modeled. We have been able to do this because for the most part computation has occurred within the artificial environment of a computer separated from the outside world except for some input channels whose inputs have been rigorously specified.

While we have seemingly been able to do a lot with type-1 computation it is what we cannot do well that argues for a new theory of computer science that embraces the more broadly general view of computation that includes both type-1 and type-2 computation.

Even the simplest animals exhibit type-2 computations that are impressively robust. By comparison our ability to build robots that explore our unconstrained environment is limited in part by a lack of understanding of how to build type-2 computations. Programs that interpret visual scenes suffer for the same reason.

When we have absolutes we can build a system that behaves perfectly. This is conventional programming. This is a rare occurrence in nature.

When success of a computation cannot be guaranteed for any reason we must:

1. Check how well the computation did.
2. Be prepared to take some constructive action if the computation didn't do well.

This formulation of an outline for type-2 computations is similar to a control system. Rather than simply assuming that the computation performs as expected, the result is measured against an expected behavior and when a deviation is detected a corrective force is applied and the computation retried in order to bring the result closer to the expected behavior—the set point. In this formulation type-2 computations consist of type-1 computations encapsulated in a control system. This allows us to benefit from everything that we know about building correct type-1 systems and requires additionally that we have mechanisms for:

1. Knowing the program goal.
2. Measuring how closely the program goal was met.
3. Applying a corrective force to bring the program behavior closer to the program goal.

Self-adaptive software is an attempt to build type-2 computations as systems that apply a corrective force by making changes to the program code.

9.4.2 An Architecture for Intelligence

We consider GRAVA to be an architecture for AI vision. There is no accepted definition of what constitutes AI. We propose the following as a useful definition that may be inadequate but might well be a partial definition for AI.

An AI program is one that can operate robustly in a natural unconstrained environment to a similar extent to which existing natural computational systems do. Nature has provided us with a wide variety of examples. Many of them we would not consider to be intelligent but by virtue of their ability to survive in our harsh environment they are by and large robust in the sense that we have been using the word throughout this thesis. So far we have been notoriously unsuccessful at producing artificial systems that can come close to the robustness exhibited by natures simplest examples.

The self-adaptive approach outlined in this thesis appears to be a step in the direction of robust computing for unconstrained environments. In this thesis, we have taken the first very small step along what promises to be a very exciting journey.

Appendix A

Corpora

A.1 SPOT Satellite Images

The multi-spectral corpus is based on images provided by the European space agency satellite SPOT.

SPOT's sensors scan Earth from a sun-synchronous, near-polar orbit of 830km altitude in two different modes - panchromatic and multispectral. The panchromatic sensor records radiation reflected from Earth's surface at a ground resolution of 10m in one waveband (0.51-0.73 microns), giving black and white imagery. The multi-spectral sensor records reflected radiation at a ground resolution of 20m in three, discrete wavebands - 0.50 - 0.59 microns (green light), 0.61 - 0.68 microns (red light), and 0.7-0.8 microns (very near infrared light). The three multispectral bands of data can be passed through three color guns of a computer monitor giving 'color composite' imagery.

We have developed our multi-spectral corpus by taking 512x512 pixel areas of interest from the 20m multi-spectral images. The corpus consists of 105 such images of various locations around the UK.

The SPOT images were annotated with the following region types:

Region Type	Type ID	Description
Industrial	R1	Factories, large warehouses and associated clutter.
Residential	R26	Sparsely populated residential areas.
Downtown	R2	Major downtown areas (denser than urban).
Urban	R4	Urban area of a town or downtown area.
Suburban	R3	tightly packed residential area.
Metro	R30	An isolated town area.
Town	R31	A town
Field	R5	A field
Farmland	R20	A collection of fields.
Park	R21	A park (usually within a dense area).
Estate	R36	A major home with grounds.
Private Airport	R7	A small airport.
Commercial Airport	R8	A large airport.
Port	R15	A sea port.
River	R10	A river (only major river portions).
Sea	R11	Sea
Lake	R12	Lake
Reservoir	R13	Reservoir.
Beach	R33	Beach
Island	R18	An Island (small)
Swamp	R14	Swamp.
Frozen	R29	Snow or ice covered area.
Rock Formation	R32	Bare rock.
Forest	R35	Tree covered area (sometimes small).
Cloud	R22	A cloud obscuration (rare).
Shadow	R24	Shadow (usually cast by a cloud).
Unknown	R666	An identifiable area of unknown content.

A.2 MASS GIS Images

The gray scale corpus uses images provided by the Massachusetts Geographic Information System (MassGIS).

Figure A.1 shows a portion of an image in more detail The white line indicates the boundary of the region that in this case has been annotated as "suburban".

These medium resolution images are envisioned as the next generation of "basemap" for the Commonwealth by MassGIS and the Executive Office of Environmental Affairs (EOEA).

The Massachusetts Institute of Technology has developed a WWW based Digital Ortho-photo Browser that was used to obtain the images used in the corpus.

The following describes the manner in which the images were produced. Stereoscopic aerial photography with 80% forward and 40% side overlap was collected along flight lines running approximately north/south during spring "leaves off" periods at an altitude of 15,000 ft. with a 6 inch mapping camera with forward motion compensation.

Ground control targets were set out prior to the flight and photo recognizable points were substituted for targets that were lost. Horizontal control conforms to the Federal Geodetic Control Committee specifications for Second Order Class 2 GPS surveys. Vertical control is within 10 cm.

Aerial Triangulation (AT) block models were developed and tested for accuracy. Adjacent blocks are tied to new ones to insure a "seamless" image.

The photography was scanned at 15 microns and the images were differentially rectified. The histogram for tonal adjustment provides for a range of gray shades from 30 to 225 (out of 256), thus allowing pure black and white to be legible when over plotted on the images. Accuracy

Figure A.1: Portion of image showing detail of a region annotated as "suburban"

of the image was given precedence over tonal consistency at the edges of the images. The final digital images were clipped with the Ortho-photo Index Grid, thus the tiles do not overlap. The images meet or exceed the National Map Accuracy Standards to the extent that 90% of the well defined features fall within 0.5mm of their true position on the ground at the nominal output scale of 1:5,000 (2.5m on the ground). Additionally, the maximum displacement of well defined features is less than 5 meters. Each pixel in the digital ortho-photo image represents 0.5 meters on the ground.

Each tile contains 8,000 x 8,000 pixels which equates with a file of 64 megabytes (mb). These images have been resampled at 1 (16 mb), 2 (4 mb) and 5 (640 kb) meter resolutions.

We used the 2 meter (4Mb) TIF images to build the corpus. Out of the 109 images of the greater Boston area available at the time of this project, 40 were selected and used for inclusion in the corpus. Figure A.2 shows the area covered by the MassGIS ortho-photos. The grayed areas enclosed in a dark solid line are the tiles used in the corpus.

These 40 images were annotated using the annotation tool described in Appendix B. Although the corpus consists of only 40 images (compared to the multi-spectral corpus' 105 images), the images are 2000x2000 pixels or nearly 16 times as big as the images in the multi-spectral corpus. If the images were to be divided into 512x512 sized images, the corpus size would be 640 images.

Figure A.3 shows (about 70% of) an image with hand segmentations visible by their boundaries.

The MassGIS images were annotated with the following region types:

Figure A.2: Area covered by MassGIS ortho-photos

Region Type	Type ID	Description
Industrial	R1	Factories, large warehouses and associated clutter.
Downtown	R2	Major downtown areas (denser than urban).
Dense Urban	R41	Dense urban.
Urban	R4	Urban area of a town or surrounding downtown area.
Suburban	R3	tightly packed residential area.
Residential	R26	Sparsely populated residential areas.
Field	R5	A field.
Farmland	R20	A collection of fields.
River	R10	A river (only major river portions).
Sea	R11	Sea.
Lake	R12	Lake.
Reservoir	R13	Reservoir.
Beach	R33	Beach.
Island	R18	An Island (small).
Swamp	R14	Swamp.
Forest	R35	Tree covered area (sometimes small).

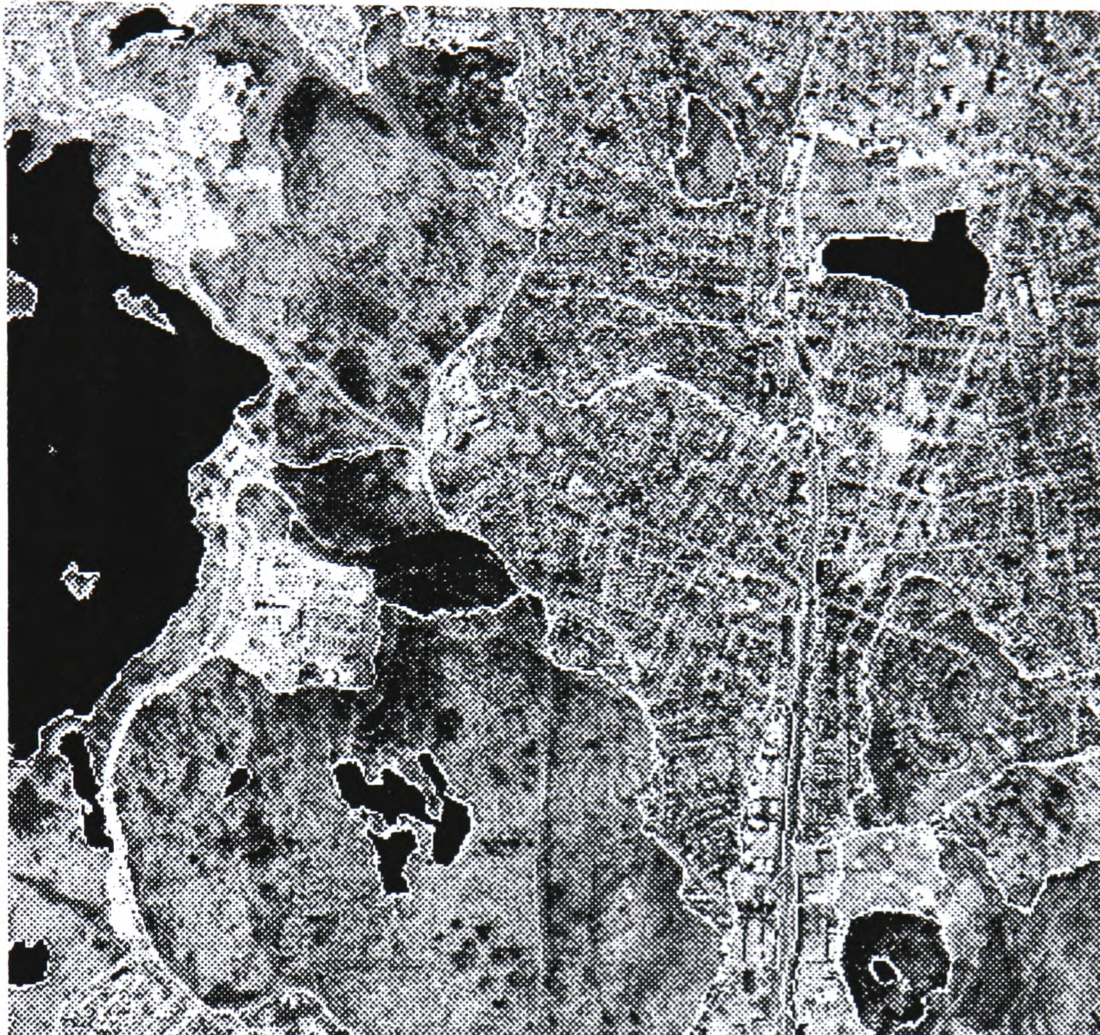


Figure A.3: Portion of a MassGIS ortho-photo image with annotations

Region Type	Type ID	Description
Mixed Vegetation	R58	Mixed Vegetation.
Grassland	R57	Grassland.
Commercial	R59	Commercial buildings.
Hanger	R42	Hanger (at an airport).
Ramp	R43	Ramp (part of an airport).
Taxiway	R44	Taxiway (part of an airport).
Terminal Buildings	R46	Terminal buildings (part of an airport).
Estate	R36	A major home with grounds.
School	R48	A school + grounds (usually with baseball fields).
Football Stadium	R51	Football stadium.
Baseball Field	R47	Baseball field.
Tennis Court	R60	Tennis court.
Golf Course	R56	Golf Course.
Park	R21	A park (usually within a dense area).
Major Intersection	R28	Major highway intersection (e.g. cloverleaf).
Parking Lot	R50	A parking lot.
Port	R15	A sea port.
Boat	R17	A boat.
Quarry	R49	A quarry.
Unknown	R666	An identifiable area of unknown content.

