

**An Architecture and Interaction Techniques for Handling  
Ambiguity in Recognition-based Input**

A Thesis  
Presented to  
The Academic Faculty

by

**Jennifer Mankoff**

Doctor of Philosophy in Computer Science

Georgia Institute of Technology  
May 2001

Copyright © 2001 by Jennifer Mankoff

# An Architecture and Interaction Techniques for Handling Ambiguity in Recognition-based Input

Approved:

---

Dr. Gregory D. Abowd, Chairman

---

Dr. Scott E. Hudson

---

Dr. Elizabeth D. Mynatt

---

Dr. Mark Guzdial

---

Dr. Thomas Moran

Date Approved \_\_\_\_\_

# PREFACE

It is difficult to build applications that effectively use recognizers, in part because of lack of toolkit-level support for dealing with recognition errors. This dissertation presents an architecture that addresses that problem.

Recognition technologies such as speech, gesture, and handwriting recognition, have made great strides in recent years. By providing support for more natural forms of communication, recognition can make computers more accessible. Such “natural” interfaces are particularly useful in settings where a keyboard and mouse are not available, as in very large or very small displays, and in mobile and ubiquitous computing. However, recognizers are error-prone: they may not interpret input as the user intended. This can confuse the user, cause performance problems, and result in brittle interaction dialogues.

The major contributions of this thesis are:

- A model of recognition that uses ambiguity to keep track of errors and mediation to correct them. This model can be applied to any recognition-based interface found in the literature, all of which include some type of support for mediation of recognition errors.
- A corresponding toolkit architecture that uses this model to represent recognition results. The toolkit architecture benefits programmers who build recognition-based applications, by providing a separation of recognition and mediation from application development.

At a high level, mediation intervenes between the recognizer and the application in order to resolve ambiguity. This separation of concerns allows us to create re-usable solutions to

mediation similar to the menus, buttons, and interactors provided in any GUI toolkit. The separation of recognition also leads to the ability to adapt mediation to situations which do not seem to be recognition-based, but where some problem in interaction causes the system to do something other than what the user intended (our ultimate definition of error). Finally, the separation of mediation allows us to develop complex mediation interactions independent of both the source of ambiguity and the application.

# DEDICATION

This thesis is dedicated to Canine Companions for Independence.

## ACKNOWLEDGMENTS

Although my name stands alone on this document, it would never have been completed without the support of countless other people. At the times when it has seemed most impossible to continue, those people have helped to show me how to take the next step.

Gregory, not only have you helped to teach me the many skills required to take the next step beyond my PhD, you have given endlessly of your time, support, and knowledge. And when I wasn't sure if I could go on physically, you gave me the support and space I needed to find the answer. You have shown me what an advisor should be, and I hope to live up to your example when I take on the same role.

Scott, you also have come to be an advisor to me. I can't tell you how much I value the many brainstorming sessions in which we have tackled and unknotted exciting new problems (sometimes tying our own brains in knots in the process). And thank you for your willingness to even go as far as implementing code for me when my hands gave out.

My biggest challenge in completing this degree was without a doubt the injury of my hands. I would not have been able to work around the limitations of this injury without the help and support of countless people. Anind Dey, Cynthia Bryant, Gregory Abowd, Joe Bayes, my family, and many others have done everything from tying my shoes or lifting bags of dog food to typing in portions of my thesis over the years since my repetitive strain injury began. Additionally, a host of wonderful therapists, musicians, and dog trainers have helped me to find solutions for dealing with my injury that gave me both the literal and spiritual strength to continue onwards. Among all of these supporters, I dedicate this thesis to Canine Companions for Independence. It was through your organization that I finally found the passion to help people with disabilities that led me to finish the degree rather than give in to my injury.

To everyone on my committee: you have all helped to contribute to my growth as a researcher and to the quality of this thesis. Thank you for the time and energy you have put towards that task. Thanks also go to the members of my research community that have contributed pieces to this work. The Java-based unistroke gesture recognizer used for demonstration in this paper, GDT, was provided by Chris Long from UC Berkeley as a port of Dean Rubine's original work [73]. It was used in Burlap (Chapter 5) and in our word predictor (Chapter 6.1). Takeo Igarashi provided the drawing beautifier recognizer, Pegasus, that was also used for demonstrational purposes [37], as well as in Figure 2-2. Bernhard Suhm provided Figure 2-1 for us from his thesis work [80].

Anind, my friend and my love, you have shown me how to live with and accept my injury without letting it overtake me. You have helped me to know and to give priority to the things that are important in my life. You are always willing to hear my ideas, no matter how ill-formed they are, and to help me evolve them. I am so happy to be facing life with you at my side.

Josh, Ken, and all of my other friends, both local and scattered around the country. You have all taught me many lessons as I traveled this path, and spent many hours listening to me when I needed to talk. None of this could have happened in the vacuum that would have existed without your presence.

Finally, to my parents, you have always supported me no matter what I wanted to do. From the first time I showed interest in math and computers through each of the hurdles I have faced along the way, you have always given me whatever I needed to follow my dreams.

# CONTENTS

<b>PREFACE</b>	<b>iii</b>
<b>DEDICATION</b>	<b>v</b>
<b>ACKNOWLEDGMENTS</b>	<b>vi</b>
<b>LIST OF TABLES</b>	<b>xiv</b>
<b>LIST OF FIGURES</b>	<b>xv</b>
<b>LIST OF SOURCE CODE</b>	<b>xvii</b>
<b>1 INTRODUCTION</b>	<b>1</b>
1.1 Definitions . . . . .	3
1.1.1 Recognition . . . . .	3
1.1.2 Errors . . . . .	4
1.1.3 Ambiguity . . . . .	6
1.1.4 Mediation of ambiguity . . . . .	8
1.1.5 Relationship between recognizers, errors, ambiguity, and mediation . . . . .	8
1.2 Thesis Statement, Contributions and Overview of Dissertation . . . . .	10
<b>2 MEDIATION</b>	<b>14</b>
2.1 Repetition . . . . .	14
2.1.1 Examples . . . . .	15
2.1.1.1 Example of multimodal repetition . . . . .	15



2.1.1.2	Example of unimodal repetition . . . . .	16
2.1.1.3	Comparison of different repetition mediators . . . . .	16
2.1.2	Issues and problems . . . . .	19
2.2	Choice . . . . .	20
2.2.1	Examples . . . . .	20
2.2.1.1	Description of ViaVoice <sup>TM</sup> mediation . . . . .	21
2.2.1.2	Description of Pegasus mediation . . . . .	22
2.2.1.3	Comparison of different choice mediators . . . . .	22
2.2.2	Issues and problems . . . . .	27
2.3	Automatic Mediators . . . . .	28
2.3.1	Examples . . . . .	29
2.3.2	Issues and problems . . . . .	30
2.4	Meta-mediation: Deciding When and How to Mediate . . . . .	30
2.4.1	Examples . . . . .	30
2.4.2	Issues and problems . . . . .	31
2.5	The Need for Toolkit Support for Mediation . . . . .	31
<b>3</b>	<b>AN ARCHITECTURE FOR MEDIATION</b>	<b>33</b>
3.1	Existing Support for Recognized Input . . . . .	34
3.1.1	GUI toolkit architectures . . . . .	36
3.1.2	Other toolkit architectures . . . . .	37
3.1.3	In summary . . . . .	38
3.2	The Event Hierarchy . . . . .	41
3.2.1	The event object . . . . .	42
3.2.2	The event hierarchy . . . . .	45
3.2.3	Representing different types of errors and ambiguity . . . . .	47
3.3	Event Dispatch . . . . .	49

3.4	Mediation Subsystem . . . . .	51
3.4.1	The mediation dispatch algorithm . . . . .	52
3.4.2	Mediation . . . . .	54
3.4.3	Meta-mediation . . . . .	57
3.4.4	Example . . . . .	58
3.5	Additional Issues . . . . .	61
3.5.1	Modifying dispatch to provide backward compatibility . . . . .	61
3.5.2	Extending recognizers . . . . .	65
3.5.2.1	Delayed recognition . . . . .	66
3.5.2.2	Guided rerecognition . . . . .	68
3.5.2.3	Domain and range filters . . . . .	69
3.5.3	Avoiding endless loops . . . . .	70
3.5.4	Future issues . . . . .	70
3.6	Instantiating the Architecture . . . . .	71
<b>4</b>	<b>OOPS: MEDIATION IN THE GUI WORLD</b>	<b>72</b>
4.1	The Architecture of a Standard GUI Toolkit . . . . .	74
4.2	The Architecture of OOPS . . . . .	76
4.2.1	OOPS input . . . . .	76
4.3	Backwards Compatibility . . . . .	79
4.4	Meta-mediators and Mediators in the Library of OOPS . . . . .	81
4.4.1	Meta-mediators . . . . .	83
4.4.2	Mediators . . . . .	88
4.4.2.1	Choice mediators . . . . .	90
4.4.2.2	Repetition mediators . . . . .	94
4.4.2.3	Automatic mediators . . . . .	95
4.5	Case Study: A Word Predictor . . . . .	97

4.5.1	Many words . . . . .	102
4.5.2	A new mediator . . . . .	103
4.5.3	Combining recognizers . . . . .	104
4.6	Different Levels of Involvement . . . . .	105
<b>5</b>	<b>AN EXTENDED EXAMPLE: DEMONSTRATING SUFFICIENCY</b>	<b>108</b>
5.1	Overview of Burlap . . . . .	110
5.2	Types of Ambiguity . . . . .	114
5.3	Mediation . . . . .	115
5.3.1	Repetition mediation . . . . .	115
5.3.2	Choice mediation . . . . .	118
5.3.3	Automatic mediation . . . . .	120
5.3.4	Meta-mediation . . . . .	122
5.4	How Burlap was Created . . . . .	125
<b>6</b>	<b>SOLVING NEW PROBLEMS: ADDED BENEFITS</b>	<b>127</b>
6.1	Adding Alternatives . . . . .	127
6.1.1	Problem . . . . .	127
6.1.2	Solution . . . . .	130
6.1.3	Toolkit support for adding alternatives . . . . .	131
6.1.4	Reusability . . . . .	132
6.1.5	Analysis . . . . .	132
6.2	Occlusion in Choice Mediators . . . . .	133
6.2.1	Problem . . . . .	133
6.2.2	Solution . . . . .	133
6.2.3	Toolkit support for dealing with occlusion . . . . .	135
6.2.4	Reusability . . . . .	135
6.2.5	Analysis . . . . .	136

6.3	Target Ambiguity . . . . .	136
6.3.1	Problem . . . . .	136
6.3.2	Solution . . . . .	137
6.3.3	Toolkit support for target ambiguity . . . . .	139
6.3.4	Reusability . . . . .	139
6.3.5	Analysis . . . . .	140
6.4	Rejection Errors . . . . .	140
6.4.1	Problem . . . . .	140
6.4.2	Solution . . . . .	142
6.4.3	Toolkit support for guided rerecognition . . . . .	142
6.4.4	Reusability . . . . .	143
6.4.5	Analysis . . . . .	144
6.5	Summary . . . . .	144
<b>7</b>	<b>GENERALIZING THE ARCHITECTURE: A NEW DOMAIN</b>	<b>147</b>
7.1	Motivation for Mediation in Context-Aware Computing . . . . .	148
7.2	Combining the Toolkits . . . . .	149
7.2.1	The Context Toolkit . . . . .	149
7.2.2	Adding ambiguity to the Context Toolkit . . . . .	152
7.3	Mediating Simple Identity and Intention in the Aware Home . . . . .	154
7.3.1	The modified In/Out Board . . . . .	154
7.3.2	Issues and problems . . . . .	158
7.4	Other Settings For The Architecture . . . . .	160
7.4.1	The ambiguous event hierarchy . . . . .	160
7.4.2	The mediation subsystem . . . . .	160
7.4.3	The input handling subsystem . . . . .	161
7.4.4	Additional details . . . . .	161

<b>8</b>	<b>CONCLUSIONS</b>	<b>163</b>
8.1	Contributions . . . . .	163
8.2	Future Work . . . . .	165
8.2.1	Evaluation . . . . .	165
8.2.2	More examples . . . . .	166
8.2.3	New settings . . . . .	167
8.3	Conclusions . . . . .	168
<b>A</b>	<b>The complete set of algorithms needed to handle ambiguous event dis-</b>	
	<b>patch</b>	<b>169</b>
A.1	Maintenance Algorithms for the Event Graph . . . . .	169
A.2	Algorithms for Resolving Ambiguity in Mediators and Meta-Mediators . . .	171
A.3	The Algorithms Associated with Event Dispatch . . . . .	174
A.4	Some Meta-Mediators . . . . .	176
<b>VITA</b>		<b>178</b>
<b>BIBLIOGRAPHY</b>		<b>180</b>

## LIST OF TABLES

2-1	A comparison of repetition mediators . . . . .	17
2-2	A comparison of choice mediators . . . . .	25
3-1	A comparison of different systems supporting recognition . . . . .	35
3-2	The most important information associated with each event . . . . .	42
3-3	Constants and methods in the <i>mediation</i> class . . . . .	56
3-4	Constants and methods in the <i>meta-mediation</i> class . . . . .	57
4-1	A comparison of the default meta-mediation policies provided with OOPS .	86
4-2	A comparison of the default mediators provided with OOPS . . . . .	90
4-3	Methods implemented by a <i>layout</i> object in OOPS . . . . .	92
4-4	Methods implemented by <i>feedback</i> objects in OOPS . . . . .	92
5-1	Types of ambiguity in Burlap . . . . .	114
5-2	The full complement of mediators that are used by Burlap . . . . .	116
5-3	A comparison of two repetition mediators in Burlap . . . . .	118
5-4	A comparison of three choice mediators in Burlap . . . . .	119

## LIST OF FIGURES

1-1	An $n$ -best list from the ViaVoice <sup>TM</sup> speech system . . . . .	2
1-2	An example of target ambiguity . . . . .	7
2-1	Mixed granularity repair using a pen in a speech dictation task . . . . .	15
2-2	The mediator used in the the Pegasus system . . . . .	23
3-1	An example of recognition of user input . . . . .	45
3-2	An example trace of the methods called by <code>dispatch_event()</code> . . . . .	60
3-3	An example trace of the methods called by <code>dispatch_event()</code> . . . . .	68
4-1	A comparison of how recognized input is handled in a standard GUI toolkit and OOPS . . . . .	75
4-2	Four example choice mediators created from the OOPS choice base class . .	91
4-3	A simple text editor . . . . .	98
4-4	A simple mediator handling the word-prediction . . . . .	100
4-5	The modified button returned by the feedback object for a choice mediator that supports filtering . . . . .	104
4-6	Word prediction in combination with pen input . . . . .	105
5-1	Sketching a sample interface to a drawing program in Burlap . . . . .	109
5-2	A comparison of commands and gestures recognized by Burlap and SILK .	112
5-3	A hierarchy generated by a combination of both recognizers used in Burlap	113
5-4	An example of repetition in Burlap . . . . .	117
5-5	The end user trying to sketch some radio buttons . . . . .	120
5-6	The organization of the meta-mediators and mediators installed in Burlap .	122
5-7	An example of a stroke that is ambiguous . . . . .	124
6-1	A menu of predicted URLs in Internet Explorer <sup>TM</sup> . . . . .	128

6-2	A choice mediator that supports specification . . . . .	129
6-3	The original and modified event hierarchy produced by the modified choice mediator . . . . .	131
6-4	An example of fluid negotiation to position a mediator in the Burlap application	134
6-5	An example of mediation of ambiguous clicking with a magnifier . . . . .	138
6-6	An example of the magnification mediator being used in Burlap . . . . .	140
6-7	An example of repetition through rerecognition . . . . .	141
7-1	Sample Context Toolkit and CT-OOPS components . . . . .	150
7-2	The flow of control for mediation in modified context toolkit (CT-OOPS) .	153
7-3	Photographs of the In/Out Board physical setup . . . . .	155
7-4	The In/Out Board with transparent graphical feedback . . . . .	156



## LIST OF SOURCE CODE

3-1	The algorithm used to determine whether an event hierarchy is ambiguous ( <code>is_ambiguous()</code> ) . . . . .	46
3-2	The algorithm used to determine whether an event has siblings that are ambiguous ( <code>conflicting_siblings()</code> ) . . . . .	46
3-3	The algorithm to accept events ( <code>accept()</code> ) . . . . .	47
3-4	The algorithm to reject events ( <code>reject()</code> ) . . . . .	47
3-5	The basic event dispatch algorithm ( <code>dispatch_event()</code> ) . . . . .	50
3-6	A helper algorithm for event dispatch ( <code>generate_interps()</code> ) . . . . .	50
3-7	The algorithm called by the event dispatch system to send an ambiguous event hierarchy to meta-mediators for resolution ( <code>resolve_ambiguity()</code> ) .	53
3-8	The algorithm used when a mediator that has paused mediation wishes to pass control back to the mediation subsystem ( <code>continue_mediation()</code> ) . .	53
3-9	The algorithm used by a mediator to mediate an event hierarchy ( <code>mediate()</code> )	55
3-10	The algorithm used by meta-mediators to send event hierarchies to mediators for resolution ( <code>meta_mediate()</code> ) . . . . .	59
3-11	The algorithm called by a mediator to continue mediation that has been deferred ( <code>continue_mediation()</code> ) . . . . .	59
3-12	The modified event dispatch algorithm for components unaware of ambiguity ( <code>dispatch_event()</code> ) . . . . .	63
3-13	The algorithm called by event dispatch when a modification is made to a deferred event hierarchy ( <code>handle_modification()</code> ) . . . . .	66
4-1	An example implementation of the (one method) filter interface ( <code>filter()</code> )	84
4-2	The <code>meta_mediate()</code> method in the filter meta-mediator . . . . .	84

4-3	The <code>meta_mediate()</code> method in the queue meta-mediator . . . . .	87
4-4	The <code>meta_mediate()</code> method in the positional meta-mediator . . . . .	87
4-5	An automatic mediator that pauses input until the user has typed three characters . . . . .	89
4-6	An audio mediator . . . . .	92
4-7	The additional code needed to add word prediction to a simple text entry application . . . . .	101
A-1	The algorithm used to determine whether an event hierarchy is ambiguous ( <code>is_ambiguous()</code> ) . . . . .	169
A-2	The algorithm to accept events ( <code>accept()</code> ) . . . . .	170
A-3	The algorithm to reject events ( <code>reject()</code> ) . . . . .	170
A-4	The algorithm used to retrieve the leaf nodes of an event hierarchy once ambiguity has been resolved ( <code>leaf_nodes()</code> ) . . . . .	170
A-5	The algorithm used to determine whether an event has siblings that are ambiguous ( <code>conflicting_siblings()</code> ) . . . . .	171
A-6	The algorithm that returns the root sources of an event ( <code>root_sources()</code> )	171
A-7	The algorithm used by a mediator to mediate an event hierarchy ( <code>mediate()</code> )	172
A-8	The algorithm used by meta-mediators to send event graphs to mediators for resolution ( <code>meta_mediate()</code> ) . . . . .	172
A-9	The algorithm used to continue deferred mediation ( <code>continue_mediation()</code> )	173
A-10	The algorithm used by meta mediators to handle updates to an event graph ( <code>handle_modification()</code> ) . . . . .	173
A-11	The modified event dispatch algorithm for components unaware of ambiguity ( <code>dispatch_event()</code> ) . . . . .	174
A-12	A helper algorithm for event dispatch ( <code>generate_interps()</code> ) . . . . .	174
A-13	The final phase of the event dispatch algorithm ( <code>complete_dispatch()</code> ) . .	175

A-14	The algorithm used by the event dispatch system to send event graphs to meta mediators for resolution ( <code>resolve_ambiguity()</code> ) . . . . .	175
A-15	The algorithm used when a mediator that has paused mediation wishes to pass control back to the mediation subsystem ( <code>continue_mediation()</code> ) . .	176
A-16	An alternative implementation of positional ( <code>meta_mediate()</code> ) . . . . .	177

# Chapter 1

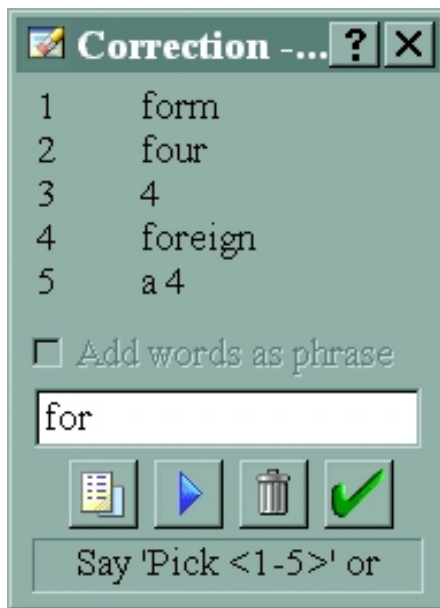
## INTRODUCTION

It is difficult to build applications that effectively use recognizers, in part because of lack of toolkit-level support for dealing with recognition errors. This dissertation presents an architecture that addresses that problem.

Recognition technologies such as speech, gesture, and handwriting recognition, have made great strides in recent years. By providing support for more natural forms of communication, recognition can make computers more accessible. Such “natural” interfaces are particularly useful in settings where a keyboard and mouse are not available, as with very large or very small displays, and with mobile and ubiquitous computing.

However, recognizers are error-prone: they may not interpret user input as the user intended. This can confuse the user, cause performance problems, and result in brittle interaction dialogues. For example, Suhm found that even though humans speak at 120 words per minute (wpm), the speed of spoken input to computers is 40 wpm on average because of recognition errors [81]. Similarly, Halverson *et al.* found that input speeds may decrease to 25 wpm due in large part to time spent correcting recognition errors [30].

Interfaces that use recognizers must contend with errors, and in fact research has shown that variations on how these interfaces handle recognition errors can reduce some of the negative effects of those errors [81, 2]. As an example, consider the menu of alternatives (called an *n*-best list) shown in Figure 1-1. It is part of the ViaVoice<sup>TM</sup> speech recognition system. It is an example of a choice-based interface, because it gives the user a choice of different possible interpretations of her input. *Choice* is one of the two common interaction



**Figure 1-1:** An  $n$ -best list from the ViaVoice<sup>TM</sup> speech system [7]. The user has just spoken a sentence, and then selected a misrecognized word for correction.

---

techniques for correcting recognition errors. In the other major strategy, *repetition*, the user repeats, or modifies, her input, either in the same or in a different modality. We call repetition and choice *mediation techniques* because they are mediating between the user and the computer to specify the correct interpretation of the user's input. Choice and repetition strategies have a fairly wide range of possible instantiations, making mediation techniques ripe for reusable toolkit-level support. These strategies were uncovered in a survey of interfaces making use of recognition, described in Chapter 2.

The goal of this thesis work is to allow an application designer to build an application that uses third-party recognizers and mediation techniques without requiring the designer to modify significantly how she designs the application and user interface. Beyond that, we hope to provide the infrastructure necessary to allow the designer to freely explore new interaction techniques that may require modifications to the application as well as to how mediation is done.

The remainder of this chapter will focus on defining the basic terms, and associated research areas, that motivated this thesis. Those definitions lead directly to the thesis statement, introduced in last section of this chapter. The chapter ends with a description of how the remainder of this dissertation will prove the underlying and overt assumptions in the thesis statement.

## 1.1 Definitions

We begin by defining *recognition*. This work focuses on using recognizers as they are, with errors, rather than trying to improve upon them. We then define the term *error* and review existing categorizations of errors and their sources. This discussion is confined to errors arising from the recognition process. Although a system cannot know the correct answer *a priori*, it may model the many possible interpretations of user input internally as a set of *ambiguous* possibilities. Section 1.1.3 of this chapter defines ambiguity and discusses different types of ambiguity that we have encountered in our work. *Mediation* techniques then serve to resolve this ambiguity by helping to determine which of those potential interpretations is the correct one, the one the user intended. Integrated architectural support for ambiguity at the input-handling level is required for this to be done properly. Finally we present a summary of problem areas, including types of errors and types of ambiguity that existing systems do not handle well.

### 1.1.1 Recognition

*Recognition* involves sensing user input or other information about the user, and interpreting it. Some traditional forms of recognition include speech, handwriting, and gesture recognition. Other types of recognition include face recognition, activity recognition (an important component of context-aware computing [76]), and word-prediction.

In general, the goal of perfect recognition is difficult because correct recognition is best defined as what the user intends, information the system does not have. In addition, when recognition is done in noisy, varied environments, accuracy typically decreases. In practice, researchers try to reduce errors instead of eliminating them. Error reduction is a difficult problem, and big improvements (5–10%) are needed before users even notice a difference, particularly for very inaccurate recognizers (~50% accuracy) [14].

### 1.1.2 Errors

In our human-centered view of the world, an *error* occurs when the system interprets (‘recognizes’) the user’s input in a way that the user did not intend. For example, if the user of a speech dictation system says “Hello Kate” and what appears on her screen is “Locate,” an error has occurred.

Errors can occur at many levels in the process that goes from sensing of user input to system action on recognized input. As Baber and Hone suggest, we can view this process as a series of levels in which input is perceived, separated into lexical tokens, and parsed into a syntactic structure [6]. At the perceptual level, errors may be caused by noise in the environment or the device doing the sensing. For example, speech recognition may fail in noisy environments. At the lexical level, errors may occur due to bad form in the input, such as a stroke that is drawn in the wrong direction. At the syntactic level, errors may occur due to bad syntax (*e.g.* a mis-spelling). Arguably, a semantic level may be added to this, in which input may make no sense from the application’s perspective.

Another way to consider the causes of errors is to examine, from a system perspective, the stages of recognition. For example, Brennan and Hulsten give the following stages of recognition for a speech system: not attending, attending, hearing, parsing, interpreting, intending, acting, reporting [11]. Errors can occur at any of these stages, or in the transitions between stages. One of the reasons it is difficult to eliminate errors is because there are so many sources of errors that need to be addressed.

What exactly can go wrong? In recognition-based input, there are three major types of errors that occur: *rejection*, *insertion*, and *substitution* errors [6]. Huerst, Yang and Waibel use slightly different terms (deletions, completions and insertions, and overwriting) to refer to the same types of errors [36].

**Rejection:** A rejection error occurs when the user's input is not recognized at all. Other terms for this are deletion, false negative, or error of omission. For example, if a user speaks too quietly, or there is too much ambient noise, a speech recognizer may not even realize that something was said. Similarly, if a stroke is too short (*e.g.* dotting an 'i'), a handwriting system may not realize that the user was writing and not clicking. Other possible causes of rejection errors include incorrect input (from an application perspective) or illegible input [36]. Rejection errors are difficult for a system to discover because by their very nature they are hidden: *The system does not know anything happened.* This usually leaves the user only one option: try again. This is one of the worst repair strategies both because the user has little feedback about the cause of the error, and because recognition is often no better during repetition [25, 24].

**Insertion:** An insertion error (also called a false positive) occurs when the user did not intend to create any input, but the recognizer produced some interpretation. For example, a speech recognizer may think the user is speaking when there was only ambient noise. Insertion errors are difficult to discover for similar reasons to rejection errors. However, since they do cause interpretation to happen, they are easier for the user to correct, and for a system to discover.

**Substitution:** Also referred to as overwriting, this is the only type of error handled explicitly by existing mediation interfaces, and maps directly onto our definition of error. A substitution error occurs when the user does something intending it to be interpreted



one way, and the system interprets it differently. Both repetition and choice mediation techniques can be used to correct substitution errors (see Chapter 2).