Appendix B

The GRAVA software architecture

B.1 Introduction

This appendix describes the software environment that was implemented to support the work described in the body of the thesis. The image annotator is a complete application and is described in appendix C.

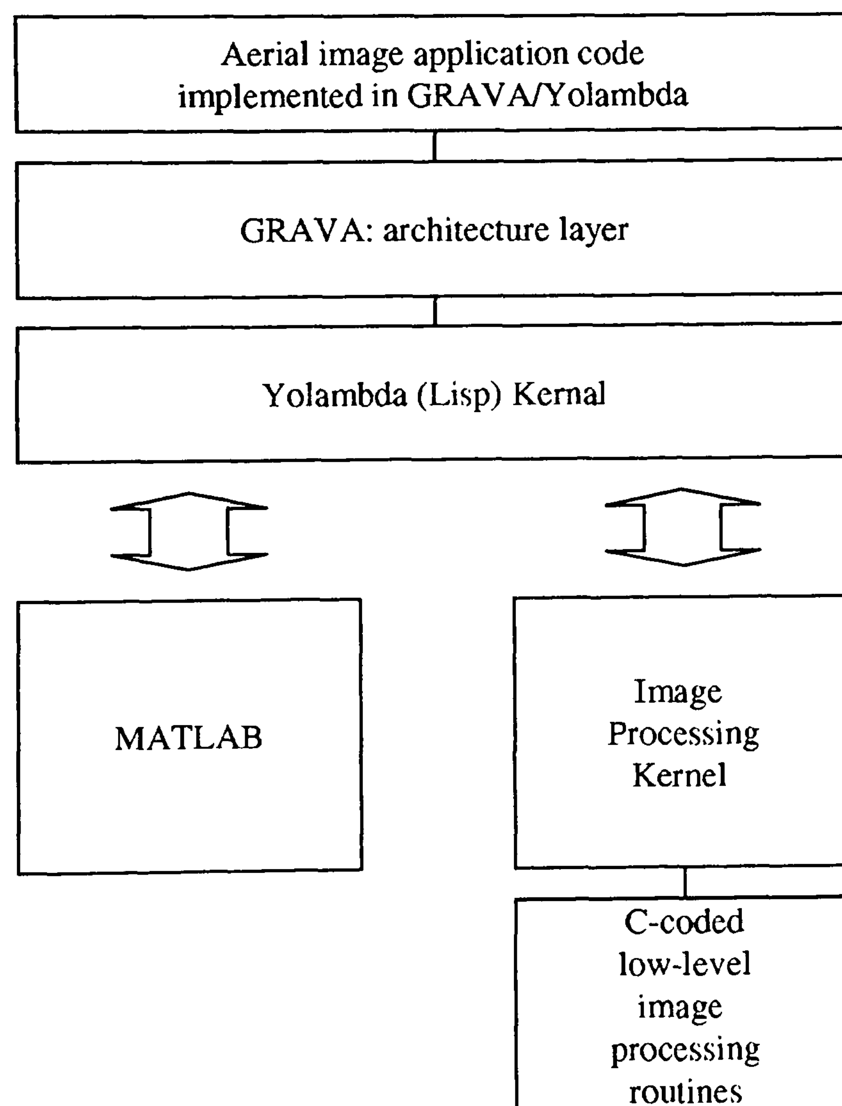


Figure B.1: Schematic view of the Software Architecture

Figure B.1 shows a schematic view of the software architecture used in building the systems described in the thesis.

Yolambda was used as the main implementation language for the project. An introduction to Yolambda is given in Appendix E. Low-level image processing capabilities including such things as reading, writing, and displaying image files was provided by an image processing kernel written in C++. Some low-level routines in support of segmentation and the gathering of statistics from the image corpus were implemented in C++ and linked in with the image processing kernel.

The GRAVA architecture layer was implemented in Yolambda and the application itself was implemented in a combination of Yolambda and GRAVA.

MATLAB was interfaced to Yolambda to provide certain essential services notably matrix math functions and plotting.

Everything except for MATLAB and a few borrowed routines were implemented by the author.

B.2 Yolambda

Yolambda is a language that was developed to support research in to reflective programming (Robertson 1992; Laddaga & Robertson 1996). It was clear from the start that reflection would play an important role in the self-adaptive architecture so a language that was designed to support research into reflection was an obvious choice. Additionally the source code was available and it could be easily integrated with other libraries making it a suitable candidate for the basis of an experimental architecture.

B.3 Image Processing Kernel

The image processing kernel is written in C++ and provided the following capabilities:

1. reading and writing of images files as TIFF, or JPG.
2. writing animated GIF files—used to observe and debug the segmentation algorithm by producing "movies" of the progress of the segmentation.
3. a library of edge detectors.
4. a library of low-level image processing primitive operations including wavelets (implemented by Xie) and region competition segmentation (implemented by Hu Xing).

B.4 Matlab interface

MATLAB provides a C++ interface which we used to interface MATLAB to Yolambda. No MATLAB code was written as part of the project. MATLAB was used entirely through its interface to Yolambda.

Below we list the MATLAB interface function as they appear to Yolambda and an example MATLAB program written in Yolambda using the interface functions.

The MATLAB functions themselves are documented in the MATLAB documentation.

The following Yolambda procedures were added to ease the integration of Yolambda and Matlab functions.

```
(defineClass MatlabMatrix (Matrix)
  ((data initform nil)
   (rows initform 1)
   (cols initform 1)
   (name initform #f)))
```

A simple Yolambda native matrix representation.

```
(openMatlab)
```

Causes MATLAB to be started and a connection made to the running MATLAB. openMatlab must be called before any other MATLAB interface functions are invoked.

`(closeMatlab)`

Closes the connection to the running MATLAB and causes MATLAB to exit.

`(makeZeroMatlabMatrix dim ...)`

Makes a zero MATLAB matrix of the specified dimensions. The matrix is created in MATLAB and local MatlabMatrix object is returned.

`(makeMatlabMatrix name dim ...)`

Makes a MatlabMatrix with the specified name and dimensions.

`(makeNamedMatlabMatrix name dim ...)`

Makes a MatlabMatrix with the specified name and dimensions.

`(matrixRef MatlabMatrix|mat i j)`

Retrieves the value at the (i, j) location specified from the Matlab matrix.

`(matrixSet! MatlabMatrix|mat i j value)`

Stores a value at the (i, j) location in the Matlab matrix.

`(matrixDimension MatlabMatrix|mat)`

Returns the dimension of the Matlab matrix.

`(makeUnitMatlabMatrix dim)`

Makes a square unit Matlab matrix.

`(copyMatrix MatlabMatrix|mat)`

Makes a copy of the provided Matlab matrix.

`(matrixMultiply MatlabMatrix|mat1 mat2)`

Multiplies Matlab matrices mat1 and mat2.

`(matrixSubtract MatlabMatrix|mat1 mat2)`

Subtracts the Matlab matrices mat1-mat2.

`(matrixTrace MatlabMatrix|mat)`

Returns the trace of the Matlab matrix.

`(getMatlabMatrix name)`

Returns a Matlab matrix by finding the already existing Matlab matrix with the matching name.

`(matrixDestroy MatlabMatrix|mat)`

Destroys the Matlab matrix.

`(mlEval command)`

Executes the Matlab command specified in the command string.

`(matrixInverse MatlabMatrix|mat)`

Inverts the Matlab Matrix.

`(matrixEigenvalues MatlabMatrix|mat)`

Returns the eigenvalues of the Matlab matrix.

`(matrixEigenvectorsAndEigenvalues MatlabMatrix|mat)`

Returns the eigenvectors and the associated eigenvalues of the Matlab matrix.

`(matrixDeterminant MatlabMatrix|m`

Returns the determinant of the Matlab matrix.

B.4.1 Matlab Demo in Yolambda

```
;;; Translation of engdemo.c into Yolambda

(define BUFSIZE 256)

;;;  * PART I
;;;  *
;;;  * For the first half of this demonstration, we will send data
;;;  * to MATLAB, analyze the data, and plot the result.

(define engdemo
  (lambda (time)
    (let ((T (makeNamedMatlabMatrix "T" 1 (length time)))) ; Create a variable for our dat
      (dotimes (i (length time)) (matrixSet! T 0 i (at time i)))
      ;; Evaluate a function of time, distance = (1/2)g.*t.^2
      ;; (g is the acceleration due to gravity)
      (withSlots (name) T
(mlEval "D = .5.*(-9.8).*T.^2;")
;;Plot the result
(mlEval "plot(T,D);")
(mlEval "title('Position vs. Time for a falling object');")
(mlEval "xlabel('Time (seconds)');")
(mlEval "ylabel('Position (meters)');")
(matrixDestroy T)))

;;;  * PART II
;;;  *
;;;  * For the second half of this demonstration, we will request
;;;  * a MATLAB string, which should define a variable X.  MATLAB
```

```
;;;  * will evaluate the string and create the variable.  We
;;;  * will then recover the variable, and determine its type.

;;;  * Use engOutputBuffer to capture MATLAB output, so we can
;;;  * echo it back.

    (let ((buffer (makeString BUFSIZE))
  (result 0))
      (engOutputBuffer *matlabhandle* buffer BUFSIZE)
      (do ()
((not (= result 0)) ())
 ;; * Get a string input from the user
 (format "Enter a MATLAB command to evaluate.  This command should~%")
 (format "create a variable X.  This program will then determine~%")
 (format "what kind of variable you created.~%")
 (format "For example: X = 1:5~%")
 (format ">> ")

 ;; * Evaluate input with engEvalString
 (mlEval (format #f "~a" (read)))

 ;; * Echo the output from the command.  First two characters are
 ;; * always the double prompt (>>).
 (format "~a" buffer)

 ;; * Get result of computation

 (format "~%Retrieving X...~%")
 (set! result (engGetArray *matlabhandle* "X"))
 (if (= result 0)
    (format "Oops! You didn't create a variable X.~%~%")
    (format "X is class ~a~%" (mxGetClassName result)))
(mxDestroyArray result)))))

;;; (engdemo '(0 1 2 3 4 5 6 7 8 9))
```

B.4.2 MATLAB Engine and MX functions

The Matlab interface consists of engine functions and MX functions. These functions are listed below for format. The definitins are in the MATLAB documentation.

Engine Functions

```
(eng-eval-string ep str)
(eng-open-single-use startcmd reserved retstat)
(eng-open startcmd)
(eng-close ep)
(eng-get-array ep name)
(eng-put-array ep ap)
(eng-output-buffer ep buffer, int buflen)
```

MX functions

```
(mx-set-alloc-listeners allochandler freehandler)
(mx-set-out-of-memory-listener outofmemoryhandler)
(mx-malloc n)
(mx-calloc n size)
(mx-free ptr)
(mx-realloc ptr size)
(mx-get-class-i-d pa)
(mx-get-name pa)
(mx-set-name pa s)
(mx-get-data pa)
(mx-set-data pa newdata)
(mx-get-pr pa)
(mx-set-pr pa pr)
(mx-get-imag-data pa)
(mx-set-imag-data pa newdata)
(mx-get-pi pa)
(mx-set-pi pa pi)
(mx-is-numeric pa)
(mx-is-cell pa)
(mx-is-char pa)
(mx-is-sparse pa)
(mx-is-struct pa)
(mx-is-complex pa)
(mx-is-double pa)
(mx-is-single pa)
(mx-is-logical pa)
(mx-is-int8 pa)
(mx-is-uint8 pa)
(mx-is-int16 pa)
(mx-is-uint16 pa)
(mx-is-int32 pa)
(mx-is-uint32 pa)
(mx-get-user-bits pa)
(mx-set-user-bits pa value)
(mx-get-scalar pa)
(mx-set-logical pa)
(mx-clear-logical pa)
(mx-is-from-global-w-s pa)
(mx-set-from-global-w-s pa global)
(mx-get-number-of-dimensions pa)
(mx-get-m pa)
(mx-set-m pa m)
(mx-get-n pa)
(mx-get-dimensions pa)
(mx-is-empty pa)
(mx-get-ir pa)
(mx-set-ir pa newir)
(mx-get-jc pa)
```

```
(mx-set-jc pa newjc)
(mx-get-nzmax pa)
(mx-set-nzmax pa nzmax)
(mx-get-number-of-elements pa)
(mx-get-element-size pa)
(mx-calc-single-subscriptpa nsubs subs)
(mx-get-number-of-fields pa)
(mx-get-cell pa i)
(mx-set-cell pa i value)
(mx-get-field-number pa name)
(mx-get-field-by-number pa i fieldnum)
(mx-set-field-by-number pa i fieldnum value)
(mx-get-field pa i fieldname)
(mx-set-field pa i fieldname value)
(mx-get-field-name-by-number pa n)
(mx-get-class-name pa)
(mx-is-class pa name)
(mx-set-n pa n)
(mx-set-dimensions pa size ndims)
(mx-destroy-array pa)
(mx-create-numeric-array ndim dims classid flag)
(mx-create-double-matrix m n flag)
(mx-create-sparse m n nzmax flag)
(mx-get-n-chars pa buf nChars)
(mx-get-string pa buf buflen)
(mx-array-to-string pa)
(mx-create-string-from-n-chars str n)
(mx-create-string str)
(mx-create-char-array ndim dims)
(mx-create-char-matrix-from-strings m str)
(mx-create-cell-matrix m n)
(mx-create-cell-array ndim dims)
(mx-create-struct-matrix m n nfields fieldnames)
(mx-create-struct-array ndim dims nfields fieldnames)
(mx-duplicate-array in)
(mx-set-class-name pa classname)
(mx-add-field pa fieldname)
(mx-remove-field pa field)
(mx-get-eps)
(mx-get-inf)
(mx-get-na-n)
(mx-is-finite x)
(mx-is-inf x)
(mx-is-na-n x)
(matlab-get-array-value pa n fnr)
        (matlab-set-array-value pa n fn)
```

Appendix C

Annotation Facility

C.1 Introduction

The annotation tool allows images to be manually segmented and labeled. The principle means of segmenting an image involves the use of a live-wire boundary snapping capability similar to that of Barrett and Mortensen (Barrett & Mortensen 1997).

The facility allows boundaries to be added, edited or deleted.

The results of annotation are written to a file in an extended SGML format so that they can be read and used easily by programs wishing to use the annotations.

The segmentation is subtractive and is implemented with a painter's algorithm. Consequently, the order of segmentation is important.

The format of the SGML file is as follows:

The annotations file contains a single annotation tag GTAnnotations with the following attributes:

- **imageName**. The name of the source image file that this file annotates.
- **author**. The name of the last person to update the annotations file.
- **creationDate**. The date and time that the annotation file was initially created.
- **modificationDate**. The date and time that the annotation file was last modified.

The GTannotations tag encapsulates all of the region annotations in the file. Each region is represented by a GTRegion tag which has the following attributes:

- **author**. The name of the person who created or last modified this region.
- **regionType**. An identifier (such as "R26") that identifies the labeling of the region. The mapping of region types to human readable names is specified in the region dictionary file.
- **regionUID**. A unique region identifier (such as "RGN42") that 'names' the region.
- **regiondate**. The date and time that the region was created or last modified.
- **coordinates**. A list of pairs of numbers that represent the coordinates of points along the boundary of the regions. The boundary of the region is defined to be the region contained within the region formed by drawing straight lines between these points.

  The natural representation of a boundary given the live-wire algorithm of (Barrett & Mortensen 1997) is a completely connected sequence of pixels, but this is cumbersome for large images and inefficient. The outline is reduced to a set of points with a line filtering algorithm as described in appendix C.

Here is an example SGML file for an image with two regions.

```
<GTAnnotations
  imageName="foo.tif"
  author="fred"
  creationDate="08/06/98 15:29:08"
  modificationDate="09/03/98 01:34:16">
<GTRegion
  author="fred"
  regionType="R20"
  regionUID="RGN40"
  regiondate="08/06/98 15:29:29"
  coordinates="1, 1, 509, 1, 509, 509, 1, 509, 1, 1">
</GTRegion>
<GTRegion
  author="fred"
  regionType="R26"
  regionUID="RGN42"
  regiondate="08/17/98 16:16:53"
  coordinates="42, 0, 37, 4, 35, 6, 36, 8, 32, 12, 27, 21, 28, 25,
  34, 31, 29, 37, 24, 41, 21, 41, 15, 37, 11, 35, 1, 37, 0, 16, 1, 1,
  3, 2, 4, 0, 26, 0, 31, 0, 35, 1, 36, 4, 36, 2, 37, 0, 43, 1">
</GTRegion>
</GTAnnotations>
```

C.2 Using the annotation facility

The annotation facility show in Figure C.1 is a Motif program that allows interactive labeling of images and editing of existing labels. The facility allows regions to be hand traced and for the regions to be assigned a label. The application screen is divided from the top into menu bar, image display pane, command input area, and message area. Commands can be entered in the command input area but most commands can be accessed through the menu bar and most interaction with the annotator is performed with the mouse.

C.3 Annotating Regions

The principle purpose of the annotator is to allow region to be defined on the image and to specify the region contents as a user defined token. Regions can be specified in three ways: Rectangular regions, polygonal regions, and live wire regions. These tracking modes can be selected from the options menu (see below).

Once a region has been specified it can be annotated by selecting a label from the annotate menu. To specify a region select the appropriate region mode from the potion menu, and then do the following as appropriate.

C.3.1 Rectangular Regions

To create a region in rectangular region mode, position the cursor at one corner, then holding down the middle mouse button drag out a rectangle as desired.

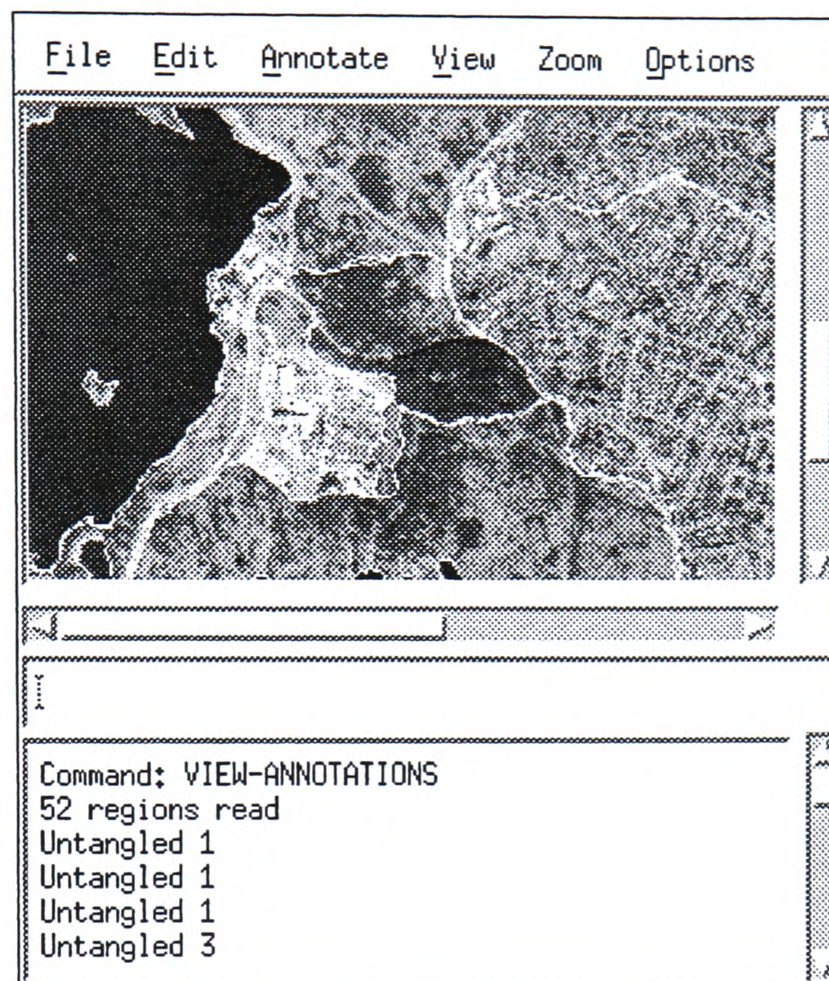


Figure C.1: The annotation facility

C.3.2 Polygonal Regions

To create a region in polygon region mode, position the cursor at a start position, then holding down the middle mouse button drag out a rubber band line. Release the middle mouse button to accept the rubber band line and then again depress the middle mouse button to drag out the next segment of the polygonal region. Repeat until all but the last segment of the polygon has been specified in this way. Finally click the right mouse button to close the polygon.

C.3.3 Live Wire Regions

To create a region in live wire region mode, position the cursor at a start position, then holding down the middle mouse button drag out a live wire line by following the contour in the image. Occasionally a new seed point will be set automatically to allow the region tracing to continue. To manually set a new seed point, release the middle button and again depress to middle button and continue tracing. Finally to close the region release the middle button and click the right mouse button to close.

C.4 Region editing

Once a region has been specified and annotated it is often desirable to make adjustments to it. This can be done for the polygon and the live wire regions. In both cases, a region is made up of a sequence of segments. In the case of a polygonal region the segments consist of straight lines whereas in the case of live-wire regions, the segments are arbitrarily shaped segments that track edges in the image. In either case the region can be edited by selecting the region and

then using the "edit selected region" command from the edit menu bar menu. These commands are described below in the menu command and keyboard command sections. The starting point for the edit can be selected by holding the right mouse button down while moving around the border. The segments of the outline will be XOR'd to indicate the position. When the mouse right button is released a small cross will indicate the position on the outline. When the "edit selected region" command is invoked the editing cursor will be placed at that location on the curve. Editing an outline does not change the label associated with the region. To change the label of a region, select the region with a left click of the mouse near the enclosing rectangle of the region and then use the "annotate command" to select a different label from the one currently in effect. The annotation will be modified to the new label.

C.5 Menu Commands

The annotation facility supports the following menu commands. The menu commands are available from the pull down menu bar. They are divided into six top level menu bar items. All of the commands in the menu bar are also available as command line commands.

C.5.1 File

These commands deal with opening and closing image files and annotations.

- Open

  Pops up a standard Motif file select menu. The image to be annotated should be selected. If an image is already being annotated, it will be closed automatically before opening the new image. The image is read in to the annotator and displayed in the image portion of the tool. If the file has already been annotated (there is a file with the same name but extension .SGML in the same directory), the annotation file is also read in.

  This command is available as a command line command as "open $<$ *filename* $>$".

- New

  Deletes any existing annotations associated with the image that has been opened. This is useful for starting the annotation process over for a particular image.

- Close

  Closes the currently open image (if any). Any unsaved annotations will be written to disk before the close operation completes.

  This is available as a command line command as "close"

- Exit

  Exit from the annotator facility. If there is an existing open image it will be closed along with the saving of unwritten annotations before the facility exits.

  This is available as a command line command as "exit".