### 1.1.3 Ambiguity

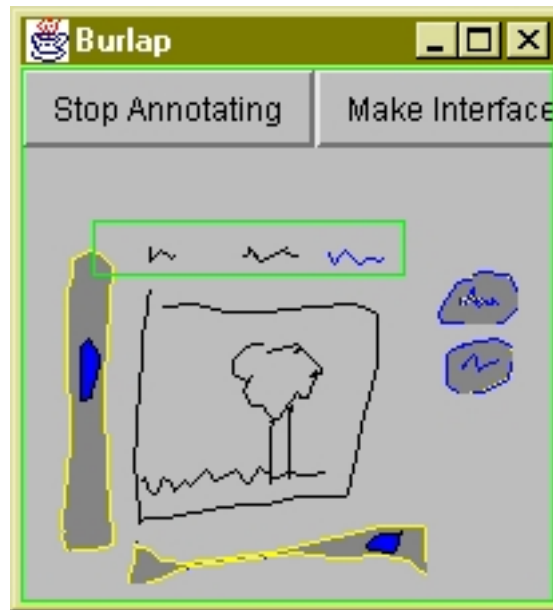
An error occurs when the system acts on an incorrect interpretation of the user’s input. In order for the error to be corrected, the system needs to find out what the other (correct) interpretation is. For example, in the case of a substitution error <sup>1</sup>, if the user intended to write “hello” and the recognizer returned “hallo”, “hallo” and “hello” are both interpretations of the input (one incorrect, one correct). We refer to input as *ambiguous* when multiple interpretations of that input exist, and the system does not know which of them is correct. Ambiguous input may or may not lead to an error: An error only occurs if the system acts on the wrong interpretation.

The concept of ambiguity is of key importance in this thesis work, because it can be used to *avoid* recognition errors. It is often easier to generate a set of likely alternatives than to know which one of them is correct. Additionally, by maintaining information about ambiguity, we can delay the decision of which interpretation is correct, thus also delaying any potential errors. This delay can be crucial to usability of a recognition-based system, because it allows the system to gather additional information that may eliminate wrong choices. In fact, it can allow the system to be pro-active about getting that information, for example by asking the user and then waiting for a response.

The user interface techniques for mediation found in the literature (See Chapter 2) deal almost exclusively with one common type of ambiguity, which we call recognition ambiguity. However, other types of ambiguity can lead to errors that could be handled through mediation. In addition to recognition ambiguity, we often see examples of target ambiguity and of segmentation ambiguity.

---

<sup>1</sup>For this discussion to apply to rejection and insertion errors, “null” is used as a possible interpretation. In the case of an insertion error, “null” is the correct choice while in the case of a rejection error, it is the (incorrect) top choice



**Figure 1-2:** An example of target ambiguity. Should the selection (green rectangle) include the scrollbar on its border?

---

*Recognition ambiguity* results when a recognizer returns more than one possible interpretation of the user's input. For example, in Figure 1-1, there are five possible interpretations returned by the ViaVoice<sup>TM</sup> recognizer.

*Target ambiguity* arises when the target of the user's input is unclear. A classic example from the world of multimodal computing involves target ambiguity: If the user of a multimodal systems says, "put that there," what does "that" and "there" refer to [84]? This diexis is resolved through another modality (pointing) in most multimodal systems [8, 9, 84, 16, 81], but could also be resolved, for example, through choice mediation. Like recognition ambiguity, target ambiguity results in multiple interpretations being derived from the same source. In the case of recognition ambiguity, it is the user's *intended action* that is in question. In the case of target ambiguity, it is the *target* of that action that is unknown. Figure 1-2 shows an example of this. The user is selecting screen elements with

a rectangle (top), but should things on the border of the rectangle, such as the scrollbar, be included in the selection?

The third type of ambiguity, *segmentation ambiguity*, arises when there is more than one possible way to group input events. If the user writes *a round*, does she mean ‘a round’ or ‘around’? Should the strokes be grouped as two segments (two words), or one? Most systems provide little or no feedback to users about segmentation ambiguity even though it has a significant effect on the final recognition results.

#### **1.1.4 Mediation of ambiguity**

Choosing the wrong possibility from a set of ambiguous alternatives causes an error. Many common errors can be traced to one or more of the three types of ambiguity described above. We call the process of correcting (and avoiding) errors *mediation*, because it generally involves some dialogue between the system and user for correctly resolving ambiguity. We call the particular interface techniques and components used to do this *mediators*. Chapter 2 presents an in-depth survey of existing mediation techniques.

#### **1.1.5 Relationship between recognizers, errors, ambiguity, and mediation**

Now that we have defined recognition, errors and ambiguity, we can address the origins of ambiguity. The system can use ambiguity to predict when errors might occur. Choosing the wrong possibility from a set of ambiguous alternatives causes an error. Rather than setting an unreachable goal for recognizers (always returning the correct answer), ambiguity allows us to define a more accessible goal—generating a set of alternatives that includes the correct answer. Even if the recognizer does not know which alternative is correct, the system may be able to use a process of elimination to find out. It may even ask the user, *via* a mediator. Some possible sources of ambiguity are listed next.

- Many recognizers will return multiple possible answers when there is any uncertainty about how to interpret the user's input. For example, IBM's ViaVoice<sup>TM</sup> and the Paragraph<sup>TM</sup> handwriting recognizer both do this, as do many word-prediction systems [3, 29].
- A separate discovery process may generate alternatives. One example strategy is a *confusion matrix*. A confusion matrix is a table, usually based on historical information about recognizer performance, which shows potentially correct answers that a recognizer may have confused with its returned answer. For example, Marx and Schmandt compiled speech data about how letters were misrecognized into a confusion matrix, and used it to generate a list of potential alternatives for the output of the speech recognizer [52].
- The design of the application may result in ambiguity. For example, if a system uses multiple recognizers, the choices returned by *all* recognizers represent an ambiguous set of alternatives.

The techniques described above are limited to two classes of errors, substitution errors and insertion errors. In both of these cases, problems arise because the recognizer *returned some response* which was wrong. It is much harder to identify rejection errors, where the user intended something to be recognized but the recognizer did not return any response at all (because for some reason it did not notice the input, or discarded it). In this case, user input is often required.

Existing systems, when they deal with errors explicitly, tend to view them quite simplistically. For example, many of the interaction techniques described in the next chapter are limited to substitution or insertion errors and recognition ambiguity. Some other problem areas that are more difficult to handle are listed below. All of these issues can be handled by more sophisticated mediation techniques, and we present examples of this in Chapter 6.

- It is hard to identify rejection errors. (See Chapter 6, page 140, for one solution to this problem).
- There is little existing support for mediating target ambiguity and segmentation ambiguity. (See Chapter 6, pages 140 and 136, for two solutions to this problem).
- It is difficult to guarantee that the correct answer is among the alternatives generated by the system (See Chapter 6, page 127, for a solution to this problem).

## 1.2 Thesis Statement, Contributions and Overview of Dissertation

The preceding set of definitions leads to our thesis statement. After the thesis statement, we discuss the contributions of each chapter in the dissertation, and show how they combine to build an argument for the correctness of this thesis statement.

**A user interface toolkit architecture that models recognition ambiguity at the input level can provide general support for recognition, as well as re-usable interfaces for resolving ambiguity in recognition through mediation between the user and computer. In addition, it can enable the exploration of mediation techniques as solutions to problems not previously handled**

Each piece of the justification of this claim can be mapped onto a chapter of this dissertation. First, we define and motivate the problem:

**Recognition ambiguity** In the current chapter, we showed how recognition errors have been viewed in the past, and illustrated how ambiguity can be used to represent those types of errors. Our goal is to give the reader an understanding of the many types of

errors and ambiguity that can arise when making use of recognizers, and to identify some of the problem areas still remaining in dealing with them.

**Mediation can resolve ambiguity** Once ambiguity has been identified, it must be addressed. We call this process mediation, because it generally involves some dialogue between the system and user for correctly resolving ambiguity. Since ambiguity is derived from the source of errors, mediation can be viewed as a way of correcting or avoiding errors. Chapter 2 presents a survey existing interfaces to recognition systems [50, 49]. This shows that the mediation is a technique used by other researchers and companies, and tells us what mediation techniques should be supported by a comprehensive toolkit. Our survey found that interactive mediation techniques generally fall into one of two classes, repetition and choice. In addition, we discuss automatic mediation techniques, and meta-mediation (approaches to selecting mediation techniques).

### **Toolkit level support for ambiguity can make mediation more accessible**

Chapter 3 details a toolkit architecture that provides reusable support for mediation techniques like those described in the literature (Chapter 2). This architecture models and provides access to knowledge about the ambiguity resulting from the recognition process [51]. Existing user interface toolkits have no way to model ambiguity, much less expose it to the interface components, nor do they provide explicit support for resolving ambiguity. Instead, it is often up to the application developer to gather the information needed by the recognizer, invoke the recognizer, handle the results of recognition, and decide what to do about any ambiguity.

The important contribution of our architecture is the separation of recognition and mediation from application development. A high level summary is that mediation intervenes between the recognizer and the application to resolve ambiguity. This separation of concerns allows us to create re-usable, pluggable solutions to mediation

similar to the menus, buttons, and interactors provided in any graphical user interface (GUI) toolkit. The separation of recognition also leads to the ability to adapt mediation to situations which do not seem to be recognition-based, but where some problem in interaction causes the system to do something other than what the user intended (our ultimate definition of error). The separation of mediation allows us to develop complex mediation interactions independent of both the source of ambiguity and the application.

A second major contribution of our toolkit architecture is a way of maintaining ambiguity until such time as it is appropriate to resolve. In a broad sense, mediators allow the user to inform the system of the correct interpretation of her input. Although recognition may result in multiple potential ambiguous interpretations of a user's input, computer programs are not normally built to handle ambiguity. Because of this, they tend to resolve ambiguity as quickly as possible, and may select the wrong alternative in the process.

The second half of the dissertation explores some important research questions generated by the thesis statement. First, in Chapter 4, we address the question of how the architecture we described can be used to modify a specific existing graphical user interface toolkit to support mediation. We did this by creating the Organized Option Pruning System, or OOPS. In addition, we developed a library of mediation techniques taken from the literature and beyond. OOPS allows an application developer to use third party recognizers, select mediators from its library, and connect both to a standard GUI user interface. The chapter ends with a case study of how OOPS was used to create a sample application that uses recognition.

Next, in Chapter 5, we demonstrate that OOPS is a functional toolkit by re-implementing SILK (Sketching Interfaces Like Crazy) [46, 44], an existing, fairly complex, state-of-the-art recognition-based application taken from the literature [51].

Then, in Chapter 6, we demonstrate that OOPS can allow us to explore problem areas not previously addressed in the literature. These include two problems identified above in our definitions section (rejection errors, target ambiguity), as well as two problems identified through our survey (occlusion, limited choices). Occlusion occurs when a mediator obstructs the view of some other user interface element, while limited choices is an issues when the correct answer is not displayed by a mediator. We developed cross-application solutions to these problems in OOPS [50].

We created two instances of the architecture described in Chapter 3, which demonstrate the ability of our architecture to solve generalized recognition problems for ubiquitous computing. In addition to OOPS, we created CT-OOPS [19], an extension of the context toolkit [75, 18, 20]. Chapter 7 briefly describes how the CT-OOPS toolkit was created and gives details on an example built using CT-OOPS. In creating CT-OOPS, we not only demonstrate the flexibility of our architecture, but we were able to explore a new problem area for mediation, implicit interactions that occur over space as well as time (*e.g.* as the user is walking down a hallway).

We conclude with an analysis of how the work presented in this dissertation fulfills the claims made in the thesis statement, and end with some suggestions for future work.



## Chapter 2

# MEDIATION

Mediation is the process of selecting the correct interpretation of the user's input from among the possible alternatives. Because *correct* is defined by the user's intentions, mediation often involves the user by asking her which interpretation is correct. Good *mediators* (components or interactors representing specific mediation strategies) minimize the effort required of the user to correct recognition errors or select interpretations.

There are a variety of ways that mediation can occur. The first, and most common, is repetition. In this mediation strategy, the user repeats her input until the system correctly interprets it. The second major strategy is choice. In this strategy, the system displays several alternatives and the user selects the correct answer from among them. The third strategy is automatic mediation. This involves choosing an interpretation without involving the user at all. We illustrate each of these strategies in the following sections with examples drawn from the literature. Our survey is followed by a discussion of meta-mediation, the task of dynamically selecting from among the many variations of choice, repetition, and automatic mediation techniques. The chapter concludes by showing how and why this survey suggests the need for toolkit-level support for mediation.

### 2.1 Repetition

Repetition occurs when the user in some way repeats her input. A common example of repetition occurs when the user writes or speaks a word, but an incorrect interpretation appears in her text editor. She then proceeds to delete the interpretation, and dictates the

# multimode correction

The image shows the word "multimode" in a bold, black, sans-serif font. Below the word, the letters "al" are handwritten in a cursive, black ink style, positioned under the "e" of "multimode".

**Figure 2-1:** Mixed granularity repair using a pen in a speech dictation task. ©Dr. Bernhard Suhm.

---

word again. This is the extent of the support for mediation in the original PalmPilot<sup>TM</sup>. Other devices provide additional support such as an alternative input mode (for example, a soft keyboard), undo of the misrecognized input, or variations in granularity such as repair of letters within a misrecognized word.

## 2.1.1 Examples

In this subsection, we will describe how repetition mediation is done in two existing applications. We will then provide a comparison of those and several other mediation techniques, highlighting the dimensions along which they vary.

### 2.1.1.1 Example of multimodal repetition

In Figure 2-1 (©Dr. Bernhard Suhm [81]), the user is dictating text with speech. When a word is misrecognized, the user must notice the mistake by looking at the text of her document. For example, when she says “multimodal correction,” the system may understand “multimode correction.” Once the user notices this, she may use either speech or her pen to correct that mistake. A correction involves first selecting the letter or word that needs to be replaced, or positioning the cursor where text needs to be inserted. In this case, she selects the letter ‘e’ to be replaced. She then corrects the mistake by writing “al” with her pen. Figure 2-1 shows this. The user may also cross out or write over misrecognized letters.

### 2.1.1.2 Example of unimodal repetition

Contrast this to a system such as the PalmPilot<sup>TM</sup>, in which repair of letters is done in the same modality as the original input. In a typical PalmPilot<sup>TM</sup> interaction, the user begins by drawing the gesture for one or more characters. When he notices a mistake, he may draw the gesture for delete, or select the mistaken character. He then draws the gesture for the mistaken character again.

### 2.1.1.3 Comparison of different repetition mediators

In general, repetition systems differ along three dimensions—modality, undo and repair granularity. The first was illustrated above. All three are described in more detail below and illustrated by a set of representative examples in Table 2-1. All of the systems shown in the table also provide UN-mediated repetition, in which the user deletes an entry and repeats it using the original system modality. UN-mediated repetition simply refers to systems that provide no special support for mediation, such as the PalmPilot<sup>TM</sup> described above. In these systems, the user corrects recognition errors exactly as he would correct his own mistakes.

**Modality:** Repetition often involves a different modality, one that is less error-prone or has orthogonal sorts of errors. For example, in the Newton MessagePad<sup>TM</sup>[4], Microsoft Pen for Windows<sup>TM</sup>[59], and in other commercial and research applications, the user may correct misrecognized handwriting by bringing up a soft keyboard and typing the correct interpretation. Similarly, both ViaVoice<sup>TM</sup> and DragonDictate<sup>TM</sup> (two commercial speech-dictation applications), as well as Chatter [52], allow the user to spell text that is misrecognized by speaking the letters, or using military spelling such as “alpha” for ‘a’, “bravo” for ‘b’, *etc.*. If speech fails in DragonDictate<sup>TM</sup> or ViaVoice<sup>TM</sup>, the user may type the word. Chatter, which is a non-GUI phone messaging application, eventually presents the user with a choice mediator (See Section 2.2 for details on choice mediation).

**Table 2-1:** A comparison of different systems in which the user resolves ambiguity by repeating her input (using a repetition mediator).

I/O <sup>a</sup> of recognizer	System	Modality of repair	Undo	Granularity of repair
Handwriting/Words	Message Pad <sup>TM</sup> [4],	Soft keyboard, or individual letter writing	Implicit	Letters
	Microsoft Pen for Windows <sup>TM</sup> [59]			
Speech/Words, Phrases	ViaVoice <sup>TM</sup> [7]	Speech, letter spelling, typing	Implicit	Letters or words
	Subm speech dictation [81]			
Speech/Names (non GUI)	Chatter [52]	Speech, letter spelling, military spelling <sup>b</sup> , with escape to choice	Implicit	Letters
	Word Prediction [3, 29, 64]	Letters (as user enters additional characters, new choices are generated)	Unnecessary (user must explicitly accept a choice)	Letters
POBox [53]				
Multimodal/Commands	“Put That There” [8, 84]	Speech, pointing gestures	Implicit	Commands

<sup>a</sup>I/O gives input/output of the recognizer. For example, the Newton MessagePad<sup>TM</sup>'s recognizer takes handwriting as input and produces a string of words as output.

<sup>b</sup>Military spelling uses easily differentiated words for letters such as “alpha” for ‘a’ and “bravo” for ‘b’

**Granularity of Repair:** In dictation style tasks, it is often the case that only part of the user’s input is incorrectly recognized. For example, a recognizer may interpret “She picked up her glasses” as “She picked up her glass.” In this case, the easiest way to fix the problem is to add the missing letters, rather than redoing the whole sentence. In another example, Huerst *et al.* noted that users commonly correct messily written letters in handwriting, and they built support for applying this style of correction before passing handwriting to the recognizer to be interpreted [36]. Similar techniques were applied by Spilker *et al.* in the speech domain [78]. Figure 2-1 shows an example of repair in which the *granularity* of correction is much smaller than the granularity of recognition. The user is correcting one character, where the recognizer produced a word phrase [81]. Thus, the granularity of the repair (letters) is smaller than the granularity of recognition (words). The opposite may also be true—A user might enter letters, but correct entire words.

**Undo:** Depending upon the type of application, and the type of error, repetition may or may not involve undo. For example, repetition is often used when the recognizer makes a rejection error (the recognizer does not make any interpretation at all of the user’s input), and in this case, there is nothing for the user to undo. In contrast, in very simple approaches to mediation, the user may undo or delete her input before repeating it (*e.g.* PalmPilot<sup>TM</sup>, “scratch that” in DragonDictate<sup>TM</sup>). In some situations, such as when entering a command, it is essential that the result be undone if it was wrong (what if a pen gesture representing “save” were misinterpreted as the gesture for “delete?”). In other situations, it may not matter. For example, if the system misinterprets “save” as “select”, the user may simply redraw the same gesture.

### 2.1.2 Issues and problems

Repetition is the simplest possible approach to handling errors, and from that perspective, it fulfills its role well. However, when repetition is used without the option to switch to a less error-prone modality, the same recognition errors may happen repeatedly. In fact, research has shown that a user's input becomes harder to recognize during repetition in speech recognition systems because he modifies his speaking voice to be clearer (by human standards) and, therefore, more difficult for the recognizer to match against normal speech [25]. A similar study showed that recognition remains at least as bad for pen repetition as it is for initial pen inputs [24]. Our solution to this problem is to do guided rerecognition, in which the repetition mediator tells the recognizer which input should be reinterpreted. We demonstrate an example of this in Section 6.4, p. 140. Guided rerecognition can be used to generate a new set of alternatives in the case of a substitution error. It can also be used to correct for a rejection error, since the recognizer is being explicitly told to interpret the user's input. The recognizer can use information about which input is being repeated to eliminate the original choice or choices and generate new ones.

In many cases, switching modalities involves significant cognitive overhead. For example, the user may have to first bring up a separate window or dialogue, and then interact with that before going back to the task at hand. Getting the user (especially a novice) to switch to these alternative modalities may be difficult [81, 30]. There are examples of alternate modalities that are more integrated with the rest of the application, and thus less awkward. In his speech dictation application, Suhm allowed users to edit the generated text directly by using a pen [81]. Even so, his studies found that users tended to repeat their input at least once in the original modality [79].

For recognition tasks that do not involve data entry (such as commands), the problem of undo becomes more difficult. The user may not know exactly what happened, even if he realizes that the system did not do what he intended. He also may not know exactly how

to undo the result (different commands may have to be undone in different ways). If undo is implicit, the system faces similar problems. In addition, the system needs to determine how much to undo. Should only the last atomic action be undone, or if the user did several related commands in sequence, should they be treated as a group?

One way to evaluate the tradeoffs between different approaches to repetition involves doing empirical user studies. Zajicek and Hewitt found that users prefer to repeat their input at least once before having to choose from a menu [86], a finding confirmed by Ainsworth and Pratt [2]. Rudnicky and Hauptman present an analytical approach to this problem [74]. They use a state diagram to represent the different phases of a correction during repetition. This can then be translated into a mathematical expression that relates recognition accuracy and input speed. Different state diagrams will result in different relationships. Baber and Hone give a good overview of the pros and cons of repetition versus choice [6].

## 2.2 Choice

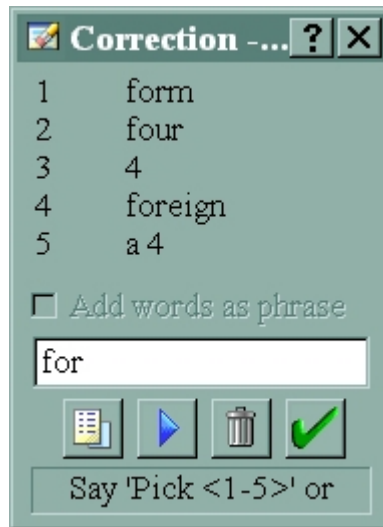
Choice user interface techniques give the user a choice of more than one potential interpretation of his input. One common example of this is an  $n$ -best list (a menu of potential interpretations) such as that shown on p. 2. We have identified several dimensions of choice mediation interfaces including layout, instantiation time, additional context, interaction, and format [50].

### 2.2.1 Examples

Two systems that differ along almost all of the dimensions just described are the ViaVoice<sup>TM</sup>[7] speech recognition system (See Figure 1-1), and the Pegasus drawing beautification system [37] (See Figure 2-2). We briefly describe and compare them, and then use them to illustrate the choice dimensions. We then illustrate how a range of systems vary along those dimensions.

### 2.2.1.1 Description of ViaVoice™ mediation

ViaVoice™ is an application-independent speech dictation system. It comes with a variety of mediation techniques [7], but we will focus on the one shown in Figure 1-1, and illustrated below. In our hypothetical example, the user is using ViaVoice™ in dictation mode and has spoken a sentence containing the word “for.” In our example, the top choice is “form,” and this is the word that was inserted into the user’s document. At some point (possibly after



more words have been spoken), the user sees that a mistake has been made, and needs to initiate mediation. ViaVoice™ provides several ways to mediate such mistakes, one of which is the choice mediator shown above. The user can request it *via* a vocal command or set ViaVoice™ up so that the mediator is always present. We’ll assume the user has chosen the latter approach. In this case, before the mediator will be useful, the user has to select the misrecognized word (either by speaking or with the cursor). Once the mediator displays the set of choices the user wishes to correct, she again has several options available to her. If the correct choice is present, she can say the words “Pick <*n*>” in order to select it from the list. If it is not present, she may enter it in the text area at the bottom of the mediator (either by spelling it or typing it). In either case her document will be updated.

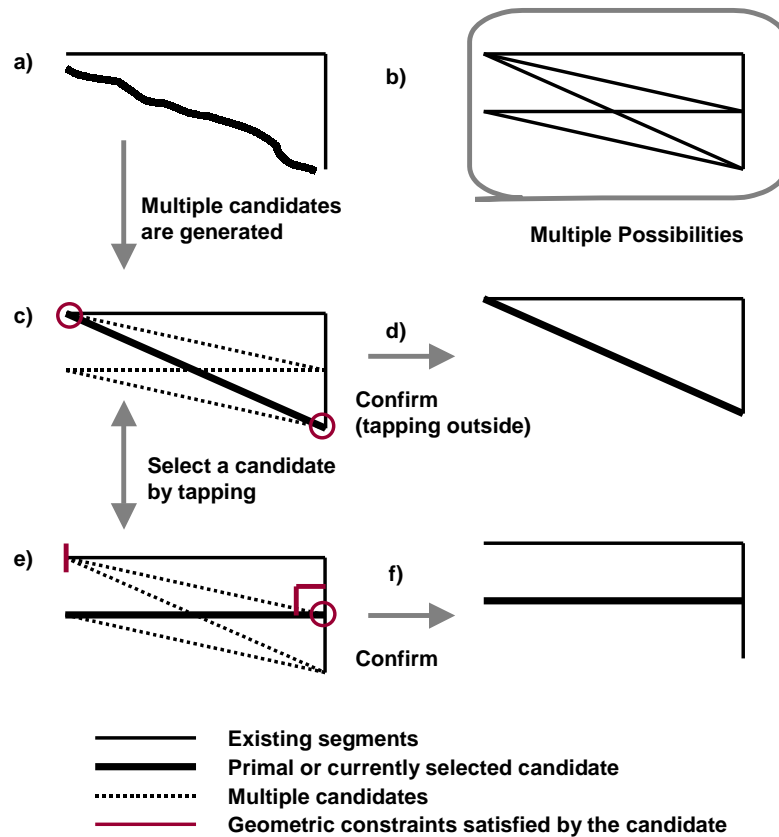


### 2.2.1.2 Description of Pegasus mediation

Pegasus recognizes user input as lines. Pegasus is a system that beautifies lines sketched by the user by straightening them out and aligning them with other lines that have already been drawn. This allows users to sketch geometric designs more easily and rapidly. Each time the user sketches a line, the Pegasus recognizer uses a set of constraints to generate different lines that might have been intended. Figure 2-2 (adapted from [37], Figure 7) illustrates a sample interaction. Figure 2-2 (a) shows an example of a current sketch along with two existing lines. Figure 2-2 (b) shows the possibilities generated by the recognizer. Figure 2-2 (e) shows the result of the interaction. Everything between (b) and (e) is mediation. As soon as the recognizer generates multiple choices, it displays them to the user (c). It highlights the top guess, which will be selected automatically if the user starts to draw another line. It also shows the particular constraints that were used to generate that guess. If the user does not like the highlighted guess, she may select a different one by tapping on it. Figure 2-2(d) shows an example of this—because the user tapped on the middle horizontal line, it has been highlighted and its associated constraints added to the drawing. The user can confirm by drawing another line or tapping outside of the figure, leading to (e).

### 2.2.1.3 Comparison of different choice mediators

Table 2-2 gives a brief overview of some commercial and research systems with graphical output, and illustrates how they differ along the dimensions of layout, instantiation, context, interaction, and format. Each system we reference implemented their solutions in isolation, but as Table 2-2 makes clear, the same design decisions show up again and again. These design decisions, which illustrate the dimensions of a choice interface, are described below. First, we contrast Pegasus and ViaVoice<sup>TM</sup> in order to illustrate the dimensions informally, and then we give more specific definitions and highlight how other existing work falls within them.



**Figure 2-2:** The mediator used in the the Pegasus system (Figure 7 in [37]). Adapted with permission of the author. The Pegasus recognizer recognizes sketched lines as straight line segments conforming to certain geometric constraints (Pegasus may also be used for prediction of future line segments). **(a)** The user draws a line. **(b)** The system generates multiple possibilities. **(c)** The choices are represented as lines on the screen, with the top choice selected and highlighted in bold. **(e)** The user can click on another line to select it. **(d, f)** Drawing another line or clicking elsewhere on the screen accepts the currently selected line.

## Comparison of ViaVoice<sup>TM</sup> and Pegasus

**Layout:** ViaVoice<sup>TM</sup> uses a menu layout that is always visible on screen, but in a separate window from the recognized text. In contrast, Pegasus does layout “in place”. Possible lines are simply displayed in the location they will eventually appear if selected (Figure 2-2(c,e)).

**Instantiation time:** The  $n$ -best list can be instantiated by a speech command, or can be always visible (even when no ambiguity is present). Instead of waiting for an error to occur, Pegasus shows the alternative lines as soon as they are generated. It is essentially informing the user that there is some ambiguity in interpreting her input, and asking for help in resolving it. Unlike ViaVoice<sup>TM</sup>, there is no mediator visible in Pegasus when there is no ambiguity present.

**Contextual information:** Pegasus also shows contextual information about the lines by indicating the constraints that were used to generate them (Figure 2-2(c,e)). The  $n$ -best list, which is the most standard type of choice mediator, shows no additional information.