C.5.2 Edit

- Select All

  If a region is currently selected it will be replaced with a region that includes the entire image. If no region is selected, a new region will be created that includes the entire image. This is useful as an initial annotation from which subsequent regions will be subtracted. For example if most of the image contains fields it is convenient to start off by making

the entire image a single region annotated as "fields" and then to subtract the non field regions with subsequent regions.

This is available as a command line command as "select-all".

- Edit Selected

  When a region is selected, this causes the region to enter "edit mode" so that the region boundary can be modified. See the section on editing existing regions.

  This is available as a command line command as "edit-selected".

- Update Annotation

  When in region edit mode, this causes the region being edited to be captured as the new region shape. This will cause the old definition of the region shape to be discarded. A region edit can be abandoned up to this point by using the escape key (see keyboard commands) instead of using the Update Annotation command.

  This is available as a command line command as "update-annotation".

C.5.3 Annotate

The annotate menu is constructed dynamically from the annotation mapping SGML file read from a location specified in the 'GRAVADICTIONARY' environment variable.

The annotation mapping file consists of SGML tags that specify region type identifiers, human readable names for the identifiers, and the layout of the annotation menu bar pull down menu.

Two SGML tags are defined for the annotation mapping file. The first is ImageClass. ImageClass has a single attribute 'name' that specifies a sub-menu categorization in the annotate pull down menu. It has no semantic import. It is simply an organizational device for categorizing the region types in the annotate pull down menu. The second SGML tag is RegionClass which has two attributes: 'name', and 'id'. The 'name' attribute specifies a human readable name for the region type. The 'id' attribute specifies the name of the region type. It is the 'id' name that is embedded within the annotation files that are generated during the annotation process. The annotation mapping file is necessary to convert region type ID's back to human readable names. Below is an example annotation mapping file for the example that is used throughout this paper.

```
<ImageClass name="Example">
<RegionClass name="Road" id="R1"></RegionClass>
<RegionClass name="River" id="R2"></RegionClass>
<RegionClass name="Field" id="R3"></RegionClass>
<RegionClass name="Swamp" id="R4"></RegionClass>
<RegionClass name="Lake" id="R5"></RegionClass>
<RegionClass name="Town" id="R6"></RegionClass>
</ImageClass>
<ImageClass name="Face">
<RegionClass name="Eye" id="R7"></RegionClass>
<RegionClass name="Nose" id="R8"></RegionClass>
<RegionClass name="Mouth" id="R9"></RegionClass>
<RegionClass name="Eyebrow" id="R10"></RegionClass>
<RegionClass name="Chin" id="R11"></RegionClass>
<RegionClass name="Hair" id="R12"></RegionClass>
<RegionClass name="Beard" id="R13"></RegionClass>
<RegionClass name="Moustache" id="R14"></RegionClass>
<RegionClass name="Ear" id="R15"></RegionClass>
</ImageClass>
```

Each ImageClass tag generates a submenu in the annotation menu with the RegionClass tags contained as sub items. This allows region types to be categorized into easy to find submenus. In complex images where there are many different region types to choose from this is essential.

The region ID's must be unique although RegionClass tags may be duplicated in different submenus.

This as available as a command line command as "annotate-region $< regionID >$".

C.5.4 View

- Pixels [Default]

  Causes the image to be displayed in the current color mode (see options commands).

  This is available as a command line command as "view-pixels".

- Annotations

  Same as Pixels but overlays the image with the region borders in the current outline color (see options commands).

  This is available as a command line command as "view-annotations".

- Edges (⇒DX, ⇒DY, ⇒Combined)

  Displays the result of convolving the image with the Sobel (Sobel 1974) operator. DX shows the result of the DX operator, DY shows the result of the DY operator and Combined shows the edge map where edge magnitude is mapped onto intensity and edge orientation is mapped onto color. The Sobel operator is applied automatically when region mode is set to "Live Wire" (see options command). The live wire tracker uses edge information in order to track edges.

  These are available as command line commands as "view-edges-dx", "view-edges-dy", and "view-edges-combined".

- Segmentation (⇒Map, ⇒Borders)

  Allows the current segmentation as defined by the annotated regions to be displayed.

  'Map' displays the regions of the image as a patchwork of colored regions where different region types are automatically allocated a different color.

  'Borders' displays the image pixels overlaid with the borders of the segmented regions in the current outline color. This is different from view-annotations which shows the annotated region boundaries. The borders shown here are the borders of the regions that result from applying the subtractive semantics to the annotation regions.

  These are available as command line commands as "view-segmentation-map", and "view-segmentation-borders".

C.5.5 Zoom

The zoom commands change the scale at which the image is displayed. The size of the image can be reduced by a Derez operation that operates in integral pixels. So Zooming out will reduce the image size to 1/4 of its size each time it is applied.

- Out

  Zoom the image out making it 1/4 of its current size.

  This is available as a command line command as "zoom-out".

- In

  Zoom the image in making it 4 times its current size. An image cannot be made bigger than its original size.

  This is available as a command line command as "zoom-in".

- To Fit

  Choose a good zoom level for viewing the image. Zoom to fit is automatically performed when a new image is loaded with the "File⇒Open" menu command.

  This is available as a command line command as "zoom-to-fit".

- None

  Display the image in its natural resolution.

  This is available as a command line command as "zoom-none".

C.5.6 Option

Various options control the operation of the annotator.

- Color Mode (⇒Color, ⇒Dither Color, ⇒Gray)

  Choose a color model. This may be needed if the X display has limited color depth.

  'Color' displays the image in true color and requires true color or pseudo color support from the X display.

  'Dither Color' displays the color image by using a 256 color palate and dithering. It presents a good color rendering on X displays that cannot provide Truecolor or Pseudocolor modes.

  'Gray' displays the image as a gray image using a 256 palate of gray scale. Sometimes when annotating a color image on an X display that cannot provide true color or pseudo color it is better to use 'Gray'. Gray scale images also look better in 'Gray mode' than in 'Dither Color' mode because all of the palate entries can be used for gray levels.

  These are available as command line commands as "options-color", "options-dither-color", and "options-gray".

- Region Mode (⇒Rectangular, ⇒Polygon, ⇒Live Wire) [Default=Rectangular]

  Selects the mode of region drawing. When 'Live Wire' is selected the Sobel operator is convolved with the image to provide the underlying support for the live wire tracker.

  These are available as command line commands as "options-region-mode-rectangular", "options-region-mode-polygon", and "options-region-mode-live-wire".

- Outline Color (⇒Red, ⇒Green, ⇒Blue, ⇒Yellow, ⇒Cyan, ⇒Magenta, ⇒White) [Default=White]

  Sometimes it is hard to see the region boundaries against the image clutter. Often in such situations it helps to change the outline color. Yellow seems to work well on color infrared satellite images.

  These are available as command line commands as "options-outline-color < *color* >".

- Annotation Mode (⇒Update Segmentation, ⇒Don't Update Segmentation) [Default=Don't Update Segmentation]

  In order to be able to view the segmentation map and borders it is necessary to perform region subtraction on the annotation regions. Since the subtraction operation is computationally intensive it is disabled by default so that annotation speed is not impeded.

  These are available as command line commands as "options-annotate-update-segmap", and "options-annotate-not-update-segmap".

- Tangle Mode (⇒Untangle Outlines, ⇒Don't Untangle Outlines) [Default=Untangle Outlines]

  Sometimes when making an outline the boundary is made to cross itself. A boundary untangling algorithm uncrosses crossed line segments to ensure a valid region boundary results. This can be disabled with the "Don't Untangle outlines" command.

  These are available as command line commands as "options-untangle", and "options-dont-untangle".

C.5.7 Keyboard Commands

The annotation facility supports the following keyboard commands for navigating and manipulating a region.

- Back space. If a region is being edited either as a live wire or polygon region whether as a new region or editing an existing region, backspace causes the previous segment to be deleted and the position moved back to the previous seed point. In the case of a polygon region, the previous straight line segment is deleted. In the case of live wire tracking, the segment up to the previous seed point is removed. The previous seed point may have been explicitly set by the user or may have been an automatic seed point.
- Escape. Escape means cancel the current section and operation. If a region is being created it is abandoned. If a region is being edited, the edit operation is abandoned. If a region is selected (and highlighted), the selection is canceled.
- Return. Return closes an open region during region creation or editing. It is equivalent to the mouse right click close operation.
- Delete. The delete key will delete a selected region. To delete a region, select the region and then hit the delete key.
- Cursor Left. If a region is selected, cursor left causes the selection to move to the previous region. If no region is selected this causes the first region to become selected.
- Cursor Right. If a region is selected, cursor right causes the section to move to the next region. If no region is selected this causes the last region to become selected.
- Cursor Up. If a region is selected this causes the selected segment within the region to be advanced. This along with cursor down (below) allows the editing position in a region to be carefully positioned.
- Cursor Down. If a region is selected this causes the selected segment within the region to be retreated.

Appendix D

Line Vectorization

D.1 Introduction

In several places, we encounter situations where we are presented with a representation of a boundary as a completely connected sequence of pixels (if you do a flood fill inside the boundary, it will not leak). Often this representation is unwieldy and inappropriate. Two examples of where we are faced with such a representation in the current work are:

1. The seeded region growing algorithm of Zhu and Yuille (Zhu & Yuille 1996).
2. The outlines generated in the image annotator by the shortest path algorithm of Barrett and Mortensen (Barrett & Mortensen 1997).

This algorithm considers a boundary to consist of straight line segments in which the endpoints of the line segments fall on pixel boundaries. A completely connected sequence of pixels is an example of such a representation. If we have a section of the boundary that is a straight line represented by the sequence of pixels that constitute the line we can discard the internal pixels without loss of information because the omitted pixels can be recovered by drawing straight line segments between the endpoints.

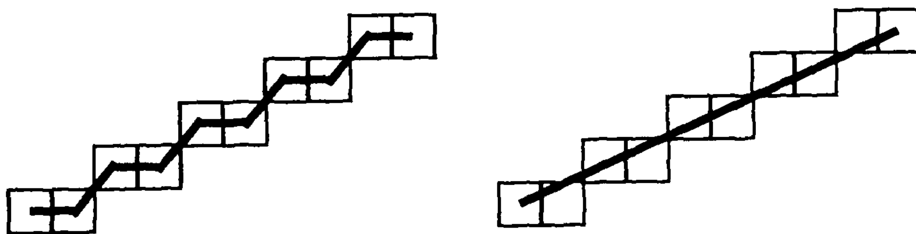


Figure D.1: Converting pixel lists to vectors

Figure D.1 shows how a straight line is represented as a sequence of pixels. In converting pixels to line segments we have to ignore small deviations from the pixel centers to the hypothetical center of the line. In most cases these deviations are artifacts introduced by the pixel size. Ignoring small deviations from the centerline does not lose useful information.

The line filter attempts to find line sequences such that the perpendicular distance of the center of each (omitted) pixel from the line is less than ϵ. In simplifying outlines we wish to retain points of salience while discarding uninteresting detail. This algorithm makes the assumption that points of high curvature are important so it attempts to position end points of lines on points of high curvature.

Figure D.2 shows the epsilon's for two pixels ϵ_1, and ϵ_1. By setting the epsilon parameter to 0.5 pixels (the default) no shape information is lost.

The algorithm depends upon being able to compute the perpendicular distance between a hypothesized line and all of the pixels in the path. It also depends upon being able to estimate

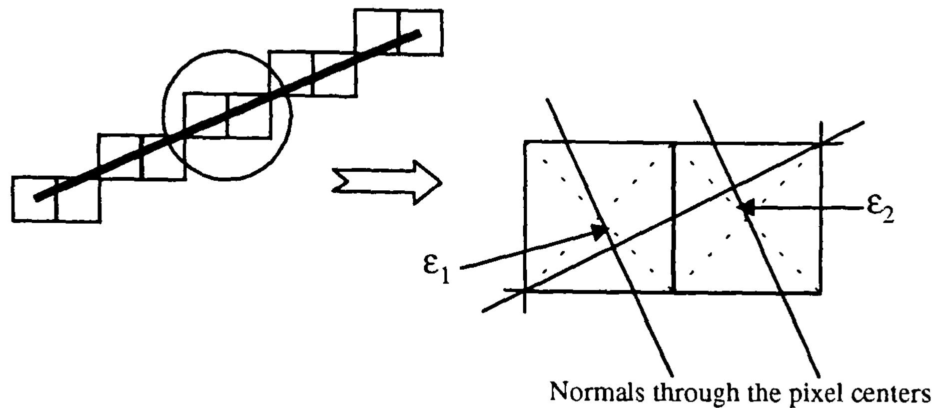


Figure D.2: The line filter's epsilon parameter

curvature at every point along the pixel sequence. Both requirements are facilitated by representing the hypothesized line in terms of the unit normal $\hat{n}$ and the perpendicular distance of the origin to the line d. Given two points A and B connected by a line, the unit normal to $\vec{AB}$ $\hat{n}$.

$$\hat{n} = \frac{\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \vec{AB}}{AB}, \theta = \frac{\pi}{2} \tag{D.1}$$

Hence,

$$\hat{n} = \frac{\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \vec{AB}}{AB} \tag{D.2}$$

and the distance of the origin from the extended line that connects A and B is the same as the perpendicular distance of an arbitrary point R on the extended line that passes through A and B and the with direction $\vec{AB}$ which passes through the origin. By choosing R=A we get

$$d = \vec{OA} \cdot \hat{n} \tag{D.3}$$

```
;;; Macro to execute 'body' in an environment in which (nx ny) is the unit normal
;;; and d is the perpendicular distance from the origin to the line that passes
;;; through points (sx sy) (ex ey).

(defineMacro withLineInNormalForm ((nx ny d) sx sy ex ey . body)
  (let ((xdist (gensym))
        (ydist (gensym))
        (norm (gensym)))
    `(let ((,xdist (- startx endx))
           (,ydist (- starty endy))
           (,norm (sqrt (+ (* xdist xdist) (* ydist ydist)))))
       ,@body)))
```

The distance between any point P that makes a perpendicular intersection at Q with the extended line that passes through A and B is therefore

$$PQ = \vec{OP} \cdot \hat{n} - d \tag{D.4}$$

```
(define pointDistanceFromLine
  (lambda (nx ny d px py)
    (abs (- (+ (* px nx) (* py ny)) d))))
```

There are many approaches to estimating curvature of a pixel sequence such as by fitting a curve to the points, computing the curvature of the curve and then mapping those values back to the original pixels. The algorithm employed in this implementation uses the three neighboring pixels on either side of each pixel to provide an estimate of the angle of turn and hence provides an estimate of the curvature. Three pixels each side of the pixel in question are used so as to smooth the effect of pixelization of the boundary while retaining the ability to detect local points of high curvature. The algorithm has proven to be effective on complex shapes such as those found in regions of aerial images.

In principle any method of computing curvature could be substituted for the one described here without affecting the algorithm.

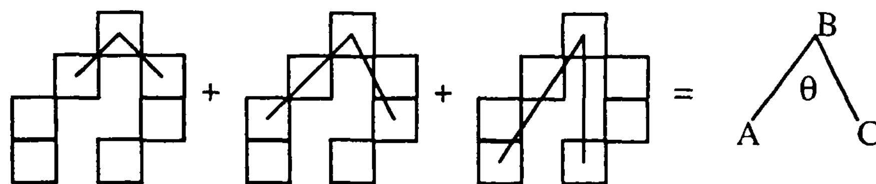


Figure D.3: Estimating local curvature

Figure D.3 shows how three estimates of local curvature for a pixel are combined to reduce the effects of discretization of the path into pixels.

The angle at a pixel is taken to be the average of three estimates:

1. the nearest neighbors of the pixel,
2. the two pixels that are two path pixels away, and
3. the two pixels that are three path pixels away.

Combining these values gives us the angle θ formed from the two lines $\vec{AB}$ and $\vec{CB}$. If the unit normals of the lines $\vec{AB}$ and $\vec{CB}$ are $\hat{n}_1$ and $\hat{n}_2$ respectively the angle between the normals ϕ is given by:

$$\cos(\phi) = \frac{\hat{n}_1 \cdot \hat{n}_2}{n_1 n_2} = \hat{n}_1 \cdot \hat{n}_2 \tag{D.5}$$

The angles between the normals is the same as the angles between the vectors, so $\phi = \theta$, $\theta = acos(\hat{n}_1 \cdot \hat{n}_2)$, and the curvature k is $\pi/\theta - 1$

$$k = \frac{\pi}{acos(\hat{n}_1 \cdot \hat{n}_2)} - 1 \tag{D.6}$$

For a straight line segment, the angle θ will be π radians yielding a curvature of zero. When θ is zero, the curvature estimate would be infinity but this could only occur if the boundary had the identical preceding and following pixels to a depth of three on both sides. Without sharing pixels, the highest curvature that can be produced by this algorithm is $\pi/acos(2/\sqrt(5)) - 1 = 5.7758$.

1. Vector $\vec{b}$ is the boundary as a completely connected sequence of pixels.
2. Compute the vector $\vec{k}$ the curvature at each point along $\vec{b}$ using the algorithm described above.

3. ϵ is a parameter that controls the maximum error in pixels.
4. Find the point of maximum curvature and push it onto the result vector. Mark the point as used. Also set the target coordinates to be the coordinates of this points neighbor.
5. Starting from the point just pushed onto the result vector, test each coordinate in turn and compute the distance of the point from the imaginary line from the source to the target coordinate. If all distances are less than ϵ push the target coordinate onto the result vector and return. The result vector contains the filtered boundary. If a point is found that is at a greater distance from the imaginary line than ϵ, go to the next step.
6. Find the point between the start point and the target point with the highest product of distance from the line and curvature. This causes a point to be selected that has relatively high curvature and a significant displacement from the line. When filtering pixel sequences with homogeneous curvature – such as the outline of a circle, this causes points of maximum deviation to be chosen. Otherwise a point of high curvature close to the point of maximum deviation will be chosen.
7. Divide the line segment into two segments at this point and recursively filter the line.

This algorithm preserves points of highest curvature while discarding points that deviate less than ϵ from the straight line between points.

The line filter can be used in two distinct ways.

1. By using a small value of ϵ (such as 0.5), a compact representation of the original boundary can be produced with little or no loss of detail.
2. By using a large value of ϵ, a set of anchor points can be produced that are suitable for seeding a snake (Kass, Witkin, & Terzopoulos 1987)

```
;;; Macro to reference the pos entry in the vector taken as a
;;; circular vector
(defineMacro cRef (vect pos)
  `(vectorRef vect (mod pos (vectorLength vect))))

;;; Macro to set the pos entry in the vector take as a circular vector.
(defineMacro cSet! (vect pos val)
  `(vectorSet! vect (mod pos (vectorLength vect)) val))

;;; Returns #t if all points between start and end have an absolute
;;; perpendicular distance of less than epsilon, #f otherwise.

(define noMajorDeviationsFromLine
  (lambda (xv yv start end epsilon)
    (let ((endx (cRef xv end))
          (endy (cRef yv end))
          (startx (cRef xv start))
          (starty (cRef yv start)))
      (withLineInNormalForm (nx ny d) startx starty endx endy
        (do ((pos start (+ pos 1)))
            ((or (> pos end)
                 (> (pointDistanceFromLine
                      nx ny d
                      (cRef xv pos)
```

```
                    (cRef yv pos))
                  epsilon))
             (> pos end)))))))

;;; returns the endpoint of the next line segment that guarantees
;;; that all intermediate points have an absolute perpendicular
;;; distance < epsilon from the line (start end) and which is
;;; chosen to be a point of high curvature.

(define findFilteredRegionEndpoint
  (lambda (xv yv curvature start end epsilon)
    (if (= start (mod end (vectorLength xv))) (set! end (- end 1)))
    (while (or (> end start)
               (not (noMajorDeviationsFromLine xv yv start end epsilon)))
           ;; The line needs to be subdivided.  Find the maximum
           ;; product of the curvature and the
           ;; distace from the line.
           (let ((endx (cRef xv end))
                 (endy (cRef yv end))
                 (startx (cRef xv start))
                 (starty (cRef yv start)))
             (withLineInNormalForm (nx ny d) startx starty endx endy
               (let ((maxdist -1) (maxpos -1))
                 (do ((pos (+ start 1) (+ pos 1)))
                     ((>= pos end) (set! end maxpos))
                     (let* ((dev (pointDistanceFromLine
                                    nx ny d (cRef xv pos) (cRef yv pos)))
                            (pcd (* (cRef curvature pos) dev)))
                       (when (> pcd maxdist)
                         (set! maxdist pcd)
                         (set! maxpos pos)))))))))
    end))

;;; Takes a sequence of pixels (xv yv) and epsilon the maximum deviation
;;; parameter and returns the filtered sequence of pixels (newx newy).

(define filterRegion
  (lambda (xv yv epsilon)
    (let* ((numpoints (vectorLength xv))
           (start 0)
           (end numpoints)
           (newx ()) (newy ()) (maxk -1) (maxpos -1)
           (curv (makeVector numpoints))
           (curvature (makeVector numpoints))
           (cpyx (makeVector numpoints))
           (cpyy (makeVector numpoints))))
    ;; Compute curvature
    (do ((pos 3 (+ pos 1)))
        ((>= pos (+ numpoints 3)) #f)
        (let ((xb (/ (+ (cRef xv (- pos 3)) (cRef xv (- pos 2))
```

```
                    (cRef xv (- pos 1))) 3))
            (yb (/ (+ (cRef yv (- pos 3)) (cRef yv (- pos 2))
                    (cRef yv (- pos 1))) 3))
            (xa (/ (+ (cRef xv (+ pos 3)) (cRef xv (+ pos 2))
                    (cRef xv (+ pos 1))) 3))
            (ya (/ (+ (cRef yv (+ pos 3)) (cRef yv (+ pos 2))
                    (cRef yv (+ pos 1))) 3))
            (xo (cRef xv pos))
            (yo (cRef yv pos)))
        (withLineInNormalForm (n1x n1y d1) xb yb xo yo
          (withLineInNormalForm (n2x n2y d2) xa ya xo yo
            (let ((k (- (/ pi (acos (+ (* n1x n2x) (* n1y n2y)))) 1)))
              (cSet! curvature pos k)
              (when (> k maxk)
                (set! maxk k)
                (set! maxpos (mod pos numpoints))))))))
;; Order with max curvature first.
(dotimes (pos numpoints)
  (cSet! cpyx pos (cRef xv (+ pos maxpos)))
  (cSet! cpyy pos (cRef yv (+ pos maxpos)))
  (cSet! curv pos (cRef curvature (+ pos maxpos))))
;; now point of greatest curvature is at position 0.
(push! (cRef cpyx 0) newx)
(push! (cRef cpyy 0) newy)
(while (< start end)
       (set! start (findFilteredRegionEndpoint
                     cpyx cpyy curv start end epsilon))
       (push! (cRef cpyx start) newx)
       (push! (cRef cpyy start) newy))
(list (list->vector (nreverse newx))
                    (list->vector (nreverse newy)))))
```

Appendix E

Yolambda and Lisp notation

E.1 Introduction

Expressions are forms that return a value. We cover these basic forms first to ensure the basis for understanding examples and descriptions of functionality in later chapters. During the course of explaining the various types of expressions, we will also be exploring several concepts that are crucial to programming in general, and to programming in Yolambda in particular. These concepts include naming, binding, accessing values, quoting, calling, control, sequencing and branching. Naming is the act of using a symbol to refer to an entity, or object. Naming is crucial to making programs that are readable and understandable to humans. A binding is a linking, temporarily or permanently, of a name to a storage location (usually in memory). This allows one to later refer to that location, and indirectly refer to its contents, in a more convenient manner. This in turn allows one to access (get or set) the value stored in a named location. So we will see descriptions of expressions that allow us to permanently or temporarily bind a name to a storage location, and then use that name to refer to the location or indirectly to the value stored in that location. Usually, context will determine whether we are using a name to refer to a location, or the value stored there. For example, when we want to get a value, the name of the location can be used to refer to that value. When we want to set a value, then we use the name of the location to refer to the location where the value will be stored. Once we have a way of using names to refer to locations or the values stored in locations, for completeness, we need a way to refer to the names themselves. This is the purpose of expressions that support quoting. Calling or invoking is the act of preparing for a computation, and then causing the instruction stream to be diverted to the location where the instructions for the computation are stored. Typically, preparing involves evaluating and storing arguments, setting up storage for local variables and return values, and storing a return address (the location that the instruction stream should revert to when the current computation is complete). Most of the computation in Yolambda is accomplished by calling procedures. The concept of control deals with how instructions, computations and subroutines are ordered. Such ordering can be implicit in the semantics of language forms, or explicit, in the form of expressions whose main task is ordering flow of control. Sequencing and conditional branching are among the simplest control flow expressions. At the end of this appendix, we take up the related topic of generalized variables in Yolambda, which allow a broadening of the concepts of naming and setting to include forms other than just variables, and the topic of equivalence predicates.