**Interaction:** In both examples, interaction is quite straightforward. In ViaVoice<sup>TM</sup>, the user says “Pick  $\langle n \rangle$ .” (thus specifying the alternative’s number in the  $n$ -best list). In Pegasus, the user can select a choice by clicking on it (Figure 2-2(e)). Drawing a new line will implicitly accept the currently selected choice in Pegasus (Figure 2-2(d,f)). The  $n$ -best list is only used to correct errors, the top choice is always sent to the current document as soon as recognition is complete.

**Format:** As stated earlier, feedback in Pegasus is in the form of lines on screen. Contrast this to the ASCII words used in the  $n$ -best list.

**Table 2-2:** A comparison of different systems that resolve ambiguity by offering the user a choice of multiple potential interpretations of her input (using a choice mediator).

I/O	System	Layout	Instantiation	Context	Interaction	Format
Handwriting/Words	MessagePad™ [4]	Linear menu	Double click	Original ink	Click on choice	ASCII words
Speech/Words	ViaVoice™ [7]	Linear menu	Speech command/continues	None	Speech command	ASCII words
Speech/Commands (non-GUI)	Brennan & Hulteen [11]	Spoken phrases	On completion	System state (audio icons)	Natural language	Pos.&Neg. natural lang. evidence
Handwriting/Characters	Goldberg <i>et. al</i> [27]	Below top choice	On completion	None	Click on choice	ASCII letters
Characters/Words (Word-prediction)	Assistive Tech [3, 29] POBox [53] Netscape [64]	Bottom of screen (grid)	Continuously	None	Select choice (command, click)	ASCII letters
		In place			Return to select, arrow for more	
Gesture/Commands	Marking Menu [43]	Pie menu	On pause	None	Flick at choice	Commands, ASCII letters
Gesture/Lines	Beautification [37]	In place	On prediction/completion	Constraints	Click on choice	Lines
Context/Text	Remembrance Agent [71]	Bottom of screen, linear menu	Continuously	Certainty, result excerpts	Keystroke command	ASCII sentences
UI description/Interface specification	UIDE [82]	Grid	On completion	None	Click on choice	Thumbnails of results
Multimodal/Commands	QuickSet [66]	Linear menu	On completion	Output from multiple recognizers	Click on choice	ASCII words
Email/Appointments	Lookout [32]	Pop up agent, speech dialogue box	On completion	None	Click OK	ASCII words

## Comparison of choice mediators found in the literature

**Layout** describes the position and orientation of the alternatives in the choice mediator on the screen. The most common layout is a standard linear menu [7, 4, 71, 66] (See Figure 1-1). Other menu-like layouts include a pie menu [43], and a grid [3, 29, 82, 53]. We also found examples of text floating around a central location [27], and drawings (in the location where the selected sketch will appear) [37] (See Figure 2-2). Another variation is to display only the top choice (while supporting interactions that involve other choices) [27]. In Lookout, the choices are represented in a dialogue box [32]. Finally, choices may be displayed *via* an audio menu or *via* spoken natural language [11].

**Instantiation time** refers to the time at which the choice display first appears, and the action that causes it to appear. Variations in when the display is originated include: on a double click [4], or other user action such as pause [43] or command [82]; based on an automatic assessment of ambiguity [32]; continuously [7, 3, 29, 64, 71, 53]; or as soon as recognition is completed [27, 37, 11, 66]. If instantiation is delayed, there is still the decision of whether to display the top recognition choice during the wait time, thus indicating to the user that recognition has occurred, or simply to leave the original, unrecognized input on the screen.

**Contextual information** is any information relating to how the alternatives were generated or how they will be used if selected. Additional context that may be displayed along with the actual alternatives includes information about their certainty [71], how they were determined [37], and the original input [4]. Brennan and Hulteen use audio icons to give the user additional clues about system state [11]. Most of the systems we surveyed did not provide any additional context beyond the actual alternatives.

**Interaction**, or the details of how the user indicates which alternative is correct, is generally done with the mouse in GUI settings. Goldberg and Goodisman suggest using

a click to select the next most likely alternative even when it is not displayed [27]. Other systems allow the user to confirm implicitly the indicated top choice simply by continuing her task [27, 37]. This is also true for any system in which the user must explicitly instantiate the choice mediator [4, 7, 43]. In cases where recognition is highly error-prone, the user must select something to confirm, and can implicitly contradict the suggested interpretation [3, 29, 64, 71, 32, 53]. Some of these systems have tiered interaction in which the default changes based on a system determination of how likely an error is [11, 32]. In non-graphical settings interaction may be done through some other input mode such as speech [11].

**Format** is the method of displaying each individual interpretation. This generally correlates closely to how an interpretation will look if it is selected. Text is used most often [3, 4, 7, 27, 29, 64, 71, 66, 32], but some interpretations do not map naturally to a text-based representation. Other variations include drawings [37], commands [43], icons [82], and mixtures of these types. In addition, format may be auditory. For example, Brennan & Hulteen use natural language to “display” multiple alternatives [11].

### 2.2.2 Issues and problems

By identifying this design space, we can begin to see new possibilities. For example, although Table 2-2 shows that the continuous instantiation style has been used in text-based prediction such as Netscape’s word-prediction [64], and the Remembrance Agent [71], to our knowledge it has not been used to display multiple predicted completions of a gesture in progress. In fact, most gesture recognizers do not even generate any alternatives until a gesture is completed.

Further research in choice displays needs to address some intrinsic problems. First, not all recognized input has an obvious representation. How do we represent multiple possible

segmentations of a group of strokes? Do we represent a command by its name, or some animation of the associated action? What about its scope, and its target? If we use an animation, how can we indicate what the other alternatives are in a way that allows the user to select from among them?

Second, what option does the user have if the correct answer is not in the list of alternatives? One possibility is to build a mediator that lets you switch between choice and repetition. Essentially it is a conglomeration of mediation techniques with the positives of both. An example of this is the ViaVoice<sup>TM</sup> mediator (see Figure 1-1). Another possibility is to make choice-based mediation more interactive, thus bringing it closer to repetition. For example, an improvement to the Pegasus system would be to allow the user to edit the choices actively by moving the endpoint of a line up and down. The result would allow the user to specify any line, just like repetition does. Section 6.1 describes how we modified an  $n$ -best list to allow specification of new alternatives in a word-prediction setting.

Finally, there is the question of where to put a choice mediator. Most choice mediators, like dialogue boxes, are temporary displays of information. If an application is making good use of screen space, there may not be enough free space to show the choice mediator without occluding something else. Section 6.2 describes one possible solution to this problem.

## 2.3 Automatic Mediators

Automatic mediators select an interpretation of the user's input without involving the user at all. Any recognizer that only returns one choice is doing automatic mediation. Similarly, a system that selects the top choice from the recognition results, and discards the rest is automatically mediating that ambiguity.

### 2.3.1 Examples

Three classes of automatic mediators are commonly found in the literature, and described below.

**Thresholding:** Many recognizers return some measure of their confidence in each interpretation. If this can be normalized to a probability of correctness, the resulting probability can be compared to a threshold. When an interpretation falls below the threshold, the system rejects it [70, 11, 6].

**Rules:** Baber and Hone suggest using a rule base to determine which result is correct [6]. This can prove to be more sophisticated than thresholding since it allows the use of context. An example rule might use syntactic information to eliminate words that are grammatically incorrect. Because rules often depend upon linguistic information, they benefit from knowing which words are definitely correct to use as “anchors” in the parsing process. Thus, they may benefit from use in combination with an interactive mediator. Those words that the user mediates can become anchors, since they are known to be correct.

**Historical Statistics:** When error-prone recognizers do not return a measure of probability, or when the estimates of probability may be wrong, new probabilities can be generated by performing a statistical analysis of historical data about when and where the system made mistakes. This task itself benefits from good error discovery. A historical analysis can help to increase the accuracy of both thresholding and rules. This approach may be used to enhance thresholding or rules. For example, a confusion matrix could update certainty values before applying a threshold [52], or to add new alternatives. In general, historical statistics may provide a default probability of correctness for a given answer when a recognizer does not. More sophisticated analysis can help in the creation of better rules or the choice of when to apply certain rules.



### 2.3.2 Issues and problems

One problem with automatic mediation is that it can lead to errors. Rules, thresholding, and historical statistics may all lead to incorrect results. Even when the user's explicit actions are observed, the system may incorrectly infer that an interpretation is incorrect (or correct). Only when the user explicitly notifies the system about an error, can we be sure that an error really has occurred, in the user's eyes. In other words, all of the approaches mentioned may cause further errors, leading to a cascade of errors.

Another issue is how to provide the necessary information to these mediators. Examples include (for rules) application specific context; and the results of interactive mediation to be used as anchors; or (for historical statistics) the history of past errors.

## 2.4 Meta-mediation: Deciding When and How to Mediate

The goal of meta-mediation is to minimize the impact of errors and mediation of errors on the user. Studies have shown that recognizers tend to misunderstand a some inputs much more than others, both in the realm of pen input [24] and speech input [52]. For example, *u* and *v* look very similar in many users' handwriting, and because of this may be more likely to be misrecognized. A meta-mediator might use interactive mediation only for the error prone subset.

Meta-mediators dynamically decide which mediators to use when. Note that multiple mediators may be required to resolve an entire set of ambiguous events (each eliminating only some of the choices). In this case, a meta-mediation system will decide not only which mediators should be called, but in what order they should be called.

### 2.4.1 Examples

Horvitz uses a technique called *decision theory* to provide dynamic, system-level support for meta-mediation [32]. Decision theory can take into account dynamic variables like the

current task context, user interruptability, and recognition accuracy to decide whether to use interactive mediation or just to act on the top choice. Horvitz refers to this as a mixed-initiative user interface [32]. Simpler heuristics, such as filtering techniques that select a mediator based on the type of ambiguity or interpretation present, may also be important components of a meta-mediation system. Most other systems simply have hard coded decisions about when and which mediators are chosen, rather than more dynamic, intelligent approaches.

#### **2.4.2 Issues and problems**

In designing a meta-mediation strategy, it is difficult to strike the correct balance between giving the user control over the mediation dialogue, and limiting the negative impacts of interrupting them and asking them unnecessary questions. This balance is very specific to both task and situation, and must be determined through a combination of experimentation and experience. It is our hope that the existence of a pluggable toolkit such as the one described in this dissertation will make this type of experimentation more feasible.

### **2.5 The Need for Toolkit Support for Mediation**

As we have shown, a wide variety of techniques for mediating recognition errors can be found in the literature. Anyone trying to design an interface that makes use of recognition has a plethora of examples from which to learn. Many of these examples are backed up by user studies that compare them to other possible approaches. Since most of the mediation strategies found in the literature can be placed in variations of three categories — choice, repetition, or automatic mediation — there is a lot of potential for providing re-usable support for them. This suggests the need for a library of extensible, reusable techniques drawn from our survey. Additionally, all of the techniques described here need access to recognition results, and in many cases other information as well, such as when other

ambiguity is resolved. This suggests the need for an architecture, including a model of ambiguity and algorithms which can maintain information about ambiguity and inform the correct parties when it is resolved.

We found few examples of toolkit-level support for these types of applications, particularly at the graphical user interface level (multi-modal toolkits, such as the Open Agent Architecture [66], do not usually deal directly with interface components). The next chapter describes the architecture we designed to support mediation. Chapter 4 describes the toolkit that we built by implementing this architecture and combining it with a library of mediation techniques. Chapter 4 concludes with the first of a series of examples intended to illustrate progressively the complexities of using the toolkit. Chapter 5 focuses on one very complex application and in Chapter 6 we delve into a series of sophisticated mediation strategies. Finally, Chapter 7 illustrates how the toolkit and mediation could be used in an entirely new domain.

## Chapter 3

# AN ARCHITECTURE FOR MEDIATION

A key observation about the mediation strategies described in the previous chapter is that they can be described in terms of how they resolve ambiguity. This is important, since, as discussed in Section 1.1.3, we can model most recognition errors in terms of ambiguity. Mediation strategies may let the user choose from multiple ambiguous interpretations of her input, or add new interpretations. Interfaces to error-prone systems would benefit tremendously from a toolkit providing a library of mediators that could be used and re-used, or adapted, when error-prone situations arise. This chapter describes the architectural solutions necessary to support mediation. The goal of this chapter is to detail one of the contributions of this thesis: An architecture which can handle and maintain information about ambiguous input, while providing flexible and dynamic support for mediating that ambiguity when appropriate.

The design of this architecture was heavily influenced by typical architectures for GUI toolkits because of the maturity and robustness of support for interaction and input handling in these toolkits. In addition, our main goal in developing this architecture was to create a toolkit that would support the design of GUI interfaces that handle ambiguity in recognition. Note that we are not interested in toolkit-level support for building new recognizers, we are interested in supporting the use of existing recognition technologies. Our users are people who will write applications, not recognizers, using our toolkit. We refer to them as *designers* in this document. Their users are the people who will use the programs they create, and are referred to as *end users* in this document.

Although this architecture is presented in the abstract, we have built two concrete instances of it, OOPS and CT-OOPS which are discussed later in this dissertation (Chapters 4 and 7, respectively). OOPS is a GUI toolkit and the main artifact of this thesis work. CT-OOPS is a context-aware toolkit based on the Context Toolkit [75, 18, 20], and our goal in building it was to show that our model of ambiguity could be used in other settings. Work done with OOPS and CT-OOPS, along with the rest of this dissertation, can be viewed as a graded set of problems that progresses through more and more complex examples illustrating what this architecture can accomplish.

We begin this chapter by examining how existing toolkit architectures support recognized input (Section 3.1). This leads to a set of four major modifications to standard toolkit architectures, described in Section 3.2–3.4. We start by describing the (event) data structure used for communication between disparate components of the system such as recognizers and mediators (p. 41). We then explain how events are dispatched (p. 49), and how they are mediated (p. 51), and we end with a brief overview of the support provided for recognizers to communicate about interpretations with the application and mediators. The last section of the chapter, Section 3.6, summarizes the major contributions of our architecture, which in turn is the major contribution of this thesis.

### **3.1 Existing Support for Recognized Input**

This summary includes two major areas of toolkit architecture support for the *use of* recognition: GUI toolkit architectures and toolkit architectures that facilitate application use of recognition. An overview of the toolkit architectures described in this section shown in Table 3-1.

**Table 3-1:** A comparison of different toolkits and architectures supporting recognition. ✓ indicates full support for a feature, ✗ indicates no support. *Recognizers*—number of recognizers that may be chained together to interpret each other’s results. *Mediation*—interactive mediation (*I*: default that cannot easily be modified; *A*: only automatic (non-interactive) mediation) *Ambiguity*—support for making delayed decisions about uncertain information. *GUI*—support for integrating recognition results into the event dispatch system used by GUI components. Does not apply to Put That There [8, 84], the OAA [66] and CTK [20], which are *non-GUI* toolkit architectures.

Toolkit/Arch.	Recognizers	Mediation	Ambiguity	GUI
(1980) Put That There [8, 84]	✓	1	✓	N/A
(1990) Artkit [31]	✓	A	✗	Partial
(1992) PFSM [34]	N/A	✗	✓	Partial
(1992) Pen for Windows™[59]	1	1–2	✗	Partial
(1993) Amulet [45, 60]	1	A	✗	Partial
(1994) OAA [17, 66, 38, 56]	✓	1	✓	N/A
(1997) subArctic [35, 22]	✗	✗	✗	✗
(1997) ViaVoice™SDK	1	✗	✗	✗
(1999) CTK [20]	✓	✗	✗	N/A
(2000) OOPS [51]	✓	✓	✓	✓

### 3.1.1 GUI toolkit architectures

Mouse and keyboard input in GUI toolkit architectures is traditionally split into discrete events and delivered *via* an input handler directly to interactors, also called widgets, or interactive components, in the literature [31, 60]. This is an effective approach that saves the application developer the job of manually routing mouse and keyboard input. For example, if the user clicks on a button, the toolkit architecture handles the job of transmitting the mouse click information to the button interactor. When recognizers are added to this mix, two problems arise. First, something has to invoke the recognizer at the right time (for example, after the user has drawn a stroke). Second, something has to take the recognition results and send them to the proper places. The second problem is similar to the problem solved by GUI toolkit architectures for keyboard and mouse input, and similar solutions may be applied.

One of the earliest GUI toolkits to solve both of these problems was the Arizona Retargetable Toolkit (Artkit) [31], a precursor to subArctic [35, 22] (on which OOPS is based). Artkit grouped related mouse events, sent them to a gesture recognition object, and then sent the results of recognition to interactors or other relevant objects. Artkit took the important step of integrating the task of invoking the gesture recognition engine into the normal input cycle, an innovation repeated later in Amulet [45]. In addition, objects wishing to receive recognition results did this through the same mechanism as other input results. However, since Artkit did not support ambiguity, the recognizer was expected to return a single result. Also, gesture results were a special type of input event not understood by most interactors, and thus still had to be handled specially in many situations.

An architecture supporting ambiguity was introduced in the work of Hudson and Newell on probabilistic state machines for handling input [34]. This work is most applicable to displaying visual feedback about ambiguous information and is intended to be integrated into the event handlers that translate input events from lexical into semantic events.

There are also commercial toolkits for pen input that have some support for a specific recognizer/mediation combination [58]. Examples include Microsoft Pen for Windows<sup>TM</sup>[59], and the Apple MessagePad<sup>TM</sup> development environment [4]. In general, these come with default mediators intended for use by all applications. In the speech world, commercial toolkits such as IBM ViaVoice<sup>TM</sup> provide application programming interfaces (APIs) that allow programs to receive recognition results or modify the recognition grammar.

### 3.1.2 Other toolkit architectures

In addition to toolkit support for building recognition-based interfaces, it is useful to consider toolkit support for separating recognition from the application. Multi-modal toolkits generally support multiple levels of recognition (recognizers that interpret events produced by other recognizers), and focus on providing support for combining input from multiple diverse sources [66, 65]. Multi-modal input generally combines input from a speech recognizer and one or more other input modes. The additional modes may or may not involve recognition, but because of speech, there is always at least one recognizer present. Bolt used a combination of speech and pointing in his seminal paper on multi-modal input [8]. In contrast, the Quickset system [16], which uses the Open (or Advanced) Agent Architecture [17], combines speech with gesture recognition and natural language understanding. Another class of toolkit architectures with similar goals is context-aware computing toolkits [76, 20, 13, 63, 77], summarized in [18]. These toolkit architectures take input from sensors, and pass it to “recognizers” to be interpreted, in order to modify application behavior to take advantage of dynamic information about the end user and her environment. Both raw input and interpretations may be delivered to the application.

Explicit support for ambiguity was addressed in the Open Agent Architecture: Oviatt’s work in mutual disambiguation uses knowledge about the complementary qualities of different modalities to reduce ambiguity [66]. In addition, McGee, Cohen and Oviatt experimented with different confirmation strategies (essentially interactive mediation) [56].



### 3.1.3 In summary

All of the toolkit architectures described in this overview provide support at some level for routing input (either mouse and keyboard events, recognition results, or sensed input) to interested parties. Table 3-1 summarizes the contributions most relevant to this thesis. The ability to route input, regardless of whether it was created by a recognizer or a standard interface device, without requiring “specialized” intervention from the application designer, is an important requirement taken from this past work. The last column of Table 3-1 shows which toolkits support some aspect of this process. A complete solution would provide backward compatibility with existing interfaces even when the input device is changed, as long as the new input device or recognizer produces events of the same type as the previous input device (*e.g.* mouse or keyboard).

Most of the non-GUI toolkits (the toolkits that do not provide any support for building graphical interfaces) also provide support for routing recognition results to other recognizers to be recognized, *ad infinitum*. This is another important requirement for our work, because it makes it possible to chain different third party recognizers together in interesting ways. For example, a recognizer that translates strokes to characters could be chained to a recognizer that converts characters to words, without either recognizer having to know about the other. This becomes especially important when there is uncertainty and a need for correction, although existing systems. The second column (Recognizers) of Table 3-1 shows which toolkits support this.

Recognizers generally generate uncertain results, and many recognizers actually produce multiple possible interpretations of user input. Two of the architectures described in this survey provide some support for modeling ambiguity (See the column labeled “Ambiguity” in Table 3-1). We believe that the choice of when and how to resolve ambiguity should be up to the application designer and not constrained by the toolkit architecture.

Without full support for ambiguity, the application has to commit to a single interpretation of the user's input at each stage, throwing away potentially useful data earlier than necessary, or requiring the application designer to build solutions from the ground up for retaining information about ambiguity. This lack of support also makes it difficult to build re-usable solutions for mediation. The difficulty arises because application designers must either resolve ambiguity at certain fixed times or explicitly track the relationship between ambiguity and any actions taken at the interface level. The third column (Mediation) of Table 3-1 shows which toolkits support this.

Resolution of ambiguity generally happens through mediation, and the choice of how to mediate should also be flexible. Two of the toolkits in Table 3-1 provide one or two example mediators that the application designer can use. None of the toolkits provide the kind of re-usable, pluggable support for mediation necessary to allow interface designers to add to the possible choices for mediation.

In conclusion, what we learn from this overview is that few toolkit architectures provide a principled model for dealing with recognition or with the ambiguity resulting from recognition, and none integrate this into the toolkit input model shared by on-screen components and event handlers. For example, looking at Table 3-1, only the architecture described in this dissertation supports any number of combination of *recognizers* and *mediators*, and supports *ambiguity* and *GUI integration*, although each is supported individually in some other toolkits. PFSM (work involving probabilistic state machines) comes closest to the goals of this thesis work [34]. However, this architecture that was never integrated into a full-fledged toolkit.

The architecture presented in this chapter tries to support the best features of each of the toolkits we surveyed, and this means it has to include unique elements, not found anywhere else. Our goal is to develop an architecture that supports ambiguous input and provides a separation of concerns that keeps mediation, recognition, and the application independent. The separation of concerns allows individual recognizers and mediators to be

used with a new application, or to be modified or replaced without requiring significant changes to an existing application. This goal required three major modifications to existing typical toolkit architectures:

**The event hierarchy:** The first modification is to the structure or object used to represent an event. An event is now part of a graph of other events, all related because they are interpretations of something that the user did. This event graph is a crucial communication mechanism used to support the separation of concerns provided by our architecture, and it is also the place where information about ambiguity is stored.

**Event dispatch:** The second modification is to the event delivery system of the toolkit architecture. Unlike standard event dispatch, the dispatch system must allow dispatched events to be interpreted (by recognizers), and then recursively dispatch these interpretations. Additionally, it automatically identifies when ambiguity is present and asks the mediation subsystem to resolve it.

**Mediation subsystem:** The third modification is the addition of a mediation subsystem responsible for resolving ambiguity. This includes an algorithm for deciding which mediator resolves ambiguity when, support for informing the producers and recipients of events when those events are accepted or rejected, and a library of common mediation techniques (based on the survey presented in Chapter 2).

The rest of this chapter will talk about each of these three major modifications in detail, followed by a discussion of some additional issues that are less central. We will use an example of a word predictor that is being used for a dictation task as a running example. A word predictor tries to predict what text the user is entering from the initial characters he has typed [42, 5, 53, 26, 54, 55, 57].

## 3.2 The Event Hierarchy

An event, in the sense meant by most toolkits, is a record of a significant action. It generally represents some discrete, simple piece of data that the system has sensed, such as a key press or change in location of the mouse. In this architecture, that concept is retained, with slight modifications. Events not only include sensed data, but also generated, or interpreted, data: When a recognizer of some sort interprets data, the results of that interpretation are also considered to be events. Here, we define a *recognizer* as a function that takes one or more events and produces one or more interpretations of those events (which are themselves considered events). For example, in the GUI world, a recognizer might start with a series of mouse events and produce text (See Figure 3-1 for an example of this). It could start with text and produce more text (A word predictor such as that described in Section 4.5 is an example of this). Or it could start with audio and produce mouse events, which might cause a button to depress. DragonDictate<sup>TM</sup> does this, allowing users to control the mouse with their voices).

Events that represent interpreted data may be ambiguous. We represent the level of ambiguity of an event in two ways. First, it has an associated *certainty*, a probability that represents how likely a recognizer thinks it to be true, as compared to other alternatives. Second, it has a *closure* value, a logical value that indicates whether it is right or wrong or still ambiguous. If it is *closed* (because it has been *accepted* or *rejected* by the user), then it is no longer ambiguous. If it is *open*, then it is still ambiguous.

Traditionally, an event is given an identifier, such as a key press of an 'f' key. In our architecture, this is augmented by a typing mechanism. For example, an event may be of type *characters*. Objects that receive events can register interest by type. Because the type of an event may be inherited, a recognizer may create special events, but associate them with an existing, known class. This means, for example, that an interactor need not know whether a piece of text is generated by a keyboard or a speech recognizer as long as in both

**Table 3-2:** The most important information associated with each event.

**closure:** its current closure state (accepted, rejected, open)

**certainty:** its certainty estimate (given by the producer)

**id:** a unique id identifying the event type

**interpretations:** a set of events, generated by recognition of this event

**sources:** a set of events of which this is an interpretation.

**producer:** the object (recognizer, sensor, etc) which created this event

**user\_data:** other important data which can be stored in the event and retrieved by whoever needs it. This provides a customizable communication mechanism, when necessary, to augment the more general systems for communication provided automatically by this architecture.

**feedback:** some way of displaying the event. This simplifies the process of developing a generic mediator.

---

cases it inherits from *characters*.

In addition to type, each event retains information about the relationship between itself and any interpretations of it. The result is a graph, generally referred to in the literature as a hierarchical event graph [61].

So, in summary, the concept of an event has been modified in three critical ways. First, it may represent uncertain (ambiguous), interpreted data as well as directly sensed data. Second, it may be identified both by a unique identifier, and also by its inherited type. Third, it is part of a hierarchy of related events. We will discuss the specifics of each of these modifications, and then explain how our modifications allow us to handle the types of ambiguity and errors normally found in recognition-based systems.

### 3.2.1 The event object

Table 3-2 shows a list of some of the most important data associated with each event object, data that facilitates ambiguity, typing, and the creation of an event hierarchy.

**Ambiguity:** We represent ambiguity in an event using the concepts of *closure* and *certainty*. *Closure* is the system’s representation of whether an event is ambiguous (open), or not ambiguous (closed). An event becomes closed once it is accepted or rejected. *Certainty* is an estimate of how likely a given event is to be correct, generally supplied by the recognizer.

**Type:** We represent the type of an event in two ways. *id* is one component of an event’s type. The other critical piece of information about an event’s type is not stored in an event: it is the event’s place in the inheritance hierarchy (this static type information is distinct from an event’s place in the hierarchy of interpretations, which is assigned dynamically).

**Event hierarchy:** Finally, each event is part of a larger event hierarchy. This graph is represented using a node and link model in which each event is a node, and contains pointers (links) representing edges to other events in the graph. *Sources* is a set of all of the incoming edges in the graph. *Interpretations* is a set of all of the outgoing edges in the graph. An edge going from an event to its interpretation indicates that a recognizer produced the interpretation from the event.

For example, if the user has typed the letter ‘f’, and the word predictor thinks that this is “farther” with a 30% certainty, “father” with a 50% certainty, and “further” with a 20% certainty, the word predictor would create three events. All three events would be of type **characters**, with the id F. They would be **open** since the question of which is correct has not yet been resolved. The certainty for “farther” would be .3, while “father” would be .5 certain and “further” would be .2 certain. All three events would share the same source event (‘f’), and have no interpretations. Correspondingly, ‘f’ would be updated to have “farther,” “father” and “further” as interpretations.

In addition to the important changes described above, events store three other pieces of information: their producer, additional user data, and a feedback object. While these

are not essential for modeling ambiguity, they help the toolkit architecture to support the separation of concerns between recognizers, mediators, and the application.