E.2 Expressions

A Yolambda program is a sequence of imperative, value returning expressions. These expressions include variable references, literal expressions, procedure calls, lambda expressions and conditionals. Some additional expression types, included for convenience, expressiveness, and readability, include definitions, binding constructs, derived conditionals, sequencing and iteration constructs, and quasiquotation.

In practice, a Yolambda program is a sequence of binding (or naming) expressions, which assign values to variables, followed by a sequence of procedure calls. The bindings are basically definitions, which give data values to some variables and procedural values to others. The procedure call expressions are sequences of Yolambda expressions, which have a specific semantics. The first element of the sequence must evaluate to a procedure object, and later expressions return values which are passed as arguments to the procedure object. The initial element of the procedure call can either be a variable which names a procedure object, or a lambda expression, which is a kind of anonymous procedure.

The procedure call sequences are represented in Yolambda programs as lists. A list is a parenthesized sequence of elements. List structure is uniformly used to represent programs in Yolambda. The following is a trivial example of a Yolambda program, illustrating some of these ideas:

```
(define red 128)
(define green 64)
(define blue 144)
(define (computeColor r g b)
  (+ r (* 256 g) (* 256 256 b)))
(computeColor red green blue)
```

In the above program, there are 5 top-level lists. The first three define variables which name colors, and have assigned specific integer values. The fourth form assigns a procedure object to a variable, where the body of the procedure assembles three parameters into a single number, using multiplication and addition. The final list is a procedure call, whose first element is the variable that was defined as a procedure, and whose remaining elements are arguments to the procedure call. Each of these expression types will be discussed in more detail next.

E.3 Procedure Calls

The code for invoking procedures is central to the task of programming with a procedural language like Yolambda. The topic of procedure calls includes both named procedures (whether primitive, derived or defined by the programmer) and anonymous procedures called Lambda expressions.

(operator operand ...) [Syntax]

A procedure call is a list whose first element is an expression (usually the name of the procedure to be called) which returns a procedure object. The remainder of the list is the sequence of arguments for the procedure. The arguments and the operator expression are evaluated, and the values of the argument expressions are passed to the value of the operator expression. The order of evaluation is unspecified, and should not be relied upon. Procedure calls are how one causes Yolambda to make computations, and hence they are a very important class of expressions. Flow of control is managed by explicit sequencing of procedure calls, or by nesting calls within definitions of calls, or as expressions that compute arguments to procedure calls.

Examples:

```
(size '(1 2 3)) → 3
((lambda (x y) (or (and x (not y)) (and y (not x)))) #t #f)
→ #t
```

The next example is much like the first, except that a call to a procedure which makes lists is the argument to the size procedure: (size (list 1 2 3)) → 3 Variable References variable [Syntax]

A variable reference is an expression construct consisting solely of a variable. Conceptually, a variable is an object with a location and a value. A variable is bound to a location, and has as its value the contents of the location. The value returned by a variable reference is the value stored at the bound location. It is an error to refer to an unbound variable. Variables allow one to name abstractions, and separately associate values with the named abstractions. The value associated with the name can change during the course of computations, and be used to represent a (partial) state of the computation.
Examples: In the following examples, variables are defined and then referenced:

```
(define versionNumber 12)
versionNumber  →  12

(define *register* 1)
*register*  →  1
```

(the typeName variable) [Syntax] typeName—variable [Syntax]

A typed variable reference is a form containing the, followed by the name of a class, followed by the name of a variable. This form restricts the range of values for variable. The second form listed above is an abbreviation for the first form, and is most often used in Yolambda code. A typed variable can be used anywhere that an untyped variable can be used. In Yolambda, all type names are the names of classes.
Examples:

```
In the following examples,  variables are defined and then referenced:
(define (the Integer versionNumber) 12)
versionNumber  →  12

(define MemoryLocation|*register* 1)
*register*  →  1
```

E.4 Lambda Expressions

Lambda expressions are the primary means of specifying procedures, as well as the primary binding construct. A lambda expression can be thought of as an anonymous procedure. The define construct can be used to provide a name for the procedure specified by the lambda expression. The semantics of the lambda expression are the foundation for much of Yolambda coding constructs. The binding constructs, such as let, are defined in terms of lambda, for example.

(lambda formals body) [Syntax]

The value returned by a lambda expression is the procedure object which the lambda expression specifies. The environment in effect at the time of the evaluation of the lambda expression is retained with the procedure, and extended by binding the formal parameters at the time of a procedure call. During a procedure call, the formal parameters are bound to locations, and the procedure's arguments are stored in those locations. Then the expressions in the body of the lambda expression will be evaluated, and the result returned. The following description is of the form of formals.

(variableOrTypedVariable ...)

Formals are a parenthesized list of variables, either plain or typed variables.
Examples:

```
(lambda (boolean|x boolean|y)
  (or (and x (not y))
```

```
    (and y (not x))))

(define equivalent
  (lambda (a b)
    (or (and a b)
        (and (not a)(not b)))))

(define printPair
  (lambda (pair|object port|outport)
    (format outport
            "(~a . ~a)"
            (car object)
            (cdr object))))
```

E.5 Literal Expressions

Literal expressions include both constants, such as numbers or character strings, and quoted representations of symbols, lists or other Yolambda objects.

(quote datum) or 'datum [Syntax]

A literal expression is a quoted representation of a Yolambda object. Expressions may be quoted by prepending a quote sign, or by being the second item of a list whose first item is the symbol "quote". Examples:

```
'()
(quote name)
'(this is a quoted list)
```

datum: [Syntax] A symbol followed by a ":" is another representation of a quoted symbol. This form is typically used to denote keywords, in keyword value pairs, such as in some arguments to a procedure.
Examples:

```
(defineHost foo directory: bar)
```

constant [Syntax]

A constant is a number, character, string, or boolean value that doesn't change its value (as opposed to variables). The boolean constants are #t and #f for true and false, respectively. #f is the only false value a predicate can return. Nil is the constant that represents the empty value, for example, the empty list.
Examples:

```
#
A
"The string of chars"
#t
3.14159
27
```

E.6 Conditionals

The primary conditional expression is the if expression. Alternative derived conditionals include cond, case, and, and or. In Yolambda, the order of evaluation of arguments to a procedure is undetermined. But conditional expressions have their constituent forms evaluated in a determined,

left-to-right order. This makes them useful for guaranteeing argument evaluation order, as well as for evaluating expressions only when a guard expression is satisfied.

(if test consequent [alternative]) [Syntax]

The conditional expression, denoted by the reserved word if in the first position of the list, includes at least two additional expressions, and an optional third. The test expression is evaluated first, and its value is determined. That value is false just in case the returned value is #f, otherwise the returned value is considered true. If test yields a true value, then the consequent is evaluated and its result is returned as the value of the conditional. If test is false, and an alternative expression is specified, the result of the conditional is the result of evaluating alternative. If the test expression is false, then if no alternative expression is specified, the result of the conditional is unspecified.
Examples:

```
(if (pair? x) 42)
(if (eq? x y) 'same 'different)
```

(cond (test action) ...) [Syntax]

The cond expression implements a conditional with a series of tests, each of which has an associated action. The test, action list are together called a clause. Tests are evaluated in order only until one evaluates to true. Only one action is evaluated, the one associated with the first test which succeeds. In the most usual case, each clause in the cond expression is a list whose first element is the test or guard, and whose action is a series of expressions. The result of the last expression in the action list is returned as the value of the cond expression. However, there are special cases: when no test evaluates to true, cond returns #f; if the last clause has the reserved word else in place of test, then if no previous test is true, the action associated with else is executed; and if there are no expressions following the first true test, then the value of that test is returned. A final special case involves special syntax for the action, as follows: expression where expression evaluates to a procedure with one parameter. If the test associated with this action is the first one to evaluate to true, then the procedure value of expression is invoked with the value of test as its argument. The result of invoking this procedure is then returned as the value of the cond expression.
Examples:

```
(cond
  ((> (weight 'john) 175)
   (heartTest 'john overweight: #t))
  ((> (cholesterol 'john) 200)
   (triGTest 'john 200)
   (heartTest 'john highCholesterol: #t))
  ((memq 'john typeAList)  heartTest)
  (else
    (push! 'john typeBList)
    (certifyHeart 'john)))
```

(case key clause1 clause2 ...) [Syntax]

The case expression implements a conditional where the expressions of the first clause which matches a key are executed. The key to be matched is the result of evaluating the key field of the case expression. The clauses have the following syntax:

```
((item ...) expression ...)
(else expression ...)
```

A match is found if the result of evaluating key is eqv? to any of the unevauated items of the first element of a clause. When a match is found the remaining expressions of clause are evaluated, in order, and the result of the last of these is returned as the value of the case expression. If there are no matches, then an else clause is executed if one was supplied. If there are no matches, and no else clause was supplied, the result of the case expression is unspecified.
Examples:

```
(case (diagnosis 'john)
  ((arhythmia highBloodPressure angina)
   (set! doctor 'heartSpecialist))
  ((emphysema tuberculosis)
   (set! tester 'radiologist)
   (set! doctor 'lungSpecialist))
  (else
    (set! doctor 'generalPractitioner)))
```

(and expression ...) [Syntax]

The and expression implements the conditional which returns false unless all of the component conjuncts are true. Each of the expressions in the and expression are evaluated in turn until one is false, at which point #f is returned. If no constituent expression is false, the value of the last constituent expression is returned.
SeeAlso: if, or
Examples: (and (even? num) (¿ num 2) 'notPrime)

(or expression ...) [Syntax]

The or expression implements the conditional which returns true unless all of the component disjuncts are false. Each of the expressions in the or expression are evaluated in turn until one is true, at which point #t is returned. If no constituent expression is true, #f is returned.
Examples:

```
(or (memq 'john typeAList)
    (highBp? 'john)
    (eqv doctor 'heartSpecialist))
```

E.7 Assignment

(set! variable expression) [Syntax]

The assignment expression evaluates the second expression argument and stores the resulting value in the location to which variable is bound, in the current lexical or global environment. The result of the assignment expression is unspecified.
Examples:

```
(set! fruits #(apple orange prune))
```

E.8 Definitions

In Yolambda, definitions may occur at the top level of programs, and also at the beginning of the body portions of some syntactic expressions. Simple definitions are simply shorthand for the binding of a value to a variable, to support a later reference to the variable (or call of the procedure if the value was a procedure object) to which the variable is bound. Definitions are also used to introduce a polymorphic procedure (a procedure whose behavior varies with the types of arguments it receives) or add a method to an existing polymorphic procedure.

```
(define variable expression) [Syntax]
(define (variable formals) body) [Syntax]
(define (variable formals . formal) body) [Syntax]
```

In the first form of define above, the environment is extended by adding the name: variable, binding variable to a location, and storing the value of expression in that location. The following diagram illustrates this idea:

Defining: binding name to location, storing value. If the expression is a lambda expression, as in the following: (define variable (lambda (formals) body)) where formals stands for a sequence of formal parameters, then this form defines a procedure named variable, and formals corresponding to the sequence of parameters of the procedure, which are bound to the values of the arguments passed in a procedure call: (variable formals) The second form of define is also a way of defining a procedure, and making the name variable refer to that procedure. However, the second form of define also does a great deal more. This form causes the procedure to be treated as a method to be added to a polymorphic procedure, including taking care of the method combination bookkeeping that must be done when a method is added. This is the preferred way to define a procedure in Yolambda, because it allows the programmer to take fuller advantage of Yolambda's object oriented capabilities, including inheritance, and incremental development. We will discuss polymorphic procedures and method combination in more detail later in the manual. In the third form of define, the formal parameter after the dot means that a single formal parameter is bound in the call to the list of all the arguments to the call which are left after all the formals have been bound as they would in the second case of define. This form of define thus supports procedure calls with variable number of arguments.