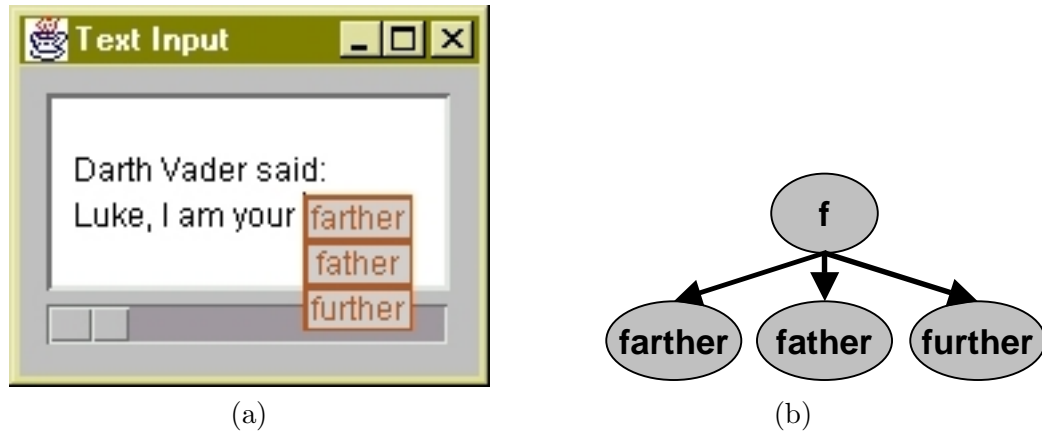
**Producer:** By storing the producer of an event, we are able to inform that producer (a recognizer) automatically that an event was accepted or rejected by the user, thus facilitating learning. In fact, we support a variety of recognizer features, described in more detail in Section 3.5.2. In addition, when necessary, the application may communicate with the recognizer that created an event directly. Although this sort of communication is discouraged, if an application designer wishes to design application-specific recognizers and mediators it is possible.

**User Data:** *user\_data* is an unassigned slot for additional data. This provides a mechanism for allowing recognizers, mediators, and the application to communicate more directly with each other, and is generally not used unless an application designer is building intensely application-specific recognizers and mediators.

**Feedback:** The *feedback* slot of each event contains a feedback object that has some way of displaying the event. Each feedback object knows how to generate a textual representation of an event, and may also support something more sophisticated. Thus, a mediator need know nothing about an event to display it or to display information about it. For example, our default *n*-best list mediator simply displays the textual representation of each leaf node of an event hierarchy in a menu format.

Continuing our example, the producer of “farther,” “father” and “further” would be the word predictor, and `user_data` would generally be empty. Both events would inherit the default feedback object provided for any event of type characters: a class that knows how to display text on screen or return a text (string) representation of its associated event.

The next subsection gives more details on the event hierarchy and how it is managed and updated by the toolkit architecture. Our description of the modifications to the event



**Figure 3-1:** An example of recognition of user input. (a) The user has typed the character ‘f’. A word predictor has interpreted it as the word ‘farther’, ‘father’, or perhaps ‘further’. (b) An ambiguous event hierarchy associated with this example.

---

object concludes with a description of how the event hierarchy is able to model the different types of errors and ambiguity highlighted in the introductory chapter of this thesis.

### 3.2.2 The event hierarchy

Figure 3-1 illustrates the example we have been using throughout this chapter. In (a), the user has typed an ‘f’. This character was recognized by a word predictor as the word ‘farther’, ‘father’, or perhaps ‘further’. The ambiguous event hierarchy shown in part (b) of the figure represents each of these interpretations. This ambiguous event hierarchy, a directed graph representing how sensed data is used, or recognized, is an extension of previous work in (unambiguous) hierarchical events by Myers and Kosbie [61]. Myers and Kosbie were interested in representing how events were used (if a series of mouse events resulted in the user saving a document to disk, that action would be represented as an event in their hierarchy). This is useful for supporting undo and programming by demonstration. We also use a hierarchy to represent how events were used, but we consider interpretation by a recognizer to be a “use” of an event. Note that an event may have any number of parent



**Source Code 3-1:** The algorithm used to determine whether an event hierarchy is ambiguous (`is_ambiguous()`)

```

boolean is_ambiguous() {
    // There are conflicting siblings
    if ( conflicting_siblings () not empty)
        return true;
    // There is at least one ambiguous interp.
    for each interpretation
        if interpretation .is_ambiguous()
            return true;
    // An open event is by definition ambiguous
    return this is open;
}

```

**Source Code 3-2:** The algorithm used to determine whether an event has siblings that are ambiguous (`conflicting_siblings ()`)

```

Set conflicting_siblings () {
    Set conflicts ;
    for each sibling {
        if (( sibling is not related to this) &&
            ( sibling is open))
            conflicts .add(sibling);
    }
    return conflicts;
}

```

---

events of which it is an interpretation. For example, if the user had typed two characters (say, ‘f’ and ‘a’), the word predictor might have only suggested “farther” and “father” and both of those interpretations would have had two sources (the ‘f’ and the ‘a’).

By modeling recognition and ambiguity in this way, we can support certain recognizer independent methods for managing ambiguity. First of all, we can normally identify the presence of ambiguity simply by examining the event hierarchy. When two events that are *open* (neither *accepted* nor *rejected*) share the same parent (are siblings), they are considered to be in conflict (Source Code 3-2), and the event hierarchy they are part of is considered to be ambiguous<sup>1</sup>. The algorithm for identifying ambiguity is given in Source Code 3-1.

When an event is accepted or rejected, its status is changed from *open* to *closed*. At that point, we guarantee that the status of that event will no longer be changed. Until that point, anything which received an event may show some feedback about that event, but actions that involve modifying application data structures should only be taken if and when an event is accepted.

As individual events are accepted or rejected, there are certain modifications that must

---

<sup>1</sup>It can be useful to circumvent this in some cases by allowing sibling events to be tagged as “related” when they should not be in conflict. For example, if two boxes are drawn next to each other, they might both share a line. Thus that line has two valid sibling interpretations that are not in conflict.

<p><b>Source Code 3-3:</b> The algorithm to accept events (<code>accept()</code>)</p> <pre> // Accept events, propagating up hierarchy accept() {   // Accept sources   for each source in sources     source.accept();   // Reject conflicting siblings   for each sibling in conflicting_siblings ()     sibling.reject ();   closure = accepted;   // Notify recognizer that produced this event   producer().notify_of_accept (<b>this</b>); } </pre>	<p><b>Source Code 3-4:</b> The algorithm to reject events (<code>reject()</code>)</p> <pre> 1 // Reject events, propagating down hierarchy 2 reject () { 3   // Reject interpretations 4   for each interpretation 5     interpretation.reject (); 6 7   closure = rejected; 9 // Notify recognizer that produced this event 10 producer().notify_of_reject (<b>this</b>); 12 } </pre>
---	--

---

be made to the event graph. When an event is rejected, its interpretations are also rejected. The logic behind this is that if an event is wrong, that is to say, did not really happen, then interpretations of it also cannot be correct. When an event is accepted, its source events are also accepted. This guarantees that there is a logical, correct path by which this event was derived: Its source can no longer be rejected, because that would force us to reject it as well. In addition, any events (*e.g.* siblings) that conflict with the accepted event are rejected. Source Code 3-2 shows how conflicting events are identified. Source Code 3-3 and 3-4 show the algorithms for `accept()` and `reject()`.

### 3.2.3 Representing different types of errors and ambiguity

In the introductory chapter to this dissertation, we discussed the standard types of errors (Section 1.1.2, page 4) and ambiguity (Section 1.1.3, page 6), normally found in recognition-based interfaces. They are: rejection errors; substitution errors; insertion errors; recognition ambiguity; segmentation ambiguity; and target ambiguity. Here we will explain how each of them may be modeled in our system.

**Rejection Errors** are difficult to model because they arise when the system does not know that there is input to be recognized. However, it would be possible to create a special “null” event when a recognizer could not come up with any valid interpretations. This “null” event could also be useful when the recognizer does create interpretations. In this case, it represents the chance that none of the interpretations created by the recognizer are correct.

**Substitution Errors** occur when the user’s input is misrecognized, and that is exactly what this architecture is designed to handle. Rather than selecting the wrong choice up front, all of the possible choices can be represented in the event hierarchy and the user can be asked by a mediator which is correct.

**Insertion Errors** are really a special case of substitution errors in which no answer is correct. It is up to the mediator to allow the user to reject all suggested interpretations.

**Recognition Ambiguity** is a form of ambiguity associated with insertion and substitution errors. Recognition ambiguity results when multiple interpretations share the same sources.

**Segmentation Ambiguity** results when multiple interpretations have non-identical, overlapping source sets.

**Target Ambiguity** is an additional level of interpretation in the event hierarchy. For example, if the user’s input has been interpreted, as an “open” command, the system must identify what should be opened. The possible choices would generally be represented in the event hierarchy as interpretations of the “open” event. At that point, target ambiguity becomes indistinguishable from recognition ambiguity. Details on how target ambiguity is generated have to do with the way events are dispatched, and are described in more detail in the Section 3.3.

In conclusion, the ambiguous event hierarchy is powerful enough to handle all six types of ambiguity and errors found in the literature. In the next two sections, we will go on to describe the method by which events are dispatched and ambiguity is generated as well as the mediation subsystem. In combination, this section and the next two sections describe the most important requirements of an architecture that wishes to support ambiguity and recognition.

### 3.3 Event Dispatch

The modified events described above play the essential role of a recognizer independent description of the user's input and how it is being interpreted. It is the responsibility of the event dispatch portion of the toolkit architecture to manage the creation of the event hierarchy, and to decide when any given application component, recognizer, or mediator, will receive an event in the hierarchy. Many toolkit architectures dynamically gather a series of eligible objects that may consume (use) each event, an idea that was proposed early on in GUI Toolkit research [23]. The toolkit architecture will normally dispatch (deliver) an event to each such eligible recognizer in turn until the event is consumed. After that, the event is not dispatched to any additional recognizers because of an underlying assumption that this would cause conflicts and confusion. Variations on this basic scheme show up in a wide range of user interface toolkits [35, 60, 83, 72]. A slightly different scheme, in which every eligible object receives each event, shows up in most non-GUI toolkits (*e.g.* the Context Toolkit [20]). We based our dispatch algorithm on the second model because the first approach is a degenerate case of it: the conflicts inherent in the first model are a form of target ambiguity that can be modeled in our system (see the previous Section). In practice, a toolkit may support both policies, as OOPS does (Chapter 4).

In its simplest form, the event dispatch system has two steps, starting on lines 5 and 8 in Source Code 3-5. First, on line 5, an event is dispatched to all interested recognizers

**Source Code 3-5:** The basic event dispatch algorithm (`dispatch_event()`)

```
// Dispatch a sensed event, building up the
// associated hierarchy
public void dispatch_event(event e) {
    // Dispatch e to all recognizers, recursively
    generate_interps(e)
    // Resolve ambiguity. (See Source Code 3-7).
    if e.is_ambiguous() {
        mediation_subsystem.
            resolve_ambiguity(e, null);
    }
}
```

**Source Code 3-6:** A helper algorithm for event dispatch (`generate_interps()`)

```
1 // Dispatch an event to recognizers, recursively.
2 // Returns the newly generated interpretations.
3 public Set generate_interps(event e) {
4     Set local_new_interps;
5     Set new_interps;
6     for each recognizer
7         if (recognizer.event_is_useful(e)) {
8             recognizer.use(e);
9             local_new_interps.add(
10                e.new_interpretations());
11        }
12    for each interpretation in
13        local_new_interps
14        new_interps.add(
15            generate_interps(interpretation));
16    new_interps.addall(local_new_interps);
17    return new_interps;
18 }
```

---

(Source Code 3-6). This is done recursively—all interpretations that are created are also dispatched. The toolkit architecture may support a variety of policies for deciding which recognizers are interested in input. For example, many existing toolkits may dispatch events selectively based on their position or type [35, 22, 60]. Second, on line 8, the hierarchy is mediated (if it is ambiguous).

Consider again our example of a word predictor (a recognizer that receives input). When it receives a character (such as ‘f’), it creates several interpretations of that character, one for each word that character might be the start of (such as “farther”, “father” and “further”). First the character is passed on to any other interested party. Then each of the interpretations are dispatched. In this case, there are no other interested recognizers. Since the resulting hierarchy is ambiguous, it is passed on to the mediation subsystem, the second step of the event dispatch system, described next.

## 3.4 Mediation Subsystem

As mentioned in the previous chapter, a recognition error is defined by the user's intent, and neither the recognizer nor the toolkit necessarily know what the correct interpretation is. It is through mediation that this is determined, often with the help of end users. Until mediation is completed, information about all known possible interpretations is stored using the ambiguous event hierarchies described above.

Once all interpretation of a given piece of raw input is complete (after Source Code 3-6, called on Line 5 of Source Code 3-5), the architecture determines if mediation is necessary by examining the hierarchy (using the `is_ambiguous()` method, Source Code 3-1, see line 7 of Source Code 3-5). If there is ambiguity, the hierarchy is sent to the mediation subsystem (line 8 of Source Code 3-5), which passes the ambiguous event hierarchy through a series of meta-mediators and mediators until all ambiguity is resolved (See Section 3.4.1 and Source Code 3-7 for further details). A meta-mediator encodes a policy for selecting which mediator should resolve ambiguity when. A mediator is simply something that accepts or rejects events in a hierarchy.

When a node is accepted, all of its parents are automatically accepted (since there would be no valid path to the accepted node if the parents were rejected). See Source Code 3-3 for this algorithm. This means that once a leaf node of the event hierarchy is accepted, all of the other nodes in the hierarchy are uniquely determined. The implication is that a re-usable mediator can be created that displays leaf nodes to the user. An  $n$ -best list might display  $n$  of the leaf nodes in a given hierarchy. Since semantic relevance often increases as additional layers of recognition are added to the hierarchy, this is a reasonably safe strategy. Once a leaf node is chosen by the user, and accepted by the mediator, mediation is completed because the event hierarchy is no longer ambiguous. This  $n$ -best list is re-usable because it does not depend on any specific recognizer, but instead makes use of our consistent, recognizer independent internal model of ambiguity (the ambiguous event hierarchy).

If the application designer were to use that  $n$ -best list with the hierarchy in the example used throughout this chapter, it would display the leaf nodes of the hierarchy, “farther” “father” and “further” by retrieving their feedback objects and requesting textual representations of them. When the user clicks inside one of the words, such as “farther,” the mediator would accept that word. The `accept()` algorithm would cause both siblings (“father” and “further”) to be rejected, and the source (‘f’) to be accepted. At this point, the hierarchy would no longer be ambiguous and mediation would be finished. Notice that the mediator never had to check who or what produced the events, or even what type the events were.

The previous example illustrates how a mediator functions. The mediation subsystem includes both mediators, which resolve ambiguity, and the meta-mediators, which represent different strategies for deciding at what point each mediator should be given a chance to resolve ambiguity. Additionally, mediation requires a dispatch algorithm for sending the ambiguous event hierarchy to different meta-mediators. We will first describe this dispatch algorithm, followed by details on how mediators and meta-mediators work.

### 3.4.1 The mediation dispatch algorithm

The mediation dispatch algorithm ties mediators and meta-mediators into the architecture. Remember that the event dispatch system is responsible for identifying the presence of ambiguity, and passing an ambiguous hierarchy to the mediation subsystem (Source Code 3-5). At this point the mediation dispatch algorithm begins its work. It is the mechanism through which a mediator eventually receives an ambiguous hierarchy. Simply put, the mediation subsystem passes a pointer to the root of the hierarchy to a meta-mediator, which passes it to a mediator. The dispatch algorithm in the mediation subsystem encodes the decision of the order in which meta-mediators handle an ambiguous event hierarchy. It also handles the asynchronicity caused when a mediator needs to defer mediation in order to wait for user input. The reason and mechanisms for deferring mediation will be addressed

**Source Code 3-7:** The algorithm called by the event dispatch system to send an ambiguous event hierarchy to meta-mediators for resolution (`resolve_ambiguity()`)

```

// in mediation subsystem
resolve_ambiguity(event root) {
    if (not root.is_ambiguous()) return;
    // select each meta-mediator in the
    // queue in turn
    for each meta-mediator
        switch(meta-mediator.
            meta_mediate(root))
        case RESOLVE:
            if (not root.is_ambiguous())
                return;
        case PASS:
            continue;
        case DEFER
            cache meta-mediator with root
            return;
}

```

**Source Code 3-8:** The algorithm used when a mediator that has paused mediation wishes to pass control back to the mediation subsystem (`continue_mediation()`)

```

1 // in mediation subsystem
2 continue_mediation(event root,
3     meta-mediator mm) {
4     if (not root.is_ambiguous()) return;
5     // pick up where we left off - go to
6     // the next meta-mediator in the queue
7     // just as if m had returned immediately
8     for each meta-mediator after mm
9         switch(meta-mediator.
10             meta_mediate(root))
11         case RESOLVE:
12             if (not root.is_ambiguous())
13                 return;
14         case PASS:
15             continue;
16         case DEFER
17             cache meta-mediator with root
18             return;
19 }

```

---

in more detail when we describe how mediators work. Since event dispatch is synchronous, and mediation may take time (for example, while the end user selects an item in a menu), deferring provides a way to switch to an asynchronous mechanism so that the system can still respond to new input from the user.

Source Code 3-7 shows how the mediation subsystem passes an ambiguous hierarchy to each meta-mediator in turn in the synchronous case, while Source Code 3-8 shows the algorithm used in the asynchronous case. The main difference between these methods is that, in the asynchronous case, things continue where they left off (with the next mediator in line), rather than starting from scratch. Meta-mediators are kept in a queue. The programmer controls the order in which meta-mediators are installed, and can reorder them dynamically at runtime as needed.



As an illustration, consider the hierarchy in our running example with the word predictor. Suppose that the application designer had set up a single meta-mediator containing a single mediator, an *n*-best list like the one shown in Figure 3-1. The mediation dispatch algorithm (Source Code 3-7) would pass the hierarchy to the meta-mediation policy, which would in turn pass it on to the mediator. This mediator would display a menu of the choices, and then *defer* because it needs to wait for user input. At this point, the meta-mediator would exit, as would the dispatch algorithm (Source Code 3-7, line 13). This is important because, as stated, the entire dispatch process is synchronous, and no other events would be able to be dispatched until ambiguity is either resolved or deferred.

Now that further events can be dispatched, it is possible that they may cause a recognizer to add new interpretations to the hierarchy while mediation is deferred. In this case, the deferring mediator will be notified about these changes and given a chance to mediate the new hierarchy. Once the user clicks on a choice, the mediator tells its meta-mediator to call `continue_mediation()`, the algorithm shown in Source Code 3-8. This causes mediation to pick up exactly where it left off. In this example, the event hierarchy is no longer ambiguous and mediation is complete.

The next two sections will give more details on exactly how a mediator defers mediation, and how a mediator causes `continue_mediation()` to be called when necessary (this is done in the `wait_completed()` method in the example mediator of Source Code 3-9). They describe the other possible return values for mediators and meta-mediators, and give details on how a meta-mediator selects a mediator and passes control to it. Additionally, they describe how a mediator actually mediates an event hierarchy.

### 3.4.2 Mediation

A mediator is an object that accepts or rejects events in an ambiguous event hierarchy. Mediators fall into two major categories—automatic and interactive. *Automatic* mediators use various algorithms to decide which interpretation to accept on their own. *Interactive*

**Source Code 3-9:** The algorithm used by a mediator to mediate an event hierarchy (`mediate()`)

```
// in a mediator
mediate(event root, Set relevant_events) {
    if some event in root is correct
        event.accept()
    if some event in root is wrong
        event.reject ()
    if this is done and an event was accepted or rejected
        return RESOLVE
    if this is done and no changes were made
        return PASS
    if this is not done and we have to wait for some reason
        set up wait condition
        cache root
        return DEFER
}
// the wait condition has happened
wait_completed(event root) {
    // mm is the meta-mediator in which this is installed
    mm.continue_mediation(root, this)
}
```

---

mediators (such as *choice* and *repetition*) wait for the user to specify the correct interpretation. An interactive mediator may display information about the ambiguity to inform the user about possible options. An *n*-best list is an example of an interactive mediator in a GUI setting.

A hierarchy that needs to be resolved is passed to a mediator in a call to `mediate()` by the meta-mediation system (described in Section 3.4.3). Source Code 3-9 illustrates how a generic mediator might implement this method. A mediator has the option to `PASS`, `RESOLVE`, or `DEFER` mediation on an ambiguous event hierarchy.

A mediator may `PASS` if it cannot mediate the hierarchy. For example, a mediator that is only intended to handle mouse events would pass if it were to receive text events. In this case, the hierarchy will be given to the next mediator in line by the meta-mediation system.

Automatic mediators may `RESOLVE` a portion of the hierarchy by accepting and rejecting events. For example, an automatic mediator could be used to eliminate any words that are not nouns from the possible choices based on the information that the user is filling a form

**Table 3-3:** Constants and methods in the *mediation* class

<b>PASS</b>	The return value used by mediators that do not wish to mediate a certain event.
<b>RESOLVE</b>	The return value used by mediators that have mediated a certain event.
<b>DEFER</b>	The return value used by mediators that need more time to mediate a certain event.
<b>int mediate(event root, set relevant_events)</b>	Mediate an event hierarchy. See Source Code 3-9.
<b>boolean cancel_mediation_p(event root)</b>	Check if this is willing to cancel a deferred mediation on an event hierarchy so that it can begin anew. Return false if not.
<b>update(event root, Set new_events)</b>	Update this to reflect the addition of new events to an event hierarchy that it has deferred on.

---

that requires a name.

A mediator may also DEFER mediation. This may happen for two reasons. First, it may need additional information (say further input events) to make a decision. An interactive mediator always defers for this reason since it must wait for user input to make a decision. Second, it may wish to preserve ambiguity for some period of time. For example, an automatic mediator might defer mediation until it (automatically) determines that the user can be interrupted, at which point control is passed to an interactive mediator. In both cases, the mediator must call `continue_mediation()` once the deferment is not necessary, at which point things continue exactly as they would have, had the mediator returned `resolve` originally.

In addition to the `mediate()` method, mediators support `cancel_mediation_p()` and `update()`. The full complement of methods and constants supported by mediators is shown in Table 3-3. Both `cancel_mediation_p()` and `update()` deal with situations in which an event hierarchy on which a mediator has deferred, is modified. More details on how this situation is detected and handled are given in Section 3.5.2.1.

**Table 3-4:** Constants and methods in the *meta-mediation* class

**PASS** The return value used by meta-mediators that do not wish to mediate a certain event hierarchy

**RESOLVE** The return value used by meta-mediators that have mediated some or all of an event hierarchy.

**DEFER** The return value used by meta-mediators that need more time to mediate a certain event hierarchy

**int meta\_mediate(event root)** Choose a mediator and call its `mediate()` method. See Source Code 3-10. See Section 4.4.1 for more specific examples.

**handle\_modification(event root, Set new\_events)** Retrieve the mediator that deferred on root and notify it of the change. See Source Code 3-13.

**continue\_mediation(event root, mediator m)** Pick up mediation where it stopped when **m** deferred (starting with the next mediator in line after **m**). See Source Code 3-11.

**add(mediator m)** Add a new mediator to the meta-mediator.

**remove(mediator m)** Remove mediator **m**.

---

### 3.4.3 Meta-mediation

Meta-mediation handles decisions regarding when to use which mediator. For example, some mediators may only be interested in events that occurred in a certain location, while others may be only interested in spoken input, and so on. The application designer encodes decisions about how and when mediation should occur through the meta-mediation system.

Table 3-4 has a complete list of the methods and constants for the meta-mediation interface. The `meta_mediate()` method passes an event hierarchy (represented by its root, a sensed event) to its mediators according to the details of the specific meta-mediation policy. Each meta-mediation policy stores a set of mediators and encodes some policy for choosing between them. Source Code 3-10 illustrates how a generic meta-mediator might implement this method. For example, a “queue” meta-mediator might keep its mediators in a queue and pass the event hierarchy to each of them in order. A “filter” meta-mediator,

on the other hand, might pass a hierarchy to a given mediator only if the hierarchy meets certain requirements set out by a filter associated with that mediator. A more radical “simultaneous” meta-mediator might allow several mediators to resolve the same hierarchy concurrently. In fact, the application designer could create as complex a meta-mediator as she wished. One of the few examples of a meta-mediation policy found in the literature is Horvitz’s work in using decision theory to pick an appropriate mediator based on a model of the user’s interruptability [32]. Detailed descriptions of several meta-mediation policies included by default with the toolkit based on this architecture (OOPS) are provided in the next chapter.

In summary, only one meta-mediation policy is active at a time, and the mediation subsystem goes through each policy in order until ambiguity is resolved (Source Code 3-7 and 3-8). As shown in Source Code 3-10, the active meta-mediator repeatedly selects mediators and calls `mediate()` (Source Code 3-9) until all ambiguity is resolved, or it has no mediators left to select from. If a mediator defers, the meta-mediator caches the mediator and the hierarchy it is deferring, and then exits (lines 11–13 of Source Code 3-10). This causes the mediation subsystem to cache the meta-mediation policy and then exit (lines 14–16 of Source Code 3-7. When a mediator is done deferring mediation, it passes control back to the meta-mediator that called it so mediation can continue where it left off, by calling `continue_mediation()` (Source Code 3-11). That meta-mediator eventually informs the mediation subsystem of this fact (Source Code 3-8).

#### 3.4.4 Example

Expanding on our previous illustration, suppose that the application developer wants to install three different mediators. The first is an automatic mediator that rejects any interpretations of mouse events. This is a way of ensuring that the only ambiguity about user input is that generated by the word predictor. The second is an  $n$ -best list like the one shown in Figure 3-1. The last is an automatic mediator that always accepts the top choice.

**Source Code 3-10:** The algorithm used by meta-mediators to send event hierarchies to mediators for resolution (meta\_mediate())

```

// in a meta-mediator
meta_mediate(event root) {
    // selected mediators in order according
    // to this meta-mediator's policy
    for each mediator
        switch(mediator.mediate(root, null))
        case RESOLVE:
            if (not root.is_ambiguous())
                return RESOLVE;
        case PASS:
            continue;
        case DEFER
            cache mediator with root;
            return DEFER;
    }
    return PASS;
}

```

**Source Code 3-11:** The algorithm called by a mediator to continue mediation that has been deferred (continue\_mediation())

```

1 // in meta-mediator
2 // pick up mediation where it stopped
3 continue_mediation(event root, mediator m) {
4     if (not root.is_ambiguous()) return;
5     for each mediator after m based on policy
6         switch(mediator.mediate(root, null))
7             case RESOLVE:
8                 if (not root.is_ambiguous())
9                     // tell the mediation subsystem
10                    // to pick up from here
11                    // (See Source Code 3-8)
12                    mediation_subsystem.
13                    continue_mediation(root, this);
14                    return;
15             case PASS:
16                 continue;
17             case DEFER
18                 cache mediator with root;
19                 return;
20         }
21     // tell the mediation subsystem to pick
22     // up from here (See Source Code 3-8)
23     mediation_subsystem.
24     continue_mediation(root, this);
25 }

```

```

dispatch_event('f')
  word_predictor.use('f')
  "farther" "father" "further" <--
  'f'.is_ambiguous()
  true <--
  resolve_ambiguity('f')
    queue.meta_mediate('f')
      nomouse.mediate('f')
        PASS <--
        nbest.mediate('f')
          DEFER <--
          DEFER <--
    ...
  queue.continue_mediation('f')
  'f'.is_ambiguous()
  false <--

```

**Figure 3-2:** An example trace of the methods called by `dispatch_event()`

---

This mediator is a backup in case any ambiguity arises that is not resolved by the first two mediators. Order is important here: for example, if the third mediator ever received a hierarchy before the other two, there would be no ambiguity left to resolve. A simple queue-based meta-mediation policy will suffice to enforce this ordering. Figure 3-2 shows a trace of the methods that might be called in this scenario.

Once the application is running, and the user enters a letter, the event dispatch algorithm (Source Code 3-5) passes that letter to the word predictor to be recognized. Since the word predictor generates ambiguous alternatives, `resolve_ambiguity()` is called (Source Code 3-7), and that in turn calls `meta_mediate()` (Source Code 3-10) on the first installed meta-mediation policy (the application developer’s queue meta-mediator). This, in turn, calls `mediate()` (Source Code 3-9) on the first mediator. Since the root of the hierarchy is an ‘f’, not a mouse event, the mediator returns PASS. The meta-mediation policy then gives the hierarchy to the interactive *n*-best list. This mediator displays a menu of the choices, and returns DEFER. At this point, the meta-mediator also exits, as does the dispatch algorithm. This is important because the entire dispatch process is synchronous, and no other events would be able to be dispatched until ambiguity is either resolved or deferred. Now the user

clicks on a choice, and the mediator calls `meta-mediator.continue_mediation()`, which sees that the hierarchy is no longer ambiguous, at which point mediation is complete.

## 3.5 Additional Issues

So far in this chapter, we have described the most essential pieces of an architecture that provides support for ambiguity in recognition. However, a complete architecture must also deal with other issues, and they are described in this section. First and foremost is the problem of providing backwards compatibility with applications and interactors not designed to deal with ambiguity. Next is an application programming interface for recognizers that provides recognizer-independent mechanisms for communication between recognizers, mediators, and the application. A third issue that arises because of the generality of our dispatch algorithm is how to avoid the potential for an endless loop in which a recognizer repeatedly interprets its own input. This section concludes with a discussion of some additional problems that are left for future work.

### 3.5.1 Modifying dispatch to provide backward compatibility

The event dispatch algorithm presented thus far (See Source Code 3-5) assumes that all application components and interface components are “recognizers” in the sense that: **(a)** if a component makes use of an event, it creates an interpretation of the event representing what it did, to be added to the ambiguous event hierarchy; and **(b)** if a component modifies the application or interface in any way, it is able to undo anything it did with an event that at some later time is rejected. For example, an interactor may choose to show some feedback about the event as soon as it is received, but only act on it (modifying application data structures) after receiving notification that it has been accepted.

However, in both of our implementations of this architecture, we modified an existing toolkit that had widgets and application components that did not have these properties



(we will refer to these as ambiguity-unaware components). Rather than rewriting every one of these components (with a resulting lack of backward-compatibility), we developed two possible approaches to solving this problem, based on two reasonable guarantees that may be made.

The first approach essentially makes no guarantees: It simply passes the entire hierarchy as one event to an ambiguity-unaware component, and lets that component decide if and when to use any portion of it. This approach is necessary in a distributed setting in which multiple users and applications may be using sensed events. With this approach, if an application wants information, it does not have to wait for mediation to occur to use it.

The second approach guarantees that if an ambiguity-unaware component receives an event, and uses it, that event will never be rejected. This approach is very effective, for example, in a single user setting such as a typical GUI. Source Code 3-12 shows the necessary modifications to the event dispatch algorithm presented in Source Code 3-5 to support this approach. The resulting algorithm allows ambiguity-unaware components to either receive an event before it becomes ambiguous, or after any ambiguity associated with it is resolved.

One key point in this algorithm is that an application developer has control over when a component, that is not aware of ambiguity, receives input. By adding a component to a special “wait” list, the application developer is telling the dispatch algorithm to only send input events to that component at the very end of the dispatch (line 20). This allows recognition to occur before an event is used by a component unaware of ambiguity (a situation which would make that event unavailable to recognizers because any new interpretations of the event would be in conflict with the event’s existing use). If a component waits until the end of the dispatch cycle to receive events, it will only receive accepted leaf nodes, because any node that is not a leaf node has already been used (interpreted) by some other component. Leaf nodes are simply the nodes of the hierarchy that do not have any open or accepted interpretations.

**Source Code 3-12:** The modified event dispatch algorithm for components unaware of ambiguity (`dispatch_event()`)

```
1 dispatch_event(event e) {
2     // Dispatch to components unaware of ambiguity. If any of them use the event, we are done.
3     for each ambiguity-unaware component not in "wait"
4         if (component.event_is_useful(e)) {
5             component.use(e);
6             return;
7         }
8
9     // If the event was not used, dispatch it to components aware of ambiguity (identically
10    // to Source Code 3-5)
11    generate_interps(e);
12    if e.is_ambiguous() {
13        mediation_subsystem.resolve_ambiguity(e);
14    }
15    complete_dispatch(e);
16 }
17 // separate out the final portion of dispatch so that when mediation is deferred, causing
18 // dispatch_event to exit, we can still complete the dispatch without redoing everything
19 complete_dispatch(event e) {
20     // Dispatch any unambiguous leaf nodes to components unaware of ambiguity
21     for each closed event ce in e.leaf_nodes()
22         for each ambiguity-unaware component in "wait"
23             if (component.event_is_useful(ce)) {
24                 component.use(ce);
25                 break;
26             }
27 }
28 }
```

Reconsidering our example from the simpler dispatch algorithm in Source Code 3-5, suppose that the application making use of the word predictor is using a text area that has no knowledge of ambiguity. This text component, like the word predictor, makes use of key presses. In our modified dispatch algorithm, the text component would make use of each key event, and the dispatch algorithm would immediately return. As a result, recognition would never occur. So if the user typed ‘f’, it would simply appear in the text area just as if there were no recognizer present. Alternatively, the text component could be added to the list of ambiguity unaware components waiting for dispatch to complete. In this case, the word predictor would interpret the ‘f’ as one or more words (such as “farther”, “father” and “further”). The mediation subsystem would be responsible for selecting which one was correct (presumably with the users help). The final selected word, a leaf node, would be dispatched to the text area, which would display it on screen just as it would have displayed the original key press. For example, if the user selects “father”, that word will be sent to the text area and appear on screen.

Note that event dispatch is actually split into two methods in Source Code 3-12. The portion that happens after mediation is separated out, because the dispatch algorithm exits when a mediator defers mediation. Dispatch is “restarted” by a call to `continue_mediation()` (shown in Source Code 3-8). A slight modification to the `continue_mediation()` method required by the modified dispatch algorithm is that it calls `complete_dispatch()` instead of simply returning when ambiguity is resolved. `complete_dispatch()`, the second method in Source Code 3-12, passes leaf nodes to each component in the “wait” list.

The dispatch system may be further modified to provide additional support for components that are not recognizers by recording information about how each ambiguity-unaware component uses an event. An implementation of this (Source Code A-11), along with all of the other major algorithms, can be found in Appendix A. This information is recorded as an interpretation, which can then be dispatched recursively just like recognized interpretations. One reason for doing this might be to allow a recognizer to interpret an end user’s

actions. For example, if the user repeatedly clicks on a certain check box in a dialogue, a recognizer might check it automatically whenever that dialogue comes up. This usage is very similar to the original inspiration for hierarchical events [61], and could in principle support other interesting features such as programming by demonstration and undo.

### 3.5.2 Extending recognizers

Recognition is generally viewed as an isolated task by the people making use of recognizers. However, recognition is often an ongoing process in which new information is combined with old information to revise the current view of the world. An example of a type of recognizer that depends on this constant flow of information is a decision theoretic Bayesian network [32]. Although third-party recognizers can be used unchanged under this architecture, there are a set of modifications that we feel would greatly increase their effectiveness. In essence, these changes represent a set of channels for communication between recognizers and mediators. First of all, recognizers should return multiple, ambiguous interpretations when detectable ambiguity exists. Second, it is helpful for a recognizer to separate segmentation and recognition information to create an event hierarchy that reflects different types of ambiguity. Choices made during the segmentation phase of handwriting recognition, for example, have a direct consequence on which alternatives are generated.

In addition to these changes, there are several additional features that are supported by specific mechanisms in this architecture. First, when there is early information about potential alternatives, a recognizer should expose this information to the architecture so that applications or mediators may make use of it. Toolkit support for delayed recognition can allow recognizers to produce incremental results. Second, although recognizers are generally created separately from the application in which they are used, there are times when mediators and recognizers, or applications and recognizers, must communicate. We describe how the application or mediator may directly request new alternatives from a

**Source Code 3-13:** The algorithm called by event dispatch when a modification is made to a deferred event hierarchy (`handle_modification`)

```
1 // in each meta-mediator
2 handle_modification(event root, Set new_events) {
3     // retrieve the correct mediator from the cache of deferred mediators
4     mediator m = cache.retrieve root
5     if cancel_mediation_p(root)
6         // start over, from scratch
7         mediation_subsystem.resolve_ambiguity(root);
8     else
9         // update m
10        m.update(root, new_events);
11 }
```

---

recognizer using rerecognition. Finally, we support specification of both the domain of input and range of output for a recognizer.

### 3.5.2.1 Delayed recognition

Most interpretations are generated during an initial input cycle before the start of mediation. When a new interpretation is created after mediation has begun, the event hierarchy must be updated. At this point, the new event may need to be mediated. If it is added to an unambiguous (or already mediated) hierarchy, the updated hierarchy is treated just like a newly created hierarchy. However, if the updated hierarchy is currently being mediated, the active mediator must be informed of the change. The architecture handles this situation by asking the current mediator to either cancel mediation or update its state. If the mediator chooses to cancel, the hierarchy is re-mediated. Any events that have previously been accepted or rejected remain that way.

When the event dispatch system detects a modification to a deferred hierarchy, it notifies the meta-mediation system that this has happened by calling `handle_modification()` (Source Code 3-13). This method retrieves the mediator currently mediating the modified event hierarchy, and asks if it is willing to cancel mediation (line 5). If not, it updates that mediator (line 10). Otherwise, it begins mediating the modified event hierarchy from the beginning (line 7).

In most cases, an automatic mediator will return `true` when `cancel_mediation_p(root)` is called. However, an interactive mediator may return `false` rather than interrupting mediation by disappearing. For example, an *n*-best list might check for new events in `root` and add them to the list it presents to the user.

The event dispatch system detects modifications by checking the root sources of each newly created event (the original sensed events). During any event dispatch, there is only one *current* root event. All other root sources are past events, and they can be checked against the cached list of deferred event hierarchies. An example of an event that has multiple root sources is a stroke (which consists of a mouse down, a series of mouse drags, and as mouse up, all of which are root sources). The algorithm in Source Code 3-5 is simplified for clarity and does not handle this problem. In its complete form, the following lines replace line 5.

```
Set news = generate_interps(e);
Associations roots;
for each interpretation in news
    if interpretation has multiple root sources
        for each root source
            if mediation_subsystem.cache contains root
                add interpretation to interps associated with root
for each root and interps pair in roots
    meta_mediator mm = mediation_subsystem.cache retrieve root
    mm.handle_modification(root, interps);
```

For example, suppose that the *n*-best list displaying “farther” “father” and “further” is visible, and the user types an ‘a’. At this point, the word predictor may update the hierarchy to include more words starting with ‘fa’ and downgrade the likelihood of “further.” When it adds these new interpretations, the event dispatch algorithm will check to see if the hierarchy rooted at ‘f’ currently being mediated. Since it is (by ‘nbest’), the dispatcher will call `handle_modification()`, which will call `nbest.cancel_mediation_p()`. The *n*-best list will return `false`, since it would be confusing to the user to have the menu simply disappear. At this point, the event dispatcher will call `nbest.update(event root, Set new_events)`,

```

dispatch_event('f')
  word_predictor.use('f')
  "farther" "father" "further" <--
  'f'.is_ambiguous()
  true <--
  resolve_ambiguity('f')
    queue.meta_mediate('f')
    nomouse.mediate('f')
    PASS <--
    nbest.mediate('f')
    DEFER <--
  DEFER <--
dispatch_event('a')
  word_predictor.use('a')
  (words starting with 'fa') <--
  handle_modification('f', words starting with 'fa')
    nbest.cancel_mediation_p('f')
    false <--
    nbest.update('f', words starting with 'fa')
...

```

**Figure 3-3:** An example trace of the methods called by `dispatch_event()`

---

informing the *n*-best list of the new possibilities, and the mediator will update itself to display those new choices. Figure 3-3 shows a trace of the methods that might be called in this scenario.

### 3.5.2.2 Guided rerecognition

Although it is not required by this architecture, there are times when recognizers, mediators, and the application may need to communicate. For example, mediators *know* when a recognition result is accepted or rejected, information a recognizer could use for training. This information is automatically passed from mediator to recognizer when an event is accepted or rejected (see Source Code 3-3 and 3-4).

Mediators may also know when the recognizer did not produce a correct answer among any of the interpretations it provided, and in this case they can give the recognizer a chance to try again. A mediator may communicate directly with recognizers to request new interpretations, using something we call guided rerecognition.

Guided rerecognition allows the recognizer to receive information that may be domain specific, and includes an event that should be rerecognized. The new information is intended to allow a recognizer to make a better guess as to how to interpret the user's input. Recognizers supporting guided rerecognition must implement the `rerecognize(event, Object)` method, where `event` is an event that the recognizer produced as an interpretation at some time in the past and `Object` may contain additional domain specific information.

Rather than interpreting a single event, some recognizers segment input into a series of events which are interpreted as a group. Recognizers that segment input may support the `resegment(Set, Object)` method, which tells them that a mediator has determined that the events in `Set` should be treated as one segment and interpreted. In the future, we also plan to add a method that allows a mediator to request that segmentation be redone without suggesting the solution.

### 3.5.2.3 Domain and range filters

In addition to knowing when the recognizer has not produced the correct result, a mediator (or the application) may have access to information ahead of time about which results are impossible or which input is not to be recognized. Domain (input) and range (output) filters that encode this knowledge may be added to a recognizer. Ideally, a recognizer should update its priorities to reflect the information provided by these filters, but even when the recognizer is provided by a third party and does not support filtering, these filters can still be useful.

For example, suppose that the word predictor is a third party mediator. The application designer may install a domain filter that filters out any numbers and the letter 'q'. As a result, the recognizer would never see those characters and never have an opportunity to interpret them. The application designer might also install a range filter that effectively kills any interpretations created when the user types a space. A range filter is chosen in this case because the space character provides important information to the word predictor



that the previous word has ended. However, the application designer may wish to make sure that the word predictor only starts making predictions after the user has typed at least the first character of a word, and thus will filter out any interpretations of the space.

### 3.5.3 Avoiding endless loops

Because recognition results are dispatched recursively just as if they were raw input, a recognizer may be given an event that it itself created. This leads to the possibility that an endless loop can occur. For example, suppose that the word predictor had received the letter ‘a’, which is also a word. It might create the (word) ‘a’ as an interpretation of ‘a’. ‘a’ would be dispatched, and since it is a character, once again sent to the word predictor, which would once again create the word ‘a’ as an interpretation, *ad infinitum*. Even more insidiously, suppose that the word predictor interpreted the letter ‘a’ as “alpha” which was then interpreted as the letter ‘a’ by a military spelling recognizer. When the word predictor receives the new ‘a’ it would again create “alpha” . . . . The input system guards against endless loops such as these by slightly limiting a recognizer’s powers: a recognizer may not create an interpretation if it has already created an identical interpretation in the past that is an ancestor of the new interpretation. Both of the examples given above would be forestalled by this algorithm.

### 3.5.4 Future issues

There are some issues that need to be explored in more detail in the design of the architecture we have described. One particularly interesting issue is how to support undo of events that have been accepted or rejected. If an accepted event is undone, do we rerecognize its source? Do we change its state, and the state of any conflicting siblings, to **open** instead of **rejected/accepted** and then re-mediate? Do we do both? What if the event has interpretations that have already been accepted/rejected?

In general, this raises the question of whether or not ambiguity should be preserved, and for how long. For example, one could imagine adding a `PRESERVE` option to the choices that each mediator has (and a corresponding `preserve()` method to the event class). When called, this would “resolve” a portion of an event hierarchy without eliminating ambiguity. There are tradeoffs here between efficiency and complexity, but the result is richness of functionality. The complexity comes in part from the task of support resource-intensive truth maintenance algorithms, which must monitor the hierarchy for paradoxes and remaining ambiguity [62].

Another area for further investigation is how to deal with mediation in a distributed architecture where there may be competition between multiple users and applications. This is discussed in more detail in Chapter 7, where we describe an initial implementation in just such a setting.

### 3.6 Instantiating the Architecture

Based on what we have described, a toolkit built around this architecture must include an internal model of hierarchical events, a library of mediators, support for recognizers of all sorts, for adding new recognizers, and for mediation. The result is a system that supports a separation of concerns. The ambiguous event hierarchy becomes a communication mechanism that allows recognizers, mediation, and the application to operate independently on the user’s actions.

This architecture was instantiated in two diverse settings, described in the next chapter and in Chapter 7 (OOPS and CT-OOPS, respectively). This is evidence that this architecture is a general solution to the problem of handling ambiguity.

## Chapter 4

### OOPS: MEDIATION IN THE GUI WORLD

The Option Oriented Pruning System (OOPS) [51] is a GUI instance of the architecture described in Chapter 3. It was built by extending an existing user interface toolkit that had no support for ambiguity or recognition, subArctic [35, 22]. We were particularly interested in applying our architecture in a GUI setting because we wanted to show how recognition results can be combined seamlessly with other input, even in the presence of ambiguity. OOPS includes the architectural solutions necessary to support recognition and mediation, as well as a reusable library of mediators. In particular, it supports separation of recognition, mediation, and the application, and it has a modified event dispatch system that treats recognition results as first class events to be dispatched and further interpreted just as standard keyboard and mouse input is dispatched. As with keyboard and mouse events, recognition results represent user input and should be routed to the user interface, rather than handled directly by the application. This is important because even when the input method changes (*e.g.*, from handwriting to speech), the application should not need to be rewritten.

In order to support these goals, we chose to have an extremely broad definition of recognition, one that includes anything that interprets input. This means that interface components (interactors) are recognizers, since they are essentially converting user input (*e.g.* a mouse click) to actions (*e.g.* the command executed when a certain button is pressed). These interactors can be used to build an interface just like standard interactors that are unaware of ambiguity. We will use the term recognizer throughout this chapter

to refer both to standard recognizers and any components or interactors that are aware of ambiguity. Internally, they are very similar: The main difference the feedback about events is separated from action on those events in recognizers (and ambiguity aware interactors), because those events may be ambiguous. Some key properties of recognizers (including ambiguity aware interactors) follow:

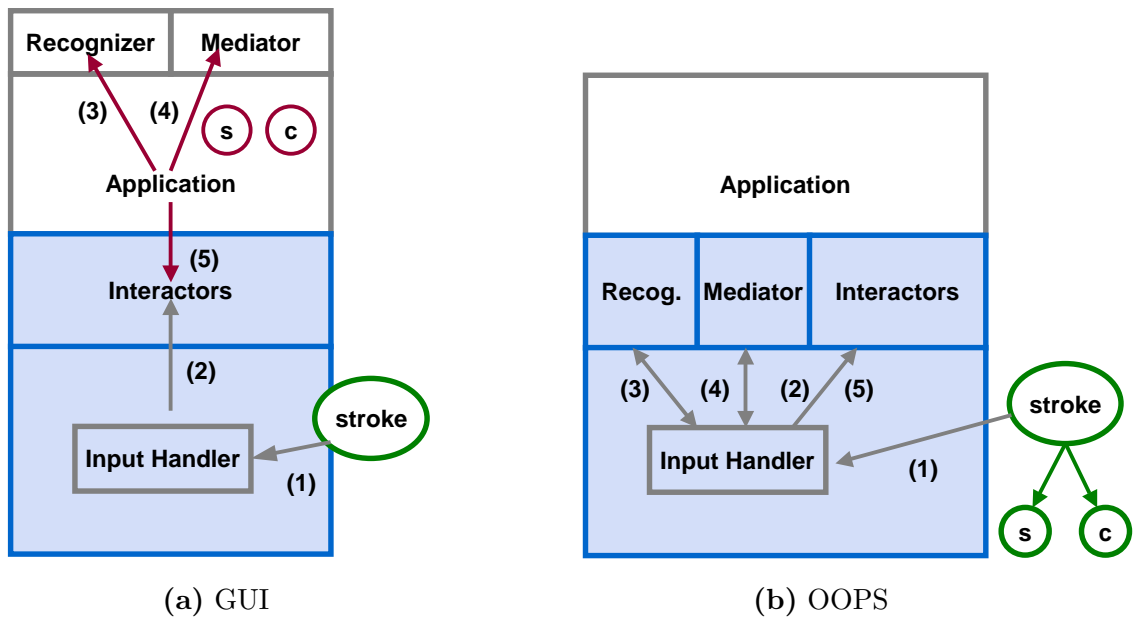
- When a recognizer gets an input event, it creates an interpretation of that event, saying how it would like to use it.
- A recognizer receives events that are ambiguous, rather than only receiving unambiguous events.
- If an interpretation is rejected, a recognizer must know how to undo any feedback it gave when it first received that event
- A recognizer may provide feedback about events while they are still ambiguous, however it only acts on an event in any non-reversible way, including informing the application about the event, once ambiguity has been resolved.

This chapter will continue to use the word predictor example to illustrate how an application developer might use various aspects of the OOPS toolkit. We start by introducing how a standard GUI handles input in the absence of recognition [31, 60] (Section 4.1). The next section describes the architecture of OOPS (Section 4.2). Section 4.3 discusses how OOPS is able to support ambiguity in interactors, and other issues related to backwards compatibility. We then describe the library of mediators provided with OOPS in Section 4.4. We conclude with a case study of how a developer would use the toolkit to write progressively more sophisticated versions of the word predictor application, starting from the simplest way of taking advantage of recognition and mediation up to application-specific mediators and use of ambiguity.

## 4.1 The Architecture of a Standard GUI Toolkit

We chose to use a 1990s era GUI toolkit as a starting place because so many of today's toolkits use the same basic model for input handling. Given access to the source code of a typical GUI toolkit, it should normally be possible to make the same changes we describe to support recognition, ambiguity, and mediation. A GUI toolkit is generally composed of a library and an infrastructure which embodies a particular architecture. Its library consists of a suite of interactors, including buttons, menus, and text areas. The infrastructure helps to handle issues such as how to display an interactor, when to redraw part of an interface, and how to route user input to the appropriate places. Most modern toolkits treat input as a series of discrete events. An application developer does not need to interact directly with the event dispatch system. Instead, an application receives callbacks or other notifications, from interactors. Figure 4-1(a) illustrates the flow of input through a typical GUI architecture, showing how an application developer would make use of a recognizer in that setting. The operating system delivers information about user actions to the GUI toolkit in the form of events (1). These events are parsed and passed on to the appropriate interactors drawn from the GUI toolkit's library by the event dispatch system (2). Anything involving recognition must be handled manually by the application.

Should an application built with such a toolkit need to make use of a recognizer, the application developer would have to handle all of the details herself, including calling the recognizer (3), checking for multiple ambiguous results, and passing them to a mediator (4). One of the most important and difficult steps would be delivering the recognition results back "down" the architecture to the appropriate interactors (5). For example, if a recognizer such as the word predictor produces text (something also done by a variety of other recognizers including handwriting, speech, and character recognizers), the application developer must determine which interactor in the interface has the current text focus (would receive text had it been typed instead of recognized), and also determine what methods



**Figure 4-1:** A comparison of how recognized input is handled in a standard GUI toolkit and OOPS. **(a)** A standard GUI architecture, which does not support recognition. The arrows originating outside the shaded box, labeled 3, 4 and 5, represent actions that must be handled by the application in this architecture but not in OOPS. **(b)** The OOPS architecture. OOPS' event handling system talks to interactors, and recognizers directly, and passes any ambiguous results to the mediation subsystem. The application does not have to deal with any recognizers or ambiguous input. Notice how the recognition results must be managed by the application in **(a)**, while they are part of an internal data structure in **(b)**.

can be used to insert text in the appropriate place in those interactors, in order to force the interface to reflect the recognition results. In essence, the application would need to reimplement parts of the toolkit infrastructure.

As with the GUI example, steps 1 and 2 are within the toolkit infrastructure of OOPS (blue box). Additionally, unlike the GUI example, everything relating to recognition is handled automatically by OOPS (3-5). This distinguishes OOPS from other toolkits that support recognized input: OOPS treats recognition results no differently from keyboard and mouse input *when they are of the same type*. Recognition results in other GUI toolkits that support recognition are not made available to the interface in the same way as “standard” input.

## 4.2 The Architecture of OOPS

### 4.2.1 OOPS input

OOPS supports separation of recognition, mediation, and the application, and it has a modified event dispatch system that handles the job of generating and resolving the ambiguous event hierarchy described in the previous chapter. Like Amulet and Artkit, two other GUI toolkits that provide support for recognized input integrated into their normal event dispatch system [45, 31], OOPS delivers recognized input through the same dispatch system as mouse and keyboard input. Unlike those toolkits, OOPS does not treat recognition results as a special type of event, but allows recognizers to produce any type of event. This means that if recognized input is of the same type as conventional input (text or mouse events), it can be used by anything that accepts input from those sources with no modifications. It does not matter how many recognizers and mediators are involved, if text or mouse events are the end result they will go to the same place as conventional mouse or keyboard input. Also, unlike those toolkits, OOPS supports ambiguity and ambiguity resolution.

In order to build OOPS, we modified the input-handling portion of an existing GUI toolkit (subArctic). The input system in OOPS looks very similar to that in any GUI toolkit, and in the simplest case, application development remains unchanged. Figure 4-1 (a) and (b) shows the differences between a standard GUI architecture and OOPS. The thick line around the majority of Figure 4-1 represents the boundaries of the architecture of the toolkit. Note that most of the changes between the two figures fall within this box. The three major modifications described in the previous chapter are all visible here. Mediators (and recognizers) are part of the library of the toolkit, and a mediation subsystem handles the job of calling mediators at appropriate times. Events are interpreted and stored in an ambiguous hierarchy (bottom right). It is this hierarchy that is passed around along the arrows. And finally, the event dispatch system itself is modified.

As an illustration, consider how an application developer would incorporate a word predictor into an application. Suppose the application developer wishes to build a simple application that allows the user to enter text in a text box. She would select a text entry interactor from the toolkit library, and add it to her user interface. If the user is typing the text, no more work would be required. However, if the developer wishes to add word prediction to this application, she would have to pass each new character that was typed to the word predictor manually. If the word predictor returned multiple choices, the application would have to pass them to some sort of mediator, and when the mediator informed the application that the user has selected a particular word, the application would have to call the specific method associated with the original text area that would add the word to the visible text. For example, many text areas require an application to call a method like `text_area.get_text()` and then concatenate the new text onto the old and call `text_area.set_text()`, passing in the combination of new and old text. The goal of this thesis work is to simplify the job of using recognition so that it requires no more effort to write an application that uses recognized input than it does to write an application that uses keyboard input. Both are simply input “devices”, and should be handled as such by



the toolkit. This means moving the job of calling the recognizer, mediating recognition results, and delivering them to interface components, down into the dispatch system of the toolkit instead of requiring the application to handle them.

As an added benefit, our modified approach to event dispatch allows some flexibility that might not show up in an ad-hoc, application-specific solution. For example, not only may a recognizer interpret a single source event in multiple ways, it may link the interpretations it creates to multiple source events if they are relevant, and this relationship will then be encoded in the ambiguous event hierarchy<sup>1</sup>. Another advantage of architectural support is that multiple recognizers (or interface elements) may interpret the same events without any additional burden to the application developer. This can result in *target ambiguity*, which like other forms of ambiguity, will be resolved by the mediation subsystem. See Section 3.2.3 for a discussion of how different types of ambiguity may arise in this architecture. A final benefit is that, just as applications may receive recognized events, so may other recognizers. This means that recognizers may interpret events (interpretations) produced by other recognizers without any special modifications.

Suppose an application developer wishes to allow the user to sketch or to type characters. If she has a character recognizer, she probably has to group mouse events into a *stroke* manually, before passing them to the recognizer (a stroke is an interpretation of a series of mouse events starting with a mouse down, a series of mouse drags, and a mouse up). She would then have to take the (potentially ambiguous) results and send them to the word predictor. In OOPS, if she added a character recognizer to her application, its results would automatically be routed to the word predictor. And OOPS provides a special recognizer that groups mouse events into strokes that would handle the job of providing the character recognizer with the correct input.

---

<sup>1</sup>A recognizer may do this either by caching interpretations to add them as new sources later, or caching sources in order to create an interpretation that includes all of them once it knows which sources to include.