Examples:

```
(define mypi 3)
(define (cadr pair|ls)
  (car (cdr ls)))
```

In this final example, the dot notation is used to allow the second argument to be optional. If only one argument is supplied in the call, then default will be the empty list. If the default argument is supplied in the call, then it will be the car of the list bound to default.

```
  (define (cadr pair|ls . default)
        (if (null? (cdr ls))
          (car default)
    (car (cdr ls))))
```

E.9 Binding Constructs

The binding constructs in Yolambda, let, let* and letrec, are conveniences derived from Lambda expressions. Like the lambda expression, they extend the environment in which they are defined (which we will call the outer environment) by adding bindings. In this way the binding constructs give Yolambda a block structure capability. In the case of each of the binding constructs, new variable bindings, i.e. fresh locations to which each named variable is bound, are created. These new bindings will supersede any bindings of those same variables that may exist in the outer environment. Upon exiting the binding construct, all the additional bindings are undone. That is, each named variable is rebound to the location (if any) to which it was bound in the outer environment, before the invocation of the binding construct. The differences between the binding constructs is the scope of the new bindings, and their relationship to the evaluation of expressions

that initialize the values for the variables. The following syntax for bindings applies to all three binding constructs:

```
((variable initialValueExpression) ...)
```

The syntax description for bindings, above, shows that bindings are a list of variable name and initialValueExpression pairs. The variable names are bound to fresh locations, and the initialValueExpressions are evaluated to produce values to be stored in those locations. Thus the variables are like the formal parameters of a lambda expression, and the nitialValueExpressions are like the arguments to the lambda expression.

(let bindings body) [Syntax]

The initial values are all evaluated, then each is stored in the new location assigned to the variable paired with the initial value. The scope of the extended environment is thus exactly body. The body of the let is then evaluated in the context of the new (extended) environment, and the value returned by the last evaluated expression in body is returned as the value of the let expression.

Examples:

```
(let ((a 1)(b 2))
  (+ a b))
```

The following example is a lambda expression that is equivalent to the example of let above:

```
((lambda (a b) (+ a b)) 1 2)
```

(let* bindings body) [Syntax]

Each initial value is evaluated, then stored in the new location assigned to the variable paired with the initial value. The bindings proceed in sequence, and the effect of each binding applies to the evaluation of the successive initial values. Thus the scope of each binding is the remaining bindings plus body. The context of each binding is the outer environment, plus the additions of each prior binding. The let* form is thus like nested let or lambda expressions. The body of the let* is then evaluated in the context of the new (extended) environment, and the value returned by the last evaluated expression in body is returned as the value of the let* expression.

Examples:

```
(let* ((a (+ 2 aValue))(b (* a 3)))
  (if (odd? a) a b))
```

The following examples are a let and a lambda expression that are each equivalent to the example of let* above:

```
(let ((a (+ 2 aValue)))
  (let ((b (* a 3)))
    (if (odd? a) a b)))

((lambda (a)
   (lambda (b) (if (odd? a) a b)) (* a 3))
 (+ 2 aValue))
```

(letrec bindings body) [Syntax]

The letrec construct is typically used to define local procedures, which may need to be recursive, or mutually recursive. In the letrec construct, all variables in the bindings are first bound to new, uninitialized locations. Then each initialValueExpression form is evaluated with

the entire letrec bindings form as its environment. This is what makes it possible to define mutually recursive local procedures. None of the variables in bindings can be assigned, nor can any values of variables be referenced, in the initialValueExpression forms. This restriction is not a problem with typical usage of letrec.
Examples:

```
(letrec ((fact
           (lambda (n)
 (if (= n 0)
     1
     (* n (fact (- n 1)))))))
  (fact 6))
```

E.10 Combining Form

There are places in Yolambda that require a single expression, where the programmer may want multiple expressions. The combining expression, begin, can be used for the purpose of making a single expression out of a collection of expressions.

(begin expression1 expression2 ...) [Syntax]

The combining construct begin is used to ensure that the component expressions are executed in sequential order. The begin expression is a single expression with sequentially executed component expressions. The result of the last expression in the sequence is returned by the begin expression.
Examples:

```
(display "Have a nice ")
(let ((currentTime
 (begin
       (set! *timeCounter* (+ *timeCounter* 1))
       (now))))
  (if (> (timeHour currentTime) 18)
    (begin
      (display "evening.")
      (set! timeCheck currentTime))
    (display "day.")))
```

E.11 Quasiquotation

Quasiquotation allows one to construct expressions from templates, where some of the expression is known in advance and other portions require evaluation. It can be used to construct vector or list (and hence program) structure.

(quasiquote template) or ‘template [Syntax]

The quasiquote form acts just like the quote form, except for its behavior with respect to expressions preceded by commas. So, for example, the following quasiquoted expressions: ‘foo and ‘(a b) are equivalent to their quoted counterparts: ’foo and ’(a b). A simple comma preceding an expression in a template causes that expression to be evaluated, and the result of the evaluation occupies within the evaluated template the place occupied by the original expression in the original template. So, for example, ‘(,(+ 1 2)) → (3). If the comma is followed by an "@", then the following expression must evaluate to a list, and the elements of the list are extracted from the list and spliced into the template in place and in order. ‘(,@(list ’+ 1 2)) → (+ 1 2)
Examples:

```
(let ((a 'foo))
  `(symbolValue ',a))

(symbolValue 'foo)

(let ((a 1)(b 2)(c 'list4))
  `(set! ,c (list ,@(list a b))))

(set! list4 (list 1 2))
```

References

Adams, R., and Bischof, L. 1994. Seeded region growing. *IEEE Trans. on PAMI* 16(6).

Agosta, J. M. 1990. The structure of bayes networks for visual recognition. In R.D. Shachter, T.S. Levitt, L. K., and Lemmer, J., eds., *Uncertainty in Artificial Intelligence, Vol 4*. North Holland, Amsterdam. 397–405.

Andrey, P., and Tarroux, P. Unsupervised texture segmentation using selectionist relaxation. *ECCV* A:482–491.

Arkin, R.; Riseman, E.; and Hansen, A. 1988. Aura: An architecture for vision-based robot navigation. Technical Report COINS TR88-07, Computer and Information Science, Univ. Massachusetts at Amherst.

Bahl, L.; Jelinek, F.; and Mercer, R. 1983. A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 5(2):179–190.

Barrett, W., and Mortensen, E. 1997. Interactive live-wire boundary extraction. 1(4):331–341.

Baum, L. 1972. An inequality and associated maximization technique in statistical estimation for probabilistic functions of a markov process. *Inequalities* 3:1–8.

Bawden, A. 1988. Reification without evaluation. In *Proceedings of the ACM Conference on LISP and Functional Programming*, 342–351.

Ben-Shaul, I.; Gazit, H.; Holder, O.; and Lavva, B. 2000. Dynamic self adaptation in distributed systems. In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software*. Springer-Verlag. 134–142.

Blake, A., and Zisserman, A. 1987. *Visual Reconstruction*. The MIT Press Series in Artificial Intelligence. Cambridge, USA: The MIT Press.

Brooks, R. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* 14–23.

Brooks, R. 1987. Intelligence without representation. *Artificial Intelligence* 47:139–159.

Brooks, R. 1991a. How to build complete creatures rather than isolated cognitive simulators. In VanLehn, K., ed., *Architectures for Intelligence*. Lawrence Erlbaum Associates. 225–239.

Brooks, R. 1991b. Integrated systems based on behaviors. *SIGART* Bulletin 2:46–50.

Buchanan, B., and Shortliffe, E. 1984. *Rule Based Expert Systems: the MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley.

Byrne, R., and Russon, A. 1998. Learning by imitation: A hierarchical approach. *J. Behavioral and Brain Sciences* 6 No. 3.

Canny, J. 1986. A computational approach to edge detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 8(6):679–698.

Charniak, E. 1993. *Statistical Language Learning*. MIT Press.

Charniak, E. 1996. Tree-bank grammars. In *Proceedings of the 13th National Conference on Artificial Intelligence*, 1031–1036.

Charniak, E. 1997. Statistical techniques for natural language parsing. 33–43.

Clearwater, S. 1995. Market-based control: A paradigm for distributed resource allocation. In Clearwater, S., ed., *World Scientific*.

Clowes, M. 1971. On seeing things. *Artificial Intelligence* 2:79–116.

Cootes, T.; Edwards, G.; and Taylor, C. 1998. Active appearance models. In Burkhardt, H., and Neumann, B., eds., *Proceedings, European Conference on Computer Vision 1998*, volume 2. Springer. pages 484–498.

Cross, G., and Jain, A. 1983. Markov random field texture models. *IEEE Trans. on PAMI* 5/1:25–39.

Dempster, A. 1968. A generalization of bayesian inference. *Journal of the Royal Statistical Society, Series B* 30:205–247.

des Rivieres, J., and Smith, B. 1984. The implementation of procedurally reflection languages. In *Proceedings 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas*, 331–347.

Dixon, K.; Pham, T.; and Khosla, P. 2000. Port-based adaptable agent architecture. In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software*. Springer-Verlag. 181–198.

Draper, B.; Collins, R.; Brolio, J.; Hansen, A.; and Riseman, E. 1988. The schema system. Technical Report COINS TR88-76, Computer and Information Science, Univ. Massachusetts at Amherst.

Ducksbury, P. 1993. Parallel texture region segmentation using a pearl bayes network. 187–196.

Erman, L.; Hayes-Roth, F.; Lessor, V.; and Raj-Reddy, D. 1980. The Hearsay-II speach-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys* 12(2):213–253.

Erman, L.; London, P.; and Fickas, S. 1981. The design and an example use of Hearsay-III. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 409–415.

Fahlman, S. 1973. A planning system for robot construction tasks. Technical Report Technical Report 283, MIT A.I. Lab.

Feigenbaum, E.; Buchanan, B.; and Lederberg, J. 1971. On generality and problem solving: A case study using the dendral program. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 6*. New York: American Elsevier. 165–190.

Forgy, C., and McDermott, J. 1977. Ops, a domain-independent production system language. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 933–939.

Francis, W., and Kucera, H. 1982. *Frequency Analysis of English Usage: Lexicon and Grammar*. Houghton Miffin, Boston.

Galef, B. G. 1998. Imitation in animals: history, definition, and interpretation of data from the psychological laboratory. In Zentall, T. R., and Galef, B. G., eds., *Social Learning: Psychological and Biological Perspectives*. Lawrence Erlbaum Associates.

Geman, S., and Geman, D. 1984. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Trans. Pattern Analysis and Machine Intelligence* 6:721–741.

Giroux, S. 1995. Open reflective agents. In M. Wooldridge, J. P. M., and Tambe, M., eds., *Intelligent Agents II Agent Theories, Architectures, and Languages*. Springer. 315–330.

Gosling, J.; Joy, W.; and Steele, G. 1996. *The Java Language Specification*. Addison-Wesley.