### 4.3 Backwards Compatibility

Up to this point, we have made the assumption that every component receiving input fully understands the notion of ambiguity, and may be given input by the toolkit event dispatch system that will later be rejected by a mediator. While this assumption gains us a certain clean consistency, it requires that every application or interactor be rewritten to separate feedback about potentially ambiguous events from action on those events (if and when they are accepted). Alternatively, every component would have to support infinite, single event undo, a difficult proposition. In either case, existing components (particularly interface components) and applications are not aware of ambiguity, and to be backwards compatible with those applications and interactor libraries, OOPS must guarantee that they only receive unambiguous, accepted input. We will refer to those components as ambiguity-unaware components.

Our goal in building OOPS was not only to support the creation and mediation of ambiguous hierarchical events, but to do it below the application programming interface (API) understood by ambiguity-unaware components that receive input. This not only supports our goal of backwards compatibility, it means that, in the simplest case, application development remains unchanged, and none of the interactors in the large existing subArctic library need to be changed to work with OOPS. All input is received in a fashion that, on the surface, is identical to the way mouse and keyboard events are normally delivered in an unmodified toolkit.

The particular set of guarantees, and associated event dispatch algorithm, that we provide are those described in the previous chapter (See Source Code 3-12, p. 63): An ambiguity-unaware interactor only receives input either before any ambiguity arises or after all ambiguity is resolved for that particular event hierarchy.

To restate the problem in terms of the recognition process: The ambiguous event hierarchy in OOPS is built up as different recognizers interpret events. Interpretations are *not*

*certain*; they may be rejected. Because of this, it is important that no irreversible actions be taken based on those interpretations. Conversely, for interactors that do take irreversible action, we guarantee that they only receive unambiguous input. Thus, an ambiguity-unaware interactor either receives raw input before any interpretation has occurred, or receives recognized input after mediation.

OOPS determines on the fly which interactors are capable of handling ambiguous input, based on their class type. An interactor that should only receive unambiguous information will automatically be given input before any recognizers. As a result, if it consumes an input event, that event will not be available to recognizers. If an application designer wishes, she can add the interactor to a special list called the “after mediation” list. In this case, it will only receive unambiguous events (or interpretations at any level of the ambiguous event hierarchy) *after* all recognizers have had the opportunity to interpret those events<sup>2</sup>. As an example, a button should probably consume input before it is recognized since the user probably intends their input to be used by the button if she clicks on it. However, the word prediction application described at the end of this chapter needs to make use of the after mediation list, because the word predictor and the text area both make use of textual input. If the text area grabbed user input as soon as it was typed, there would be no opportunity for the word predictor to interpret those key presses as words.

The down side of the guarantee is that an interactor will either preempt recognition, or if it chooses to receive input after recognizers, it will not receive input until all mediation of ambiguity is complete. This means that it cannot provide any feedback to the user about the recognized events until all ambiguity is resolved. Since we support delayed mediation, this may result in a arbitrary lag in response time.

In summary, an application developer can use an existing application with a recognizer

---

<sup>2</sup>In subArctic, the interactors in a user interface form a spatial containment hierarchy. When an interactor containing other interactors is added to the after mediation list, everything it contains is implicitly also in the list unless otherwise specified. An exception to this policy is made for focus-based dispatch, which does not normally take the containment hierarchy into account. Details on how events are dispatched in subArctic (and OOPS) can be found in the subArctic user’s manual [33]

without significantly modifying that application. Assuming that the recognition results are of a type that the application already understands (*e.g.*, text), interactors in the interface will receive them, after any ambiguity has been resolved by mediators, through the same API as standard mouse or keyboard input. Additionally, the application designer may control how and when mediation is done by interacting with the mediation subsystem. The next section will explain the details of how this is done.

#### 4.4 Meta-mediators and Mediators in the Library of OOPS

The meta-mediators and mediators in the library of OOPS are based directly on the library elements described in the literature survey of Chapter 2. Meta-mediators represent policies for deciding which mediator should be used when. Mediators are interactive and automatic library elements that resolve ambiguity. Tables 4-1 and 4-2 summarize the standard set of meta-mediation policies and mediators provided with OOPS. The application designer is intended to add both mediators and meta-mediators to the library provided with OOPS, as appropriate, in the process of designing how mediation is done in his specific application.

In the case of our word predictor, for example, if the application designer wants to experiment with different approaches to mediation, she has two decisions to make: when is it an appropriate time to mediate, and how should the ambiguous alternatives be displayed? In the OOPS architecture, she would encapsulate each of these decisions in a separate mediator, and then assign those mediators to one or more meta-mediators in the mediation subsystem. Suppose she chooses to display the choices in an  $n$ -best list. She can select this mediator from the OOPS library (described below) and use it with any of the mediators she creates to encapsulate decisions about when to mediate. If she only wishes to mediate after the user has typed at least three characters, she might create an automatic mediator that defers mediation until that point. She would then add the automatic mediator and the  $n$ -best list to a meta-mediator that selects mediators in order from a queue. The automatic

mediator would be first, and thus would defer the hierarchy. As soon as three characters were typed, the automatic mediator would `continue_mediation()` (See Section 3.4.2) and the meta-mediator would then pass the hierarchy on to the next mediator in line, the *n*-best list.

Alternatively, the application developer might wish to use two different mediators, one if there are two choices, and the other if there are more than two. Suppose that the first mediator is a button that, when the user clicks on it, swaps the visible word, and the second mediator is an *n*-best list. The application designer would create a filter that checked if the number of choices was equal to two. She would add the button mediator to a meta-mediation policy called a *filter* meta-mediator that always checks a mediator's associated filter before passing a hierarchy to it. She would then add the *n*-best list to the same filter meta-mediator with a different filter that checks if the number of leaf nodes is greater than two. Whenever a hierarchy arrives for mediation, the meta-mediator will check the filters and make sure it goes to the right place.

These two approaches could also be combined. The application developer would have to ensure that two meta-mediators were installed: First, the queue meta-mediator, and then the filter meta-mediator. She would then add the automatic mediator to the queue meta-mediator, and the two interactive mediators to the filter meta-mediator (exactly as described in the previous paragraph). The automatic mediator would be first and defer the hierarchy. As soon as it calls `continue_mediation()`, control passes back to the queue meta-mediator. Since there are no more mediators in the queue meta-mediator, the deferred hierarchy is passed to the filter meta-mediator, which sends it to the right place based on the filters it contains.

The default set of meta-mediators and mediators installed by default at runtime in OOPS is a filter meta-mediator, followed by a queue meta-mediator. Both types of meta-mediators are described below in more detail, and a complete list of meta-mediation policies is given in Table 4-1. By default, a very simple mediator (“first choice”) which simply accepts

whatever events are passed to it, is installed in the queue meta-mediator. This guarantees that if the application designer does not wish to mediate, all ambiguity will be resolved invisibly and the top choice generated by each recognizer will be delivered to the interface. This base case is essentially identical to what would happen in a toolkit with no support for ambiguity.

In the following sections, we will describe the key elements of the meta-mediator and mediator library of OOPS, illustrating along the way how the mediators given in the example above might be built, as well as how standard meta-mediation policies, choice mediation, repetition, and automatic mediation may be used.

#### 4.4.1 Meta-mediators

When an event hierarchy needs to be mediated, the mediation dispatch algorithm described in the previous chapter passes that event hierarchy in turn to each installed meta-mediation policy, *via* a call to `meta_mediate()`. Meta-mediators are installed in a queue and can be reordered by the application designer at start-up or dynamically.

As a recapitulation from the previous chapter, remember that each meta-mediator implements a `meta_mediate()` method that passes the event hierarchy (represented by its root, a sensed event) on to its mediators according to the details of the specific meta-mediation policy (Table 3-4, p. 57, has a complete list of the methods and constants for the meta-mediation interface). For example, the filter meta-mediator has a filter associated with each mediator. It only passes the event hierarchy on to mediators whose filters return true. Consider the two-interpretation filter we described above. This filter checks each incoming hierarchy for two things. First, it checks if the hierarchy has any interpretations of type characters. Second, it checks if any of those characters has more than one additional sibling. Source Code 4-1 shows how a two-interpretation filter might be implemented.

A filter is used by a filter meta-mediator to determine if its associated mediator should mediate a given hierarchy. The mediation dispatch algorithm (given in Source

**Source Code 4-1:** An example implementation of the (one method) filter interface ( filter ())

```

// two_characters checks for hierarchies with
// exactly two sibling unambiguous characters.
class two_characters implements filter {
    // root is a pointer to the event hierarchy.
    // all_interps is the hierarchy flattened
    // (a common need for filters)
    Set filter (event root, set all_interps ) {
        Set res;
        for each interpretation in all_interps
            if ( interpretation .rejected () ) continue;
            if ( interpretation is of type characters )
                if ( interpretation has 2 siblings )
                    add interpretation to res;
        }
        // returns all interps that passed the filter
        return res;
    }
}

```

**Source Code 4-2:** The meta\_mediate() method in the filter meta-mediator

```

1 // in filter_meta_mediator, which has standard
2 // implementations of update, cancel, etc.
3
4 // returns PASS, RESOLVE, or DEFER
5 int meta_mediate(event root)
6 // put all of the events in root's tree in s
7 set s = root.flatten ()
8
9 for each mediator and filter pair {
10     set res = filter . filter (root, s);
11     if (res is not empty)
12         // this is implemented just like the
13         // example given in Source Code 3-10
14         switch(mediator.mediate(root, res)) {
15             case RESOLVE:
16                 if (not root.is_ambiguous())
17                     return RESOLVE
18             case PASS:
19                 continue;
20             case DEFER:
21                 cache mediator with root
22                 return DEFER;
23         }
24     return PASS;
25 }

```

Code 3-5) calls `filter_meta_mediator.meta_mediate(event root)` (shown in Source Code 4-2). First, the filter meta-mediator loops through its filters calling `filter(root, all_interps)` (line 10). Assuming `two_characters` is installed, it will check if the hierarchy contains two ambiguous sibling characters, and return a set containing them if it does. If the set is not empty (line 11), the meta-mediator will call `mediate(root, filter_result)` (line 14). Suppose the *n*-best list mediator is installed with this filter. It will display a menu on the screen and immediately return `DEFER` while it waits for further input from the user about the correct choice. The filter meta-mediator then also returns `DEFER` (line 22), since that is what the current mediator requested, and the mediation dispatch algorithm exits, and dispatch for the event hierarchy rooted in `root` is halted for now, allowing new events to be dispatched.

A complete list of meta-mediation policies is given in Table 4-1. We have just given a detailed example of how one might use the filter meta-mediator. By associating different filters with different mediators, the filter meta-mediator can also be used to select among different possible mediators. The queue meta-mediator, whose implementation is shown in Source Code 4-3, gives each mediator in a queue a chance to mediator in order.

The positional meta-mediator illustrates a different approach to selecting mediators based on their location and the location of the events being mediated (see Source Code 4-4). Suppose that in addition to the word predictor, a pen-based command recognizer has been installed that understands gestures, such as a circle to select text, and a squiggle to erase it. Mouse events outside of the text area are presumably not gestures selecting or erasing text. The positional meta-mediation policy could be used to guarantee that the gesture mediator would not be given any events to mediate that were located outside of the text area.

The simultaneous meta-mediator is an example of something that an application developer might wish to add to the toolkit. It allows several mediators to mediate the same hierarchy simultaneously. This might be used to allow a set of automatic mediators encoding



**Table 4-1:** A description of the default Meta-mediation policies provided with OOPS. *Positional* and *Simultaneous* are not yet fully implemented.

Type	Description
Filter	Contains an ordered list of mediators, each with an associated filter. The Filter meta-mediator applies each filter to the current ambiguous event hierarchy. The first mediator whose filter returns a non-empty set of matches gets a chance to mediate. When that mediator returns, if the event hierarchy is still ambiguous, the filter meta-mediator continues going down the list, allowing mediators with matching filters to mediate, until the hierarchy is no longer ambiguous. See <a href="#">Source Code 4-2</a>
Queue	Contains an ordered list of mediators. Gives each mediator in turn a chance to mediate the current ambiguous event hierarchy in its entirety, until the hierarchy is no longer ambiguous. See <a href="#">Source Code 4-3</a>
<i>Positional</i>	Allows each mediator to have an associated interactor or bounding box representing an area on the screen. Every event always has a bounding box (which, by default, it inherits from its source). If any bounding box in the event hierarchy overlaps the bounding box associated with a mediator, that mediator is given a chance to mediate. See <a href="#">Source Code 4-4</a> . An alternative to this implementation (shown in <a href="#">Section A.4</a> , page 176) simply traverses the interactor hierarchy looking for mediators that are also interactors. It then checks if the bounding box of any mediator it finds overlaps any of the events.
<i>Simultaneous</i>	A simultaneous meta-mediator would allow several mediators to mediate the same hierarchy simultaneously. This might be used to allow a set of automatic mediators, encoding different decisions about when interactive mediators should appear, to all defer mediation on the same hierarchy. The first mediator to call <code>continue_mediation()</code> would then be able to pass the hierarchy on to its associated interactive mediator.

**Source Code 4-3:** The `meta_mediate()` method in the queue meta-mediator

*// in queue\_meta\_mediator, which has standard  
// implementations of update, cancel, etc.*

```
// returns PASS, RESOLVE, or DEFER
int meta_mediate(event root)
    // put all of the events in root's tree in s
    set s = root.flatten ()
    for each mediator in queue {
        // this is implemented just like the
        // example given in Source Code 3-10
        switch(mediator.mediate(root, s)) {
            case RESOLVE:
                if (not root.is_ambiguous())
                    return RESOLVE
            case PASS:
                continue;
            case DEFER
                cache mediator with root
                return DEFER;
        }
    }
return PASS;
}
```

**Source Code 4-4:** The `meta_mediate()` method in the positional meta-mediator

*// in positional\_meta\_mediator, which has  
// standard impl. of update, cancel, etc.*

```
// returns PASS, RESOLVE, or DEFER
int meta_mediate(event root)
    // put all of the events in root's tree in s
    set s = root.flatten ()

    // location is a rectangular area of the screen,
    // or an interactor (which has a location
    // that may change dynamically with time)
    for each mediator and location pair {
        // retrieve events in hierarchy inside
        location
        res = elements of s inside location
        if (res is not empty)
            switch(mediator.mediate(root, res)) {
                case RESOLVE:
                    if (not root.is_ambiguous())
                        return RESOLVE
                case PASS:
                    continue;
                case DEFER
                    cache mediator with root
                    return DEFER;
            }
    }
return PASS;
}
```

different decisions about when interactive mediators should appear to all defer mediation on the same hierarchy. The first mediator to call `continue_mediation()` would then be able to pass the hierarchy on to its associated interactive mediator.

In summary, meta-mediators represent different policies for selecting which mediator should mediate which ambiguous event hierarchy. By controlling the order in which meta-mediators are installed, and the mediators added to each meta-mediator, the application designer can represent fairly complex choices about which mediators should be skipped or used in a given interaction, and more generally how mediation should be handled in a given application.

#### 4.4.2 Mediators

Once a meta-mediator determines which mediator should try to mediate a given event hierarchy next, it passes the root of that hierarchy to the mediator *via* a call to `mediate()`. The meta-mediator passes two arguments to `mediate()`, the root of the event hierarchy and a set of “relevant” events, if the meta-mediator has any knowledge of which ones are relevant. For example, the *filter* meta-mediator passes each mediator the results of the call to its associated `filter`. Source Code 3-9 in the previous chapter (p. 55) suggests a general implementation of `mediate()`, while an example of a specific implementation is given below (Source Code 4-5). In general, when a mediator receives an event hierarchy to be mediated, it has three choices. It can pass, by returning the value `PASS` and not modifying the hierarchy. It can resolve some portion of the hierarchy by accepting or rejecting events, in which case it returns `RESOLVE`. Or it can defer mediation until further input arrives, in which case it returns `DEFER`. When a mediator returns `DEFER`, it may continue to modify the event hierarchy, accepting and rejecting events as relevant information arrives (such as the user clicking on the mediator to indicate a choice). Once the mediator determines that it is finished with the hierarchy, it simply calls `continue_mediation()` to pass control of the hierarchy to the next mediator in line.

**Source Code 4-5:** An automatic mediator that pauses input until the user has typed three characters

```
// three_characters is a mediator that defers mediation until the user has typed at
// least three characters.
class three_characters implements mediator {
    int counter = 0;
    public int mediate(event root, set relevant_interps ) {
        counter++;
        deferred.add(root);
        if (counter == 3) {
            counter = 0;
            for (each root in deferred) {
                continue_mediation(root);
            }
        } else {
            return DEFER;
        }
    }
}
```

---

Consider the automatic mediator that waits until at least three characters have been typed. Source Code 4-5 shows how this might work<sup>3</sup>. Assume that this mediator has been installed with a filter that checks for characters. Each time `mediate()` is called, the mediator simply increments a counter and returns `DEFER`. When the counter reaches three, the mediator calls `continue_mediation()` on each of the deferred hierarchies, thus notifying the meta-mediator that it may continue the process of resolving those hierarchies.

In addition to the `mediate()` method, mediators support `cancel_mediation_p()` and `update()`, two methods that are also supported by meta-mediators (and described in full detail in the previous chapter in Section 3.4.2). The full complement of methods and constants supported by mediators is shown in Table 3-3. As with meta-mediators, a mediator is first asked to `cancel_mediation_p()` when an event hierarchy it has deferred on is modified. If it returns `false`, `update()` will be called with a pointer to the modified hierarchy and the particular event in the hierarchy that was changed. The “three characters” mediator, for example, does not really care how or whether a hierarchy is modified, it will return

---

<sup>3</sup>A more sophisticated version should check the characters in `relevant_interps` and increment the counter based on their number and relationship (*e.g.* siblings increment the counter, sources and interpretations don't).

**Table 4-2:** A comparison of the default mediators provided with OOPS.

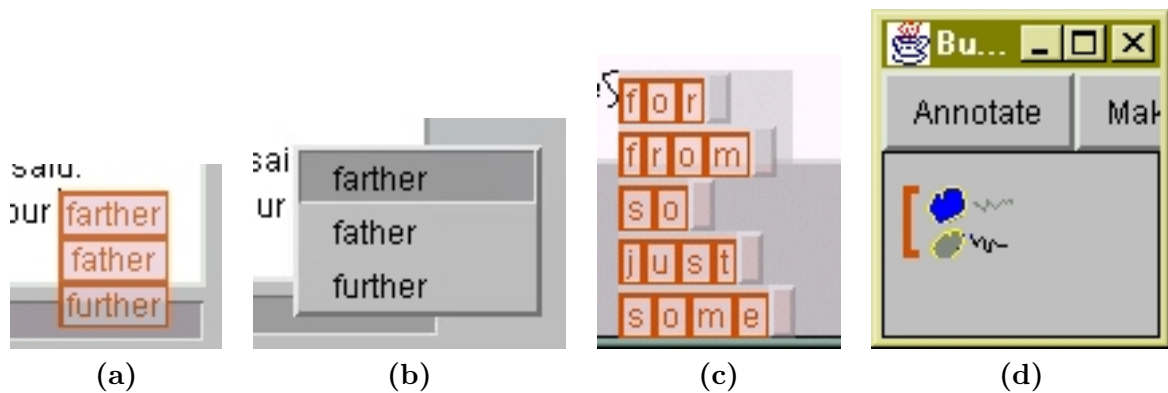
Mediator name	Type	Description
first choice	automatic	accepts the most certain event
pauser	automatic	(abstract) defers mediation until a certain condition is satisfied
action pauser	any	defers mediation until an action event arrives
stroke pauser	automatic	defers mediation until each stroke is completed
choice	choice	pluggable layout/feedback for multiple alternatives
repetition	repetition	base class for implementing repetition mediators
point	automatic	rejects very short strokes that should have been simple clicks

false if `cancel_mediation_p()` is called. On the other hand, an interactive mediator that has not yet made itself visible to the user (perhaps because it is waiting for the user to finish some task first) would probably return `true` when `cancel_mediation_p()` was called, since the user is not yet aware that mediation is in progress.

In addition to allowing the user to create mediators from scratch, OOPS comes with a set of default mediators based on the common themes found in the literature survey described in Chapter 2. Table 4-2 gives an overview of these mediators, described in more detail in the text below. Most of these mediators provide base classes which can be extended or parameterized.

#### 4.4.2.1 Choice mediators

Choice mediators are created by extending the choice base class. Figure 4-2 shows four sample choice mediators created in this fashion. The choice base class makes use of a pluggable feedback and a pluggable layout object (See Table 4-3 and Table 4-4 for details). Between them, feedback and layout provide pluggable support for all of the dimensions for choice mediators found in our survey of existing mediators (Section 2.2, page 20). The feedback object is responsible for providing user feedback about events and their context; and for supporting user input (selection of the correct choice). The layout object is responsible for



**Figure 4-2:** Four example choice mediators created from the OOPS choice base class. **(a)** A partially-transparent  $n$ -best list. **(b)** A standard menu-based  $n$ -best list. **(c)** An  $n$ -best list that supports filtering on correct prefixes (See Section 4.5.2). **(d)** A highly customized choice mediator (the line on the left of the radio buttons).

deciding when the mediator should be instantiated (become visible), and how the feedbacks for all of the events being mediated should be arranged on the screen.

Figure 4-2 shows four mediators that represent different combinations of feedback and layout classes. Three of the mediators (a, b and c) are using almost identical layout objects. All four have different feedback classes. A feedback class is expected to take an event and return an object that can display that event on the screen and will know when the user has selected that event. For example, (a) uses a feedback class called `button` that creates a label based on the event's textual representation. The button is semi-transparent and intentionally compact. It highlights itself and its associated events when the mouse is over it, and when it receives a mouse click, it notifies the choice base class that its event has been selected. (b) is a very similar class which returns non transparent objects that can be used in a standard menu. (c) uses an extension of (a) that is described in more detail in our case study (below) and in Section 6.1. (d) is fully customized and uses a feedback object that displays short horizontal dashes indicating each sketch.

**Table 4-3:** Methods implemented by a *layout* object in OOPS. Each layout object represents a specific event hierarchy (referred to below as “the current hierarchy”) on screen by showing the user a subset of the events in that hierarchy (referred to as the “relevant events”) with the help of a feedback object.

**void hide\_all()** Hide the current hierarchy (so that it is not visible to the user).

**void hide(event e)** Hide a specific event.

**void show\_all()** Show the current hierarchy (make it visible).

**void show(event e)** Show a specific event.

**void add(event e)** Add an event to the set of relevant events.

**void remove(event e)** Remove an event from the set of relevant events.

**Table 4-4:** Methods implemented by *feedback* objects in OOPS. A feedback object, when given an event, returns a representation for that event that can be used by the layout object to show the event to the user.

**String text\_feedback(event e)** Returns a textual representation of an event. The text may be displayed visually, in a menu for example, or aurally by speech.

**String interactor\_feedback(event e)** Returns an interactive component that provides visual feedback about an event. This is often implemented by calling the event’s `feedback()` method.

#### Source Code 4-6: An audio mediator

```
show_presence() {  
    // non-speech audio  
    play (“mediator_present.au”)  
}  
show_all() {  
    speak (“Did you mean”)  
    for (each event in showing())  
        speak(feedback.text_feedback(event));  
        if (event not last) speak (“or”);  
    speak (“?”)  
}
```

The layout classes used in (a,b and c) all inherit from a base layout class that only displays the top *n* choices (based on certainty, as expressed in the certainty field of an event), and handles issues such as methods for hiding and showing individual interpretations, as well as a default meta-mediator for when to instantiate (display itself on the screen).

All three layout classes are fairly similar, they position the feedback objects in a vertical column at the location of the original sensed input event that is the root of the ambiguous hierarchy. In contrast, the layout class used in part (d) of Figure 4-2 checks the the size and location of the ambiguous alternatives, and draws a vertical line to their left.

In addition to feedback and layout, the choice class supports a variety of behaviors in cases when the event hierarchy is updated with new events. In particular, it can be set to `reject_all()` existing choices, accept the top existing choice, `combine` the existing choices with the new choice, or `cancel`, which allows mediation to be canceled and begun again.

In summary, the base choice class is a fairly flexible class that can be used to create a variety of choice-based mediators. An application developer may create an instance of a choice class and assign a particular look and feel (along the five dimensions normally varying in choice mediators) by selecting feedback and layout objects from the library of OOPS. Figure 4-2 shows the forms of feedback that are currently supported. Alternatively, if a designer wishes, she may create her own feedback and layout objects. For example, if she wishes to create a mediator that displays choices aurally instead of visually, she would create a layout object by overriding two key methods. The `show_presence()` would need to be modified to indicate somehow to the user that the mediator is present (perhaps *via* non speech audio, or alternatively by speaking a phrase such as “Please clarify”). The `show_all()` method would need to be overridden to “lay out” the feedback. An audio mediator might contain Source Code 4-6:

All of the feedbacks provided with the OOPS library implement the `text_feedback()` method, so any of them could be extended to check for spoken (or DTMF) input accepting or rejecting each choice.



#### 4.4.2.2 Repetition mediators

As with choice, OOPS provides a base class supporting repetition mediation. The repetition base class is fairly generic and intended to encapsulate the major parameters of repetition identified in the survey (See Section 2.1, page 14):

**Modality:** The type of input used to do the repetition. This is modified by subclassing the repetition object

**Granularity:** The portion of input that is repaired. Users may repair part of their input instead of redoing all of it. The effect is that something is modified or replaced instead of a new event being created. Whether and how this is done is controlled by subclassing the repetition object.

**Undo:** The fate of the events being mediated. The events being mediated may be rejected before, or after, repair is done, or not at all. This can be set and changed by a simple method call.

OOPS provides one instance of this class, called *resegment*. This mediator allows the user to select an area of the screen, and provides rubber band feedback about the selection. It caches all recognition events that have not been closed that were generated by a specific recognizer (specified dynamically). When it finds cached events that fit within the rubber band box, it calls rerecognition on them. This creates a new interpretation, which the mediator dispatches. Undo of the original events is handled automatically, in that they are rejected, thus causing any classes providing feedback about them to undo that feedback.

Continuing with our word predictor example, a simpler repetition mediator might allow the user to speak the word he is trying to type. This mediator supports an alternative modality (voice), and works at the granularity of entire words or even sentences. It would automatically reject the currently visible choices, but it might combine information about other likely candidates with information returned by the speech recognizer.

In both of the examples just given, the repetition mediator needs to know which ambiguous events it is mediating. This is determined at meta-mediation time by the mediator's associated filter object. This means that this mediator should be used in conjunction with a filter meta-mediator which determines which types of ambiguity it handles. So, for example, the word predictor mediator would have an associated filter that filters for words produced by a word predictor.

#### 4.4.2.3 Automatic mediators

Automatic mediators generally encode heuristics for adding or removing events or for handling timing issues. Often, application-specific knowledge may be encoded in these mediators. All of the automatic mediators currently provided by OOPS are required to implement the set of methods required of all mediators and summarized in Table 3-3. Although there is no general base class for automatic mediators, there are certain categories of automatic mediators for which OOPS provides base classes. In particular, many of the decisions about when a choice or repetition mediator should appear on screen are actually encapsulated in automatic mediators that defer mediation until the appropriate time. These mediators all inherit from a `pauser` base class. The `pauser` class is also important because it can prevent a recognizer that requires a series of events from being preempted by another class that creates interpretations earlier. If those interpretations are accepted then the later recognition results will never be accepted since existing conflicting events will have already been chosen. An example related to this is the `three_characters` pauser described above (Source Code 4-5).

The pauser class requires one method to be implemented, `action_interp(event e)`. It is intended to be used with a filter, and it pauses mediation on every event hierarchy that successfully passes through the filter. Separately from that, it registers to receive every input event as a recognizer. Each time it receives an event, it calls `action_interp(event e)`.

If this returns true, then the pauser calls `action_found(e.root())`, which by default continues mediation on every event hierarchy that is currently paused. As an example, the `stroke_pauser` mediator provided by default with OOPS simply overrides `action_interp(event e)` to check if `e` is a completed stroke (in which case, it returns true). A stroke is a series of mouse events starting with a mouse down, followed by several mouse drags and a mouse up, and many gesture recognizers only work on entire strokes. The `stroke_pauser` helps to guarantee that recognizers that recognize strokes are not preempted by interactors using the same mouse events.

Pausing also could be encoded as a meta-mediation policy, since meta-mediation is generally in charge of deciding *when* (as well as how) mediation happens. However, by putting pausing in an automatic mediator, we gain a certain flexibility: a pauser can easily be inserted between any two mediators even when they are part of the same meta-mediation policy.

In addition to various subclasses of pauser, OOPS includes two other automatic mediators. The first, a confusion matrix, is important because it allows an application designer to use a third party recognizer that does not generate multiple alternative suggestions without modifying that recognizer. The mediator, instead, uses historical information about how the recognizer performs to suggest possible alternatives. A confusion matrix can also be useful when a recognizer regularly confuses two inputs because of a lack of orthogonality in its dictionary. For example, ‘u’ and ‘v’ are very similar when sketched, and Graffiti<sup>TM</sup> recognizers often confuse them. Similar situations arise in speech. For example, Marx and Schmandt compiled speech data about how letters were misrecognized into a confusion matrix, and used it to generate a list of potential alternatives for the interpretations returned by the speech recognizer [52].

The confusion matrix provided with OOPS is currently limited to recognizers that produce text events. When a text event arrives, it looks that string up in a hashtable. The list of strings that is returned are other likely interpretations. The mediator creates additional

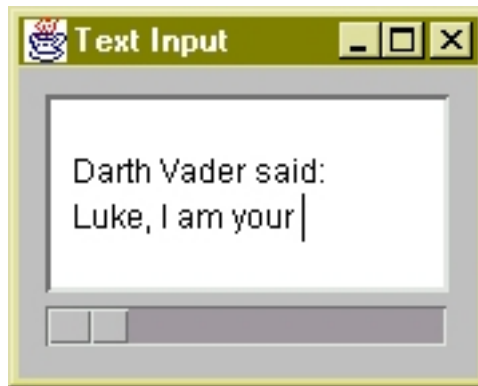
events for each interpretation in that list. The information about which events should be added, and their relative likelihood, must currently be provided by the application developer. However, in the future, we plan to develop an algorithm that will use information about which events are accepted, or rejected, to gather automatically the historical data needed to update the matrix.

The last automatic mediator provided by the system is a point mediator, that identifies very short strokes, really intended to be a single click. This is a common problem with pen-based interfaces, where a very quick press of the pen may generate several unintended mouse drag events. Normally, clicks on top of interactors are sent to the interactor, while strokes are recognized even when they overlap the interactor. This mediator handles cases when the end user is trying to click, but accidentally creates a very short stroke.

In summary, automatic mediation in OOPS plays an important role in helping to control the time at which interactive mediators are displayed, as well as guaranteeing that recognizers are not preempted when they have to wait for a series of input events. Automatic mediators may also encode information about problems with specific input devices and recognizers (such as a click that comes out as a stroke).

## 4.5 Case Study: A Word Predictor

The example we have been using throughout Chapters 3 and 4 is based on a text editor that is distributed as a test application in subArctic, shown in Figure 4-3. This application contains only a simple text entry window (`text_entry`) from subArctic's library. Since there is no recognition in this application, input is handled very simply. When the user presses a key on the keyboard, a `keypress` event is created. The event dispatch system passes it to the current text focus interactor (`text_entry`) by calling `text_entry.keypress()`. The window responds by storing the event, and displaying it on the screen. This section will step through the modifications that may be done to add support for word prediction to this application.



**Figure 4-3:** A simple text editor

---

The first modification the application designer makes is to add the word predictor to the application. When the user types a character, the word predictor generates a set of words as possible interpretations. Each additional character is used to update the set of possible words. Our application developer may use the word predictor that comes with OOPS, use a third party word predictor, or create one of her own. In order to install the recognizer, the application developer simply makes sure that it is receiving the particular type of input it needs to interpret (in this case, text). This is done by adding the word predictor to the multiple text focus set of the event dispatch system (`multiple_text_focus.add_to_text_focus()`). Note that this set allows multiple objects to have the text focus, and resolves any resulting ambiguity about who should get to use the text *via* mediation. Now, when the user types a character, it is dispatched to the word predictor. The word predictor creates a group of interpretations, resulting in recognition ambiguity.

The text entry area is also in the text focus set by default. Conventionally, input goes directly to the text entry area (before it is interpreted). Since a word predictor uses the same input as the text entry area to do its prediction, this presents a problem: The text entry area consumes typed text immediately, and circumvents the recognition, because it is

ambiguity unaware, does not support undo, and will always act immediately on any text it receives. In contrast, an ambiguity aware interactor separates feedback from action, acting on input only if and when it is accepted. Hence, an ambiguity aware text entry area would display feedback about an ambiguous event, and simply remove that feedback if the event is rejected.

Since the text entry area is not aware of ambiguity, the toolkit provides the option for the designer to add the text entry area to a special “*after mediation*” list stored in OOPS. This is done by calling `add_to_after_mediation(text_entry)`. In this case, the text entry area will still only receive unambiguous textual input, and will not receive any of the source events interpreted by the word predictor. Instead, it will receive the final interpretations created by the word predictor, once ambiguity has been resolved.

In order to resolve the ambiguity generated by the word predictor, the application designer adds a mediator to the system. This mediator represents a strategy for deciding when and how the ambiguity should be resolved. By default, mediation will be done using OOPS’ most basic automatic mediator (`first_choice`), which simply selects the most certain choice, accepts it, and rejects all the other interpretations. This is very similar to what happens today in many systems that do not support ambiguity. The original text input events, which have been “used” (interpreted) by the word predictor, are never shown to the text entry area. But the accepted choice is now unambiguous, and has no interpretations, so it is dispatched (finally, and for the first time) to the `text_entry` area, which displays it on screen. The text entry area knows nothing of the mediation that just went on, the user could just as easily have typed each letter of the word separately.

But what if that word is wrong? The user can always use repetition of some sort (*e.g.* deleting the word and trying again) to correct it. Alternatively, the designer could install an interactive mediator from the library, such as that in Figure 4-4, that asks the user for input. This is done by instantiating the mediator, and adding it to the appropriate meta-mediation policy. In some cases, a meta-mediation policy may also be instantiated



**Figure 4-4:** A simple mediator handling the word-prediction.

and installed. Source Code 4-7 shows the code that creates a choice mediator (line 17) with a button style of interaction (line 21). The code adds it to a filter-based meta-mediation policy (line 24). That filter looks for events of type `characters` (line 19).

In summary, the application designer has made three simple changes to the original application. First, she instantiated the recognizer and added it to the text focus. Second, she added `text_entry` to the `after_mediation` list of the dispatch system. And third, she instantiated a mediator and added it to the mediation list. The code for these changes can be found in Source Code 4-7.

As she gets more involved in the design of the mediator for this application, our developer will realize that there are two major questions she needs to answer, which are closely tied together.

1. What about the large number of words commonly returned in the initial stages of word prediction? Traditionally, a word-predictor shows up to  $n$  choices in a menu (sorted by certainty).
2. What if the user does not mediate, but instead types another character? Traditionally, each new letter typed causes the old set of words to be replaced by a new set of predictions.

**Source Code 4-7:** The additional code needed to add word prediction to a simple text entry application

```
1 // build_ui() is a method that is called when the application is first created,
2 // and contains the code that sets up the user interface (for example, by creating
3 // buttons and menus and putting them in the appropriate places on screen)
4 public void build_ui(base_parent_interactor top) {
5     // create the word predictor
6     word_predictor pred = new word_predictor();
7     // add it to the multi text focus, which allows multiple focus objects
8     manager.multi_text_focus.add_to_focus(pred);
9     text_edit txt;
10
11     // <.... various unmodified code setting up UI - initializes txt ...>
12
13     // add the txt area to the "after-mediation" list
14     ambiguity_manager.add_to_after_mediation(txt);
15
16     // create a choice mediator
17     choice ch = new choice();
18     // an extension of the filter class that tells the choice mediator which events to mediate
19     choose_characters cf = new choose_characters();
20     // the user is selecting the look and feel of the choice mediator here
21     ch.set_interaction(new button_feedback(), new button_layout());
22
23     // install the choice mediator with the mediation system
24     ambiguity_manager.filter_meta_mediator.add(ch, cf);
25 }
```



The remainder of this section will discuss these two questions in more depth, and then go on to illustrate how the user can combine traditional word prediction with other forms of recognized input such as pen input (sketched characters).

#### 4.5.1 Many words

The default choice mediator in the library of OOPS displays up to  $n$  (specified at instantiation time) ambiguous alternatives (sorted by certainty) in a menu. However, during the initial stages of word prediction, the word predictor may return tens to hundreds of possibilities. If an event is created for each of these, it will be dispatched, resulting in hundreds of dispatches and a significant slow down in interaction<sup>4</sup>.

Instead, a recognizer may choose to create only the top  $n$  interpretations, and then wait for calls to `rerecognize()`, at which point it returns the next  $n$  possibilities, and so on. This approach is fairly general because the system for retrieving additional interpretations is the same, no matter which recognizer is involved. We plan to update the default choice mediator to support this approach. It will provide a “more choices” option that results in a call to `rerecognize()`. The downside of this approach is that mediators will be acting on incomplete information until the additional choices are requested.

In order to provide immediate access to all of the choices, without the slow down of having them all dispatched, our designer must modify the mediator and recognizer that she is using so that they can communicate directly with each other. This is facilitated by special hooks in our event system that allow additional data to be associated with an event, independently of the event hierarchy. The designer updates the word predictor to create only interpretations for the  $n$  most likely candidates, and store a vector containing all of the other possibilities in `user_data` (a variable associated with each event in OOPS, see Table 3-2).

---

<sup>4</sup>Since event dispatch is synchronous, this can create a significant bottleneck in the interactivity of the interface

Because this requires our designer to modify the default mediator in order to give it access to the information in `user_data`, this results in a slight loss of generality. Only mediators that have been modified to know what is stored in `user_data` will have access to the complete list of possibilities. The necessary modifications are made not to the choice mediator itself, but to the layout object used with each choice mediator. Changes are made to the methods that initialize the display, add additional events, and update the visible events. For example, the default word predictor provided with OOPS stores a vector of `word_info` objects in `user_data`. The layout object is modified to keep track of `word_info` objects as well as events. The feedback object must also be modified to create a visual representation for `word_info` objects. In both cases, a duplicate of each of the methods shown in Tables 4-3 and 4-4 is created that takes a `word_info` object instead of an event as input.

#### 4.5.2 A new mediator

Why would the designer wish to modify the mediator to display more than the  $n$  interpretations created by the recognizer? One example, described in more detail in Section 6.1 is an  $n$ -best list with support for repetition. That mediator allows the users to filter the set of words being displayed by clicking on a letter in one of the top choices. For example, if one of the choices in the list is “farther” and the user actually wants the word “farthing,” the user may click on the letter ‘h’ to indicate to the mediator that it should only display words that have the prefix “farth.” In order to do this, filtering must include all possible alternatives. In addition, once filtering is added to the mediator, it makes sense to show not the most likely choices, but the most likely, *most different* choices. This provides more filtering possibilities. So even though only  $n$  choices are visible, this makes use *all* of the choices that are generated.

The default mediator, shown in Figure 4-4, displays a list of buttons. The designer must make each letter active to support filtering. Thus, by clicking on a specific letter, the user



**Figure 4-5:** The modified button returned by the feedback object for a choice mediator that supports filtering. (a) The button returned by the default feedback object (b) The modified button.

---

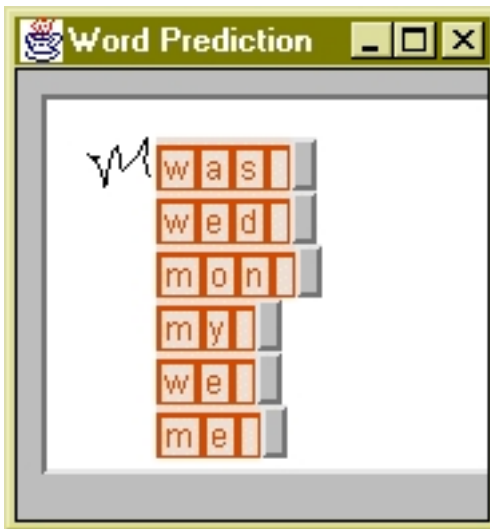
is able to indicate a filtering prefix (the letters before and including the letter on which she clicked). The original buttons are generated by a special feedback object. She subclasses this object to return a filter button on which each letter can be clicked. Figure 4-5 shows the original and modified button. Now, to select the entire word, the user must click on the grey box at the right. To filter on part or all of the word, the user clicks on a letter.

In order to make use of the additional data provided with the top  $n$  events, our designer must also subclass the default *layout* object used by the choice mediator. This object is responsible for assembling the feedback objects for each of the events it wishes to display into a coherent interactor (*e.g.* a menu). The new feedback object informs the new layout object when the user requests a filter. It filters through both the events and the data (`word_info` objects) returned by the recognizer. It then removes or adds new feedback objects as necessary to reflect the results of the filtering.

### 4.5.3 Combining recognizers

Because interpreted and raw events are treated equivalently in OOPS, it is possible to combine recognizers without modifying either one to know the other is operating. A mouse-based mediator for word prediction would be especially effective in a pen-based setting such as a PDA (personal digital assistant). A Graffiti<sup>TM</sup> recognizer could be added to the interface, so that all interaction may be done with a pen.

OOPS comes with a third party recognizer (provided by Long *et al.* [47]), that simply takes mouse input, matches it against a Graffiti<sup>TM</sup> gesture set, and returns letters. It may



**Figure 4-6:** Word prediction in combination with pen input

---

return multiple letters if the mouse input is ambiguous. In this case, the word predictor will predict words starting with *any* of those choices. The result is shown in Figure 4-6.

As a side note, it turns out that the `text_entry` area consumes mouse input (for selection and to add it dynamically to the text focus). Just as the text entry's need for text focus had the potential to conflict with the word predictor's use of text, the use of mouse input causes similar conflicts with the needs of the Graffiti<sup>TM</sup> recognizer. For the purposes of this application, we subclassed the `text_entry` area to remove all use of mouse input and to give it the initial text focus by default.

## 4.6 Different Levels of Involvement

In summary, this chapter and the preceding example illustrate how OOPS supports a range of needs for GUI development.

**Basic Recognition** At the most basic level, it is possible to add a recognizer to an application without making any changes at all to the writing of that program. When the

recognizer returns ambiguous results, the dispatch system automatically routes those results to a default mediator. In cases where the recognizer needs to intervene before events are received by the application, the application developer can easily ensure that the application receives those events after the recognizer, by making use of the `after_mediation` functionality.

**Using the library** At the next level, a user can select different mediation techniques from the library, and control how and when a mediation technique is selected by using filters and automatic mediators such as pausers.

**Extending the library** Next, the user can create her own meta-mediation policies, and mediation techniques and make use of them. Although recognition, mediation, and the application separated, the user can build domain-specific techniques that communicate across all three through additional data stored in the event hierarchy.

**Application use of ambiguous data** Finally, the user can modify an application to interact with ambiguous data as well as mediated/unambiguous data and to provide the user feedback about the ambiguous data even before ambiguity is resolved.

The first three of these levels have been demonstrated in the case study of word prediction given above. The last level is used extensively in Burlap, the application described in the next chapter.

The next three chapters represent the validation portion of this thesis. The implementation of Burlap shows that OOPS is sufficient to implement the kind of complex application found in current systems, specifically the system called SILK [46]. It should be noted that Burlap is not a complete implementation of SILK, and that Burlap includes many features not found in SILK. However, our general goal was to show that OOPS could be used to reproduce the functionality of a fairly complicated recognition-based application, and we feel that Burlap aptly demonstrates this.

In Chapter 6 we show that OOPS allows us to address problems not previously solved. There, we describe three new mediation techniques that address problems highlighted in our survey of standard approaches to mediation (Chapter 2).

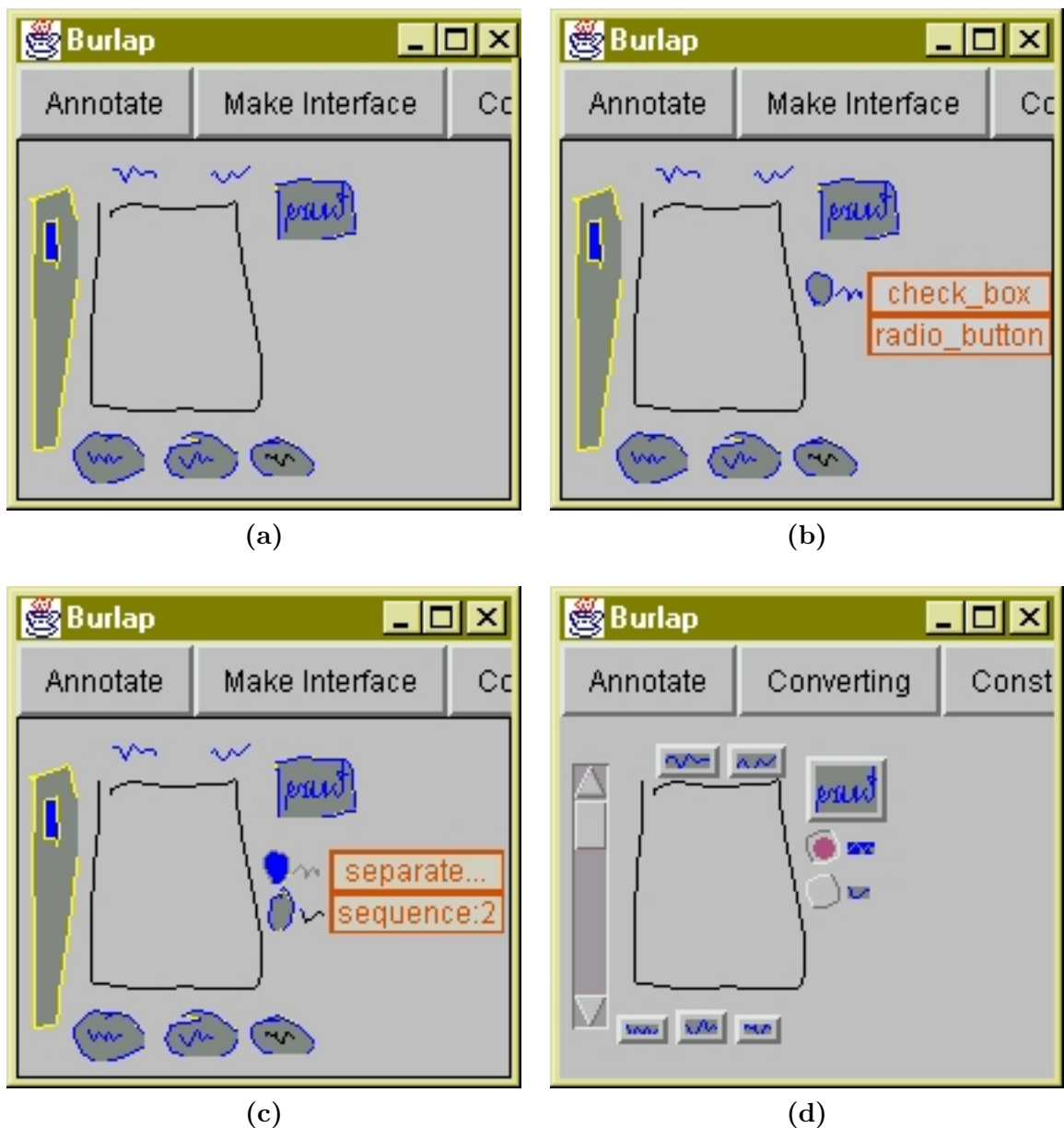
Finally, Chapter 7 shows how the architecture presented in Chapter 3 can be applied in a new domain, context aware computing. Context aware computing involves using contextual, implicit information about the user, her environment, and things in that environment, to provide information and services that are relevant. We modified the Context Toolkit [75, 18, 20], a toolkit that supports the development of applications that make use of context, to include support for ambiguity and mediation.

## Chapter 5

### AN EXTENDED EXAMPLE: DEMONSTRATING SUFFICIENCY

The goal of this chapter is to demonstrate that OOPS provides sufficient support to create the kind of complex applications found in existing recognition-based applications. We chose to show this by re-implementing one particular complex application, the sketch-based user interface builder SILK (Sketching Interfaces Like Crazy) [46, 44]. SILK is a drawing tool for early sketching of a graphical user interface. It allows the end user to sketch interactive components such as buttons, scrollbars, and check boxes, on the screen. The purpose of SILK is to allow designers to create very rough prototype interfaces, without losing the flexibility and rapid action that sketching provides.

The application described in this chapter, called Burlap, was built using OOPS and provides the core functionality of SILK as well as sophisticated support for mediation. In SILK, mediation is done through a separate dialogue window that contains support for mediating any and all issues that might arise. In Burlap, mediation is done in a variety of ways, and is integrated into the same area used for sketching. Figure 5-1(a) shows a sample user interface sketched in Burlap. An imperfect recognizer is converting the end user's sketches into interactors. The end user can click on the sketched buttons, move the sketched scrollbar, and so on. This chapter begins with a brief overview of Burlap. We then break Burlap down according to the basic research areas discussed in the first two chapters of this thesis. First, we explain the different types of ambiguity that arise in Burlap, and how they are handled. Second, we discuss the mediators used in Burlap, according to



**Figure 5-1:** The end user is sketching a sample interface to a drawing program in Burlap (a simplified version of SILK [46]). At any point in parts a-d of this figure, the end user may interact with the screen elements he has sketched. **(a)** A prototype interface to a drawing program, including (left to right) a scrollbar, a drawing area and two menus and three buttons, and a print button. **(b)** The end user adds a radio button. A choice mediator (an  $n$ -best list) asks the end user if she has sketched a check box or a radio button. **(c)** The end user adds another radio button. Another choice mediator asks the end user if she wants the two buttons to be linked (so that selecting one de-selects the other). **(d)** The end user selects “Make Interface,” and a Motif style interface is generated.



the categories identified in our survey. The chapter ends with a discussion of the impact of OOPS on the creation of Burlap, similar to the case study presented in the previous chapter for the word-prediction application.

## 5.1 Overview of Burlap

As stated, Burlap is an application that allows users to sketch user interfaces. Figure 5-1 illustrates a sample interaction in Burlap. Figure 5-1(a) shows a prototype interface for a drawing program that has been sketched in Burlap. The large central rectangle is intended to be a drawing area. The user has sketched a scrollbar (the narrow vertical rectangle to the left of the drawing area) and two menus (the squiggly lines at the top). The button at the upper right is labeled print, and the three buttons at the bottom are not yet labeled.

In Figure 5-1(b), the end user has added a radio button (below the print button). Radio buttons and check boxes look very similar when sketched, and are easily confused with each other. Once the sketch is recognized, the system indicates that some interpretations have been generated by changing the color of the lines and adding shading. If any ambiguity is present, it will be tracked *indefinitely*, until the end user tries to interact with the radio button. At this point, to react correctly, the system must know what the sketch is, and brings up a choice mediator asking the end user which interpretation of the input is correct.

The particular interaction described above is made possible because, first, a stroke pauser halts mediation until the sketch can be recognized (as a radio button/check box). Second, an action pauser pauses mediation until the user interacts with the button. And finally, a choice mediator displays the two choices. This description is a simplification of a complex process supported by OOPS' mediation subsystem. Figure 5-6 shows the complete set of mediators that is provided with OOPS.

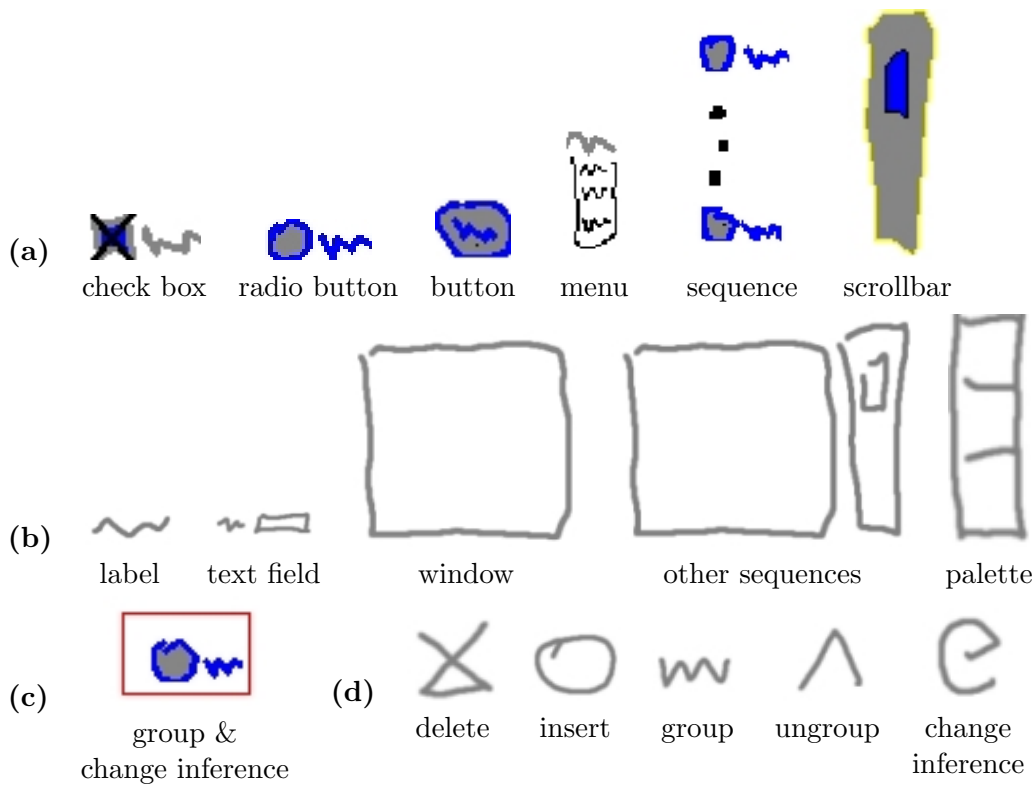
Continuing with the example interaction in Figure 5-1, the end user selects the radio button option, her intended interpretation. She then sketches a second radio button, below

the first, as shown in Figure 5-1(c). Here, the end user has drawn a sequence of two radio buttons, and the system recognizes that they are vertically aligned. This indicates that they probably should be grouped in sequence, that is, clicking one unsets the other and vice versa. However, the end user may have been intending to draw two rows of horizontal radio buttons in which case the radio buttons should be kept separate, and because of this ambiguity the system displays a menu indicating both choices.

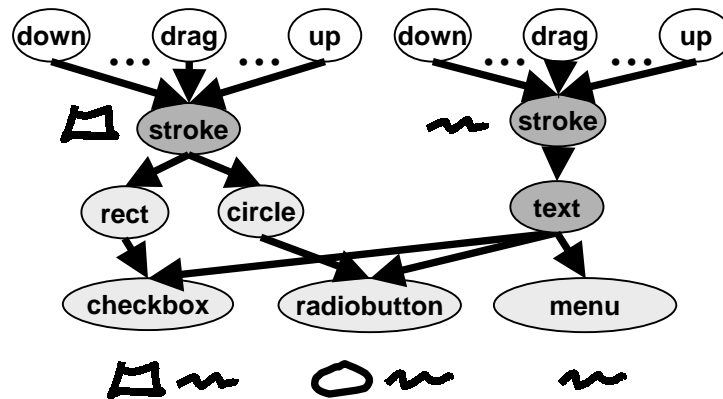
Finally, in Figure 5-1(d), the end user has selected the “Make Interface” button, and an X Windows motif-style interface is generated. When the system is in “Make Interface” mode, a sketch will still be recognized, and as soon as all ambiguity associated with a sketch is resolved, it will be converted to a motif-style interactor. At any point during her interaction, the end user may select the “Constrain” button to enter constrain mode. In this mode, the end user selects groups of buttons and they are aligned in rows or columns based on the aspect ratio of the selection rectangle. Burlap supports one other mode, called “Annotation” mode, in which the end user may annotate the user interface with comments/drawings that are not intended as interactive components. When in annotation mode, nothing is recognized.