Grupen, R. 2000. Software mode changes for continuous motion tracking. In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software*. Springer-Verlag. 161–180.

Hammersley, J., and Handscomb, D. 1964. *Monte Carlo Methods*. Chapman and Hall, London.

Hayes, G., and Demiris, J. 1994. A robot controller using learning by imitation. In Borkowski, A., and Crowley, J. L., eds., *SIRS-94 Proc. of the 2nd Intern. Symposium on Intelligent Robotic Systems*, 198–204. SIRS-94. Grenoble France July 1994.

Hayes-Roth, B. 1985. A blackboard architecture for control. *Artificial Intelligence* 26(3):251–321.

Honda, Y., and Tokoro, M. 1992. Soft real-time programming through reflection. In Yonezawa, A., and C.Smith, B., eds., *Reflection and Meta-Level Architecture, Proceedings of the 1992 International Workshop on New Models for Software Architecture.* ACM. Tokyo.

Horritt, M. 1999. A statistical active contour model for sar image segmentation. *Image and Vision Computing* 17:213–224.

Hough, P. 1962. Method and means for recognising complex patterns. *US Pattent* 3069654.

Hu, K. 1962. On the amount of information. *Theor. Prob. Appl.* 7:439–447.

IEEE. 1991. Ieee standard for the scheme programming language. IEEE Standard 1178-1990, IEEE Pisscaataway.

ISO. 1992. Information technology - coding of moving pictures and associated audio for digital storage media up to about 1,5 mbit/s. Technical Report ISO/IEC JTC1 DIS 11172, ISO.

ISO. 1995. Mpeg4 proposal packet description - revision 2.0. Technical Report ISO-IEC JTC1/SC29/WG11 N0937, ISO.

Jackson, J. 1991. *A user's guide to Principal Components.* John Wiley and Sons, New York.

Jagannathan, V.; Dodhiawala, R.; and Baum, L., eds. 1989. *Blackboard Architectures and Applications.* Academic Press.

Jelinek, F.; Lafferty, J.; and Mercer, R. 1992. Basic methods of probabilistic context-free grammars. In Laface, P., and Mori, R. D., eds., *Speech recognition and understanding. Recent advances, trends, and applications, volume F75.* NATO ASI Series. Berlin: Springer Verlag.

Karssemeijer, N. 1992. Stochastic model for automated detection of calcifications in digital mammograms. *Image and Vision Computing* 10/6:369–375.

Kass, M.; Witkin, A.; and Terzopoulos, D. 1987. Snakes: Active contour models. In *Proceedings, 1st Int. Conference on Computer Vision.* pages 259–268.

Keene, S. E. 1989. *Object-Oriented Programming in Common Lisp: A programmer's Guide to CLOS.* Addison-Wesley.

Kiczales, G.; des Rivieres, J.; and Daniel, G. 1993. *The art of the Metaobject Protocol.* MIT Press.

Kohonen, T. 1988. Self-organized formation of topologically correct feature maps. *Biological Cybernetics* 43:59–69.

Kokar, M.; Passino, K.; Baclawski, K.; and Smith, J. 2000. Mapping an application to a control architecture. In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software.* Springer-Verlag. 75–89.

Krogh, C. 1995. The rights of agents. In M. Wooldridge, J. P. M., and Tambe, M., eds., *Intelligent Agents II Agent Theories, Architectures, and Languages.* Springer. 1–16.

Laddaga, R., and Robertson, P. 1996. *Yolambda Reference Manual.* Dynamic Object Language Labs, Inc.

Laddaga, R. 1996. *DOIT Reference Manual.* Dynamic Object Language Labs, Inc.

Lamping, J.; Kiczales, G.; Rodriguez, L.; and Ruf, E. 1992. An architecture for an open compiler. In Yonezawa, A., and C.Smith, B., eds., *Reflection and Meta-Level Architecture, Proceedings of the 1992 International Workshop on New Models for Software Architecture.* ACM. Tokyo.

Lau, T., and Ho, Y. 1997. Universal alignment probabilities and subset selection for ordinal optimization. *Journal of Optimization Theory and Applications* 93:455–489.

Leclerc, Y. G. 1989. Constructing simple stable descriptions for image partitioning. *Int. J. of Computer Vision* 3:73–102.

Ledeczi, A.; Bakay, A.; and Maroti, M. 2000. Model-integrated embedded systems. In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software.* Springer-Verlag. 99–115.

Lee, W.; Golston, J.; Gove, R.; and Kim, Y. 1994. Real-time mpeg video codec on a single-chip multiprocessor. *Digital Video Compression on Personal Computers: Algorithms and Technologies* 2187:32–42.

Maes, P., and Nardi, D. 1988. *Meta-Level Architectures and Reflection.* North-Holland.

Maes, P. 1990a. Designing autonomous agents. In Maes, P., ed., *Designing Autonomous Agents.* MIT/Elsevier. 1–2.

Maes, P. 1990b. Situated agents can have goals. In Maes, P., ed., *Designing Autonomous Agents.* MIT/Elsevier. 49–70.

Makowski, P.; Ssrensen, T.; Therkildsen, S.; Stsdkilde-Jorgensen, H.; and Pedersen, E. M. 2000. Upper and lower probabilities induced by a multivalued mapping. *Proc. Intl. Sot. Mag. Reson. Med.* 8:1547–1547.

Malenfant, J.; Cointe, P.; and Dony, C. 1991. Reflection in prototype-based object-oriented programming languages. In *Proceedings of the OOPSLA Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming.*

Mann, W., and Binford, T. 1994. Probabilities for bayesian networks. In *Proceedings Image Understanding Workshop.* Morgan Kaufman, San Francisco.

Marr, D. 1976. Early processing of visual information. *Phil. Trans. R. Soc. Lond. B* 275:483–524.

Marr, D. 1982. *Vision.* W. H. Freeman and Company.

Mataric, M., and Pomplun, M. 1998. Fixation behavior in observation and imitation of human movement. *Cognitive Brain Research* 7 No. 2:191–202.

Mataric, M. 1991. Behavioral synergy based on behaviors. *SIGART* Bulletin 2:85–88.

Mataric, M. 1992. Bahavior-based systems: Main properties and implications. In *Proceedings, IEEE International Conf. on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems.* IEEE. pages 46-54.

Mataric, M. 2000a. Getting humanoids to move and imitate. *IEEE Intelligent Systems* July/August 2000:18–24.

Mataric, M. 2000b. Sensory-motor primitives as a basis for imitation: Linking perception to action and biology to robotics. In nehaniv, C., and K.Dautenhahn., eds., *Imitation in Animals and Artifacts.* MIT Press, Cambridge.

McClelland, J., and Rumelhart, D. 1986a. *Parallel distributed processing. 1. Explorations in the microstructure of cognition. 2. Psychological and biological models.* MA: MIT Press.

McClelland, J., and Rumelhart, D. 1986b. The programmable blackboard model of reading. In *Parallel distributed processing.* MA: MIT Press. 122–169.

McDermott, D., and Sussman, G. 1973. The conniver reference manual. Technical Report AI Memo 259, MIT A.I. Lab.

Meng, A. 2000. On evaluating self-adaptive software. In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software.* Springer-Verlag. 65–74.

Minsky, M. 1975. A framework for representing knowledge. In Winston, P. H., ed., *The Psychology of Computer Vision.* McGraw-Hill, New York.

Moody, J., and Darken, C. 1989. Fast learning in networks of locally-tuned processing units. *Neural Computation* 1(2):281–294.

Musliner, D.; Goldman, R.; Pelican, M.; and Krebsbach, K. 1999. Self-adaptive software for hard real-time environments. *IEEE Intelligent Systems* 14(4):23–29.

Musliner, D. 2000. Imposing real-time constraints on self-adaptive controller synthesis. In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software*. Springer-Verlag. 143–160.

Nair, D., and Aggarwal, J. 1996. A focused target segmentation paradigm. In *Proc. of the Fourth European Conference on Computer Vision, Cambridge, UK*, 579–588.

Noda, H.; Shirazi, M.; and Kawaguchi, E. An mrf model-based method for unsupervised textured image segmentation. *ICPR* B:765–769.

Osterweil, L., and Clarke, L. 2000. Continuous self-evaluation for the self-improvement of software. In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software*. Springer-Verlag. 27–39.

Pavlovic, D. 2000. Towards semantics of self adaptive software. In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software*. Springer-Verlag. 50–64.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, California: Morgan Kaufmann Publishers.

Rao, A., and Georgeff, M. 1995. Bdi-agents: from theory to practice. In *Proceedings of the First Intl. Connference on Multiagent Systems, San Francisco*.

Reece, S. 2000. Self-adaptive multi-sensor systems. In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software*. Springer-Verlag. 224–241.

Robertson, P. 1985. Opsada3. In Johnson, J. P., ed., *Washington Ada Symposium*, 139–146. Association for Computing Machinery.

Robertson, P. 1992. On reflection and refraction. In Yonezawa, A., and C.Smith, B., eds., *Reflection and Meta-Level Architecture, Proceedings of the 1992 International Workshop on New Models for Software Architecture*. ACM. Tokyo.

Rosenschein, J., and Zlotkin, G. 1998. Designing conventions for automated negotiation. In Huhns, M. N., and Singh, M. P., eds., *Readings in Agents*. Morgan-Kaufman. 353–370.

Rowley, S.; Shrobe, H.; and Cassels, R. 1987. Joshua: Uniform access to heterogeneous knowledge structures or why joshing is better than conniving or planning. In *Proceedings of the sixth national conference on Artificial Intelligence*, 48–58.

Schabes, Y. 1992. Stochastic lexicalized tree-adjoining grammars. In *Proceedings of the 14th International Conference on Computational Linguistics*.

Shafer, G. 1976. *A Mathematiccal Theory of Evidence*. Prinston University Press.

Shannon, C. 1949. Communicating in the presence of noise. *Proc. IRE* 37:10–21.

Shrobe, H., and Doyle, J. 2000. Active trust management for autonomous adaptive survivable systems (atm's for aass's). In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software*. Springer-Verlag. 40–49.

Simon, G.; Kovacshazy, T.; and Peceli, G. 2000. Transient management in reconfigurable systems. In P. Robertson, R. L., and Shrobe, H., eds., *Self-Adaptive Software*. Springer-Verlag. 90–98.

Smith, B. 1982. Reflection and semantics in a procedural language. Technical Report 272, MIT Laboratory for Computer Science.

Smith, B. 1984. Reflection and semantics in lisp. In *Proceedings 11th Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah*, 23–35.

Sobel, I. 1974. On calibrating computer controlled cameras for for perceiving 3-D scenes. *J. Artificial Intelligence* 5:185–198.

Steele, G. 1984. *Common Lisp: The Language*. Digital Press.

Viterbi, A. 1967. Error bounds for convolution codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory* 13:260–269.

Wallace, C. 1990. Classification by minimum-message-length inference. In Goos, G., and Hartmanis, J., eds., *Advances in Computing and Information–ICCI'90*. Springer-Verlag. 72–81.

Waltz, D. 1975. Understanding line drawings of scenes with shadows. In Winston, P. H., ed., *Psychology of Computer Vision*. McGraw-Hill New York. 19–91.

Weber, S.; Mataric, M.; and Jenkins, O. 2000. Experiments in imitation using perceptuo-motor primitives. In *Autonomous Agents*, 136–137. ACM Press. New York.

Winston, P. 1975. Learning structural descriptions from examples. In Winston, P. H., ed., *The Psychology of Computer Vision*. McGraw-Hill book company, NY.

Xie, Z., and Brady, J. 1996. Texture segmentation using local energy in wavelet scale space. *Image Understanding*.

Zadeh, L. 1999. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems* 100:9–34.

Zhu, S., and Yuille, A. 1996. Region competition: unifying snakes, region growing, and bayes/mdl for multiband image segmentation. *IEEE Trans. on PAMI* 18(9):884–900.