Burlap includes input from two different recognizers. The first, a third-party recognizer provided by Long *et al.* [47], recognizes strokes as basic elements such as lines, rectangles, and circles. Long’s recognizer can be trained to work with a specific set of gestures, and we used a subset of the gesture set provided with SILK [44], shown in Figure 5-2. This recognizer only returns the top choice for each gesture, and we use a confusion matrix to supplement that choice with other likely candidates. See p. 95 for a full description of how the confusion matrix provided with OOPS works.

The second recognizer used in Burlap, the interactor recognizer, looks for groups of basic elements that may be interactive components. This recognizer is based on the rules used by the interactor recognizer in SILK. It uses information about the size and relative position of circles, rectangles, lines and *etc.*, and interprets them as interactive components (which may



**Figure 5-2:** A comparison of gestures recognized by Burlap and SILK. Gestures are simply sketched in both Burlap and SILK. Commands are drawn with the right mouse button depressed in both Burlap and SILK, so there is no ambiguity between commands and sketches of interactive components. (a) The gesture set supported by Burlap. For menus, only the squiggle (grey) is sketched. Sequence refers to any sequence of radio buttons or check boxes horizontal or vertical. Scrollbars can also be horizontal or vertical. (b) Additional gestures supported by SILK but not by Burlap. Other sequences includes sequences of buttons, menus (a menubar) and combinations of windows and scrollbars in different orientations. (c) Commands supported by Burlap (the red selection box). This command is invoked by using a different mouse button to avoid confusion with sketching. (d) Command gestures supported only by SILK. A complete implementation of Burlap should include delete, insert and ungroup (group and change inference are combined in Burlap). For now, the mediators provided with Burlap are sufficient to handle most system-caused errors, and many user errors, without them.



**Figure 5-3:** A sample of the hierarchy generated by a combination of the unistroke gesture recognizer and the interactor recognizer. The white events are sensed mouse events. The events in light grey are ambiguous.

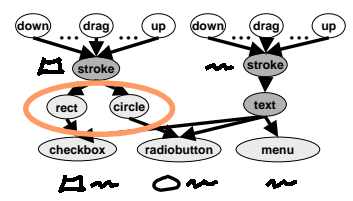


eventually be converted to motif-style interactors). When we built the interactor recognizer, we chose not to limit the number of possible suggestions that it generates<sup>1</sup>. However, since we recognize a smaller variety of interactive components than SILK does, this does not have a large impact on the amount of ambiguity.

Figure 5-3 shows an example of the sort of event hierarchy commonly generated by these two recognizers in tandem. Figure 5-1(b) shows the sketch that generated this hierarchy (the new radio button). The user has sketched two strokes (represented at the second level of the hierarchy). The third level of the hierarchy represents the interpretations done by Long’s recognizer, combined with the confusion matrix. The result was that the first stroke is either a *circle* or *rectangle* (this is a case of recognition ambiguity), and the second stroke is *text*. The results generated by the first recognizer were further recognized by the interactor recognizer, which generated the third level of the hierarchy. Here we have three possibilities, each with overlapping sources: *checkbox*, *radiobutton*, and *menu*.

The remainder of this chapter will illustrate some of the complexities of Burlap. First, we will show how it handles all three types of ambiguity highlighted in Chapter 1. Second,

<sup>1</sup>Landay’s thesis work gives guidelines for using relative certainty to eliminate options, that we currently do not use.

**Table 5-1:** Types of ambiguity in Burlap

Type of Ambiguity	Source of Ambiguity	Example
Recognition	Confusion Matrix	(from Figure 5-3) 
Segmentation	Interactor Recognizer	
Target	Ambiguous clicks. See Chapter 6.	

we discuss the four meta-mediators and ten mediators installed in Burlap and explain how they work together to support the styles of mediation illustrated above. The chapter ends with a discussion of the impact of OOPS on the creation of Burlap, along the lines of the case study presented in the previous chapter for the word-prediction application.

## 5.2 Types of Ambiguity

The types of ambiguity demonstrated by Burlap are summarized in Table 5-1. Recognition ambiguity is the most prevalent type of ambiguity in Burlap. Burlap uses a confusion matrix to generate recognition ambiguity, because its third party recognizer only returns the top choice. An example is the rectangle and circle in Figure 5-3. The circle was the interpretation returned by the recognizer, the confusion matrix added the square because of historical information indicating that they are often confused.

Segmentation ambiguity is generated by the interactor recognizer. Remember that segmentation ambiguity arises when there are multiple ways to group input events. Algorithmically speaking, this happens when two different interpretations have overlapping but differing sets of parents. All of the interactive components in Figure 5-3 fit this definition. They each share only one parent (source) event with each other. Another example of segmentation ambiguity in Burlap is illustrated in Figure 5-1(c). In this case, the question is whether the radio buttons should be grouped as a unit or remain separate.

Target ambiguity may also arise in Burlap. For example, the check mark at the bottom right of Table 5-1 crosses two radio buttons. Which should be selected? Currently, this type of target ambiguity is not detected by Burlap. Ambiguity may also arise between a sketched interactive component or an interactor, and a recognizer. For example, how do we know if the end user intended to click on a button or sketch a new interactive component in Burlap? This is handled by an automatic mediator, without end user involvement. A third example of target ambiguity in Burlap involves a separate recognizer that treats the mouse as an ambiguous point, for example represented as an area, as described in Chapter 6 (See Figure 6-5).

## **5.3 Mediation**

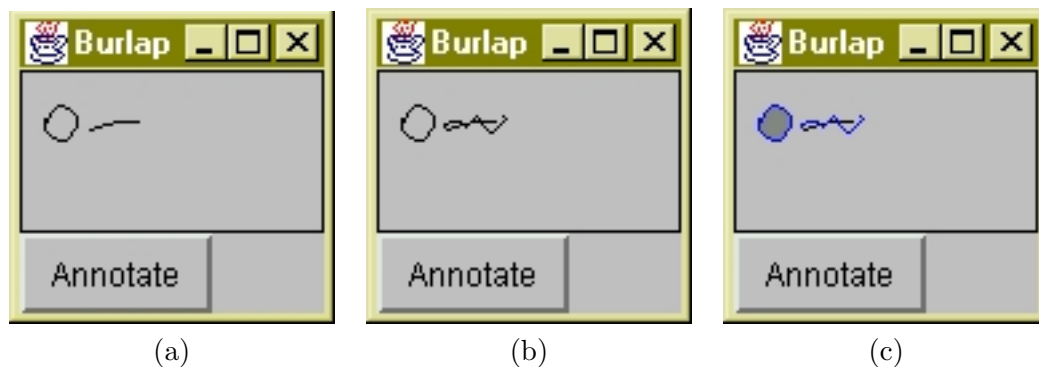
The full set of mediators installed in Burlap is shown in Table 5-2. Their organization within Burlap is shown later in Figure 5-6. The remainder of this section will be devoted to explaining what roles they play and how the end user interacts with them.

### **5.3.1 Repetition mediation**

Burlap handles rejection errors by allowing the end user to redraw any part of the interactor. This is an example of un-mediated repetition: there is no mediator involved in this task, the repeated strokes are handled just like the original strokes. Figure 5-4 shows an example

**Table 5-2:** The full complement of mediators that are used by Burlap. See Figure 5-6 for the relationship between mediators and policies. More detailed descriptions of the repetition and choice mediators are given in Table 5-3 and Table 5-4, respectively. *Name* is the label for the mediator used in the text of this document. *Superclass* is the base class from which mediator inherits. *Category* is the mediator type (repetition, choice, or automatic). *Policy* is the meta-mediation policy in which the mediator is installed. *Description* is a summary of the mediator’s purpose.

Name	Superclass	Category	Policy	Description
Magnification	Mediator	Repetition	Magnification Policy	Magnifies mouse events that overlap multiple sketched components or interactors
Property Preferable Mediator (PPM)	Property Filter	Automatic	Queue Policy1	Accepts events that match a property (which subclasses filter):
Is Annotation ..	Property	Automatic	PPM	Accepts annotations
Stroke Killer ...	Property	Automatic	PPM	Rejects strokes
Stroke Pauser	Pauser	Automatic	Queue Policy1	Defers mediation until each stroke is completed
Point Mediator	Mediator	Automatic	Queue Policy1	Looks for strokes that <u>should have been clicks</u>
Rerecognition	Repetition	Repetition	Filter Policy	Handles rejection errors through repetition
Interactor Mediator	Choice	Choice	Filter Policy (filters for sketched components)	Shows the end user possible interactors once an action is taken
Sequence Mediator	Choice	Choice	Filter Policy (filters for sequences)	Shows the end user possible groupings of radio buttons that <u>need to be mediated</u>
Action Pauser	Pauser	Automatic	Queue Policy2	Defers mediation until the end user takes an action <i>e.g.</i> clicks on an interactor



**Figure 5-4:** An example of repetition in Burlap. (a) The end user trying to sketch a radio button. (b) Due to a rejection error she must repeat part of her input (partial repair repetition). (c) Now the recognizer responds appropriately.

of repetition in Burlap. The end user’s first attempt at sketching a radio button is not recognized (a). She repeats part of the sketch by drawing the text of the radio button again (b), and recognition is completed (c). There is no need for undo; it does not matter if there are a few extraneous strokes on the screen (they actually add to the “sketched” effect)<sup>2</sup>

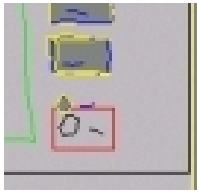
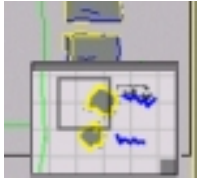
In addition, Burlap supports a form of mediated repetition (guided rerecognition) by allowing the end user to select unrecognized events. The mediator is given a filter that tells it to cache all events created by the interactor recognizer. When the user selects an area of the screen (using the right mouse button), the mediator is invoked and checks which cached events are inside the selection box. It then asks the interactor recognizer to generate a new set of possible interactive components from those strokes by calling `resegment()`. See Section 3.5.2.2 for a more detailed description of how `resegment()` works. A quick summary of *Rerecognition*’s features as compared to other mediators in the literature survey of Section 2.1 are:

- It uses pen input
- It uses an alternate modality (a selection box rather than sketches)

<sup>2</sup>These strokes remain visible when the “Make Interface” button is selected, but should the interface be written out to a file as code (this is not implemented), it would be trivial to eliminate them.



**Table 5-3:** A comparison of two repetition mediators in Burlap. Illustrates how they vary along the dimensions defined in Section 2.1

Name	Modality	Undo	Granularity	Example
Rerecognition	Selection box (different)	Automatic	Strokes	
Magnification	Click (same)	Automatic	Strokes	




- It handles undo implicitly
- It works at the granularity of strokes

Finally, the magnification mediator shown in Table 5-3 can also be viewed as a form of repetition. When the end user's click overlaps multiple sketched components or interactors, the mediator magnifies that area of the screen. The end user then tries again in the same modality (clicking), and when he succeeds in clicking on only one component, the mediator goes away. A more detailed description of this mediator and its applications can be found in Section 6.3.

### 5.3.2 Choice mediation

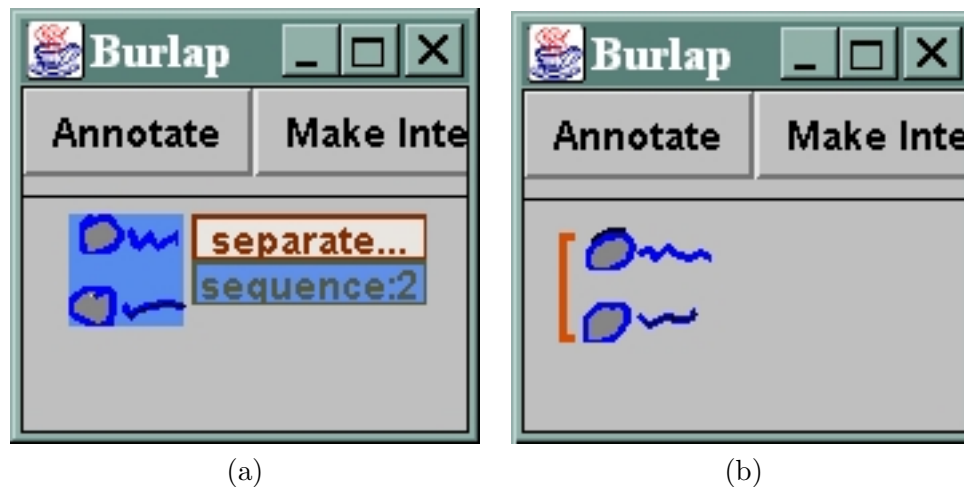
Burlap uses choice mediation techniques for most types of ambiguity. For example, the menu shown in Figure 5-1(b) was created by an interactive mediator that is asking the end user to indicate whether she intended to draw a radio button or a check box. Our goals in designing this and the other choice mediators used in Burlap, seen in Figure 5-1(c), Figure 5-5(a) and Figure 5-5(b), were twofold:

**Table 5-4:** A comparison of three choice mediators in Burlap, illustrating how they vary along the dimensions defined in Section 2.2

Name	Instantiation	Context	Interaction	Layout/Format
Interactor Mediator	Click on ambiguous sketch	Strokes	Select option	
Sequence Mediator	Immediate	Groups	Select option	
Sequence Mediator	Immediate	Groups	Continue/X out	

- We wanted to illustrate that OOPS could handle mediation of a fairly complex task, and that such mediation could be varied along the dimensions defined in our survey. Table 5-4 shows how the 3 choice mediators developed for Burlap differ along the dimensions described in our literature survey (Section 2.2).
- We wanted to show that mediation could be incorporated into the interface of Burlap in a way that did not have to interrupt the end user in her primary task of sketching.

For example, the recognition ambiguity mediator only appears when the end user tries to *interact* with an ambiguous sketch. It does not appear while she is sketching. In another example, the second sequence mediator, shown in Figure 5-5(b), is designed to require no action from the end user most of the time. The recognition result it represents is usually correct, and because of this, it accepts that choice by default if the end user continues sketching. The end user only has to take explicit action if she wants to reject the choice (by ‘X’ing out the mediator). These sorts of design decisions and the impact on the overall end user interaction were not the focus of this thesis, although they have been studied in the past [48], and would make an excellent topic for future work.



**Figure 5-5:** The end user trying to sketch some radio buttons. In (a) the system uses a menu to ask the end user if the buttons should be grouped. In (b) a totally different layout is used for the same purpose.

### 5.3.3 Automatic mediation

Automatic mediation plays a crucial role in Burlap. First, automatic mediators help to decide between multiple uses of the same input. Second, they make sure that issues of timing will not affect how input is recognized. And finally, they help to handle special cases such as annotation mode (a mode selected by pressing the “Annotation” button, in which the user’s strokes are treated as annotations, or comments, about the user interface and not recognized). A quick summary of each of the automatic mediators in Table 5-2 is given below.

**Property Preferable Mediator** This mediator contains a set of properties (which inherit from `filter` (See Source Code 4-1. It immediately accepts events that match any one of its properties. Each property may also specify a set of events to be rejected. Below is a list of the properties used in Burlap:

**Is Annotation** checks stroke events to see if they are marked as annotations (an attribute stored in `user.data` ). If they are annotations, any recognition of the strokes is rejected<sup>3</sup>.

**Stroke Killer** is installed dynamically based on what mode the interface is in. When the end user selects the “Constrain” button, all strokes are rejected because mouse events are only interpreted as selection boxes in that mode. When the user presses the right mouse button in order to select an area for rerecognition, the same situation holds.

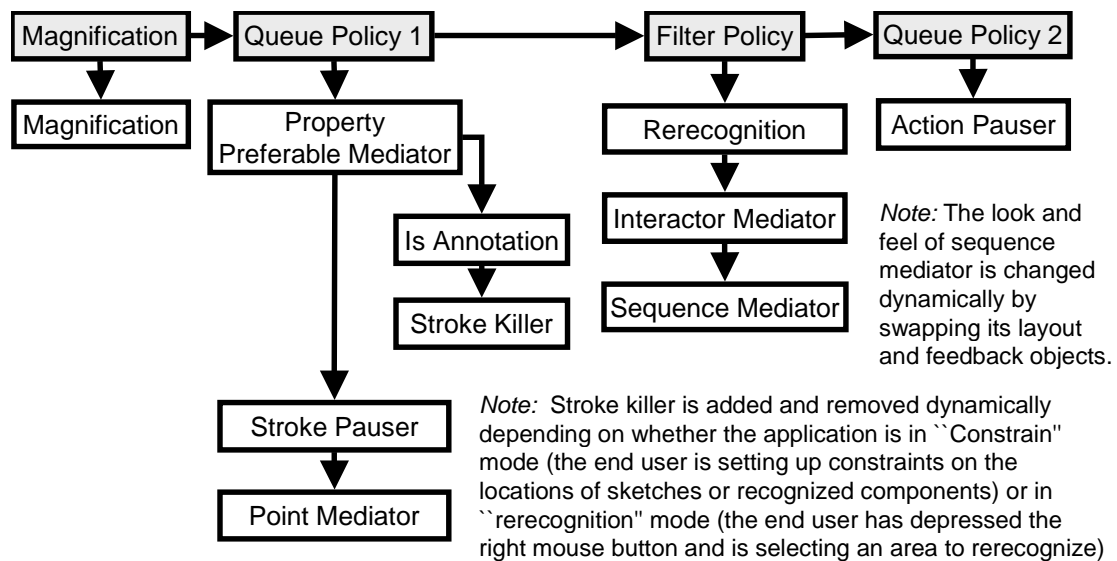
**Stroke Pauser** defers mediation on mouse events until the mouse is released. Remember that each new sensed event is normally mediated before the next event in line is dispatched, so without this mediator, the very first mouse press event (the root of an ambiguous event hierarchy), and each subsequent mouse drag event, would be mediated long before the graffiti recognizer ever received a complete stroke to recognize. When the stroke pauser receives a hierarchy rooted at a mouse event, it defers mediation and adds the hierarchy to a queue. When it receives a mouse release event, it knows that the stroke has been completed and recognized (because all interpretations of that mouse release will be created dispatched before the mouse release is mediated). At this point it allows mediation to continue on each event in the queue.

**Point Mediator** The *Point Mediator* is described in detail in the previous chapter. It identifies “strokes” (sets of mouse clicks) that were really intended to be a single click, and rejects them, leaving the clicks to be handled unambiguously.

**Action Pauser** This mediator defers mediation on an ambiguous sketch until the user interacts with it (causing an action event to be generated). It represents one way of encoding the decision of when certain types of mediation should occur. In Burlap’s

---

<sup>3</sup>This filter could also be installed as a domain filter in the graffiti recognizer. Any strokes that are marked as annotations before they are passed to that recognizer would then be ignored.



**Figure 5-6:** The organization of the meta-mediators (top row, grey background) and mediators (white background) installed in Burlap.

case, this mediator is used to ensure that the recognition ambiguity mediator only appears once the user starts an interaction, rather than as soon as ambiguity is generated. The action pauser defers mediation on all ambiguous event hierarchies that it receives. When an action event arrives, mediation continues and the choice mediator receives an ambiguous event hierarchy containing the ambiguous sketches in which the interaction occurred.

### 5.3.4 Meta-mediation

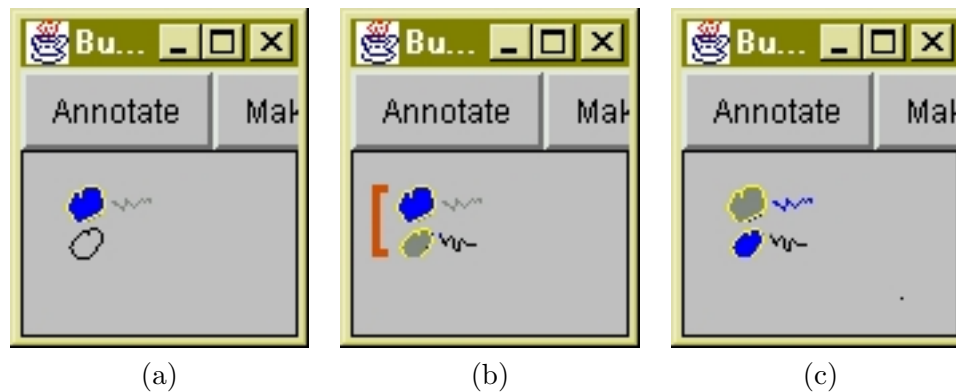
The meta-mediation strategies used in Burlap, and the mediators contained in each of them, are illustrated in Figure 5-6. We will walk through the example interaction in Figure 5-1 (where the user draws a radiobutton, followed by a second radiobutton, in an existing sketched interface) to explain why these particular mediators and meta-mediators were arranged as shown.

**The first stroke** When the user clicks to begin the first stroke of the radiobutton, the

magnification mediator checks to make sure that the mouse down event does not overlap existing sketches that have been recognized as interactive components. The *Property Preferable Mediator* then checks that the newly-begun stroke is not tagged as an annotation. *Stroke Killer* is not installed unless the system is in constrain mode, or the user is indicating an area for rerecognition. Finally, mediation on the mouse down event is deferred by the stroke pauser because one of its interpretations is a stroke. As the user draws the stroke, each of the mouse events is treated like the first, eventually queued up by the stroke pauser. When the stroke is completed, those events are dequeued and the *Point Mediator* checks to see if the stroke should be rejected, because it appears to be a click (it is short and all the events are within a small region of the screen). *Rerecognition* is installed with a filter that looks for interpretations created by the interactor recognizer. Since the stroke is a circle, and there are no valid sketches of interactive components that consist of isolated circles, *Rerecognition* and the next two mediators ignore it and it ends up in the queue of the *Action Pauser*.

**The second stroke** The second stroke is treated identically to the first up until the point when the point mediator checks that it is not a click. However, now that there are two strokes, the interactor recognizer has created several interpretations. At this point, the event graph looks like the one shown in Figure 5-3: there are interpretations to be mediated (such as button, scrollbar, *etc.*). This causes *Rerecognition* to cache the interpretations, and *PASS*, then the interactor mediator (which is installed with a filter that looks for these interpretations) receives the event graph to be mediated. However, the interactor mediator is designed not to interrupt the user while she is sketching, and so it defers mediation.

**The third stroke** This stroke is treated identically to the first. Until the fourth stroke, it will have no connection to the event graph generated by the first two.



**Figure 5-7:** An example of a stroke that is ambiguous. **(a)** Burlap provides feedback about the stroke even though its correct interpretation is unknown. **(b)** The end user draws an additional stroke, causing even more ambiguity. However, Burlap provides feedback that more information is known. **(c)** The end user has mediated the ambiguity.

---

**The fourth stroke** This stroke is treated like the others until just after *Rerecognition*.

However, once this stroke is drawn, a new interpretation is created by the interactor recognizer: An interpretation that groups the two radio buttons as a sequence. The *Sequence Mediator*, which is installed with a filter that looks for sequences, receives the event hierarchy to be mediated, and immediately displays feedback to the end user suggesting how the radio buttons might be grouped. If the end user selects “sequence:2,” all of the ambiguity involving all four strokes will be resolved, and any mediators deferred on portions of that ambiguity will be notified that they should stop mediating. If the end user selects “separate,” the sequence event will be rejected, removing the connection between the two sketched buttons, and mediation will continue. If and when the end user clicks on one of the buttons, causing an action event to fire off, the sketch mediator will know that it is time to mediate that sketch and display the menu of choices (radio button or check box).

## 5.4 How Burlap was Created

Overall, the design of Burlap benefited significantly from both our survey and the OOPS toolkit. For example, because the toolkit maintains information about ambiguity and separates feedback from action, Burlap can provide feedback to end users about how their strokes are being interpreted without throwing away ambiguous information that may help later to translate strokes into interactive components correctly. Figure 5-7 demonstrates an example of this. In **(a)**, the user has sketched a stroke that cannot yet be interpreted as an interactive component. Still, feedback about the stroke (the bottom circle) is provided even though recognition cannot yet occur. After the user draws another stroke **(b)**, recognition can occur, and the system changes the color and shading of the strokes to indicate that they have been interpreted. It also adds a new line indicating that the two sketched components are intended to be linked. In **(c)**, the user begins to interact with the button on the bottom, and thus implicitly accepted the interpretation that the sketched components are linked. The mediator is gone and the selection (the dark circle) has switched from the top to the bottom button.

In general, the OOPS toolkit allows Burlap to delay mediation as long as appropriate, and then to choose the best style of interactive mediation once mediation is necessary. Since the main task of a user of Burlap is *sketching* a user interface, he may never need to mediate some of his sketches, and Burlap will never require him to do so. If and when he wishes to interact with the sketches or convert the sketch to a “real” user interface, Burlap will ask him to mediate where necessary.

Burlap also illustrates how multiple recognizers can be combined in OOPS. Burlap makes use of two recognizers, and pen input may also be used as an annotation (unrecognized), to draw constraints, or, as input to mediators, sketches, or standard interactors. OOPS helps to determine to which of these uses pen input should go, in any given instance. In the case of the recognizers, order is important (one recognizer interprets the results of the



previous one), but they are not in conflict with each other. Each of the other uses must do conflict with each other. For example, if the end user has clicked on a sketched component (to select it, for example), that input is never also interpreted as a new sketch. Similarly, if the user is interacting with a mediator, the input produced will not show up on screen as an annotation.

Several different mechanisms are involved in deciding which interpretation an input event has. For example, some things (sketches, interactors, mediators), only receive input when it is spatially located inside them. Others (constraints, annotation) only receive input when the user has put the system in a certain mode (by clicking on the appropriate button). Finally, the two recognizers almost always receive pen input (unless a sketch or interactor has grabbed and used it), and it is the role of mediators to reject the recognition results when appropriate.

In summary, OOPS allowed us to recreate a fairly complex application (Burlap). Although Burlap is not a complete version of SILK, it does things SILK does not with respect to recognition ambiguity, thanks to OOPS' flexible support for mediation.

## Chapter 6

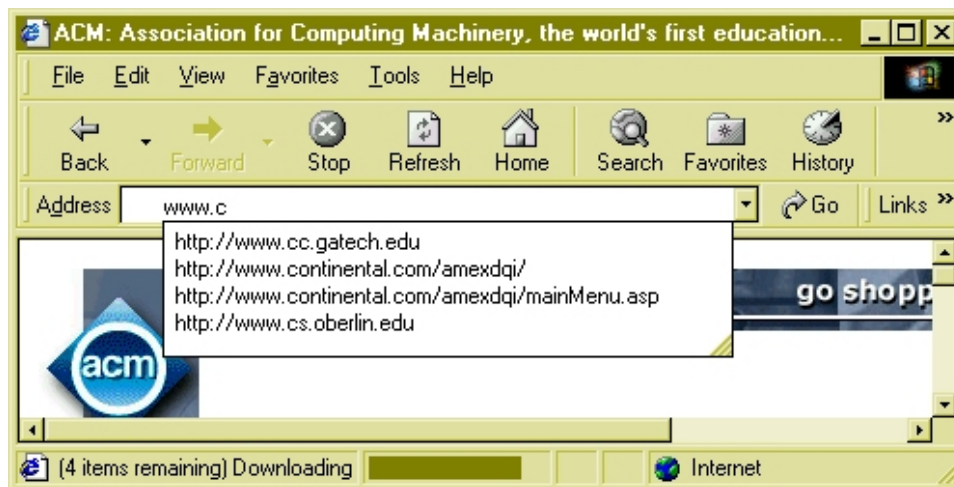
# SOLVING NEW PROBLEMS: ADDED BENEFITS

The previous chapter showed that the our architecture allows us to reproduce existing state of the art applications (in particular, Burlap). This chapter describes four new interactive mediation techniques. Each of these techniques is designed to illustrate a method for overcoming a problem uncovered in our survey of existing mediation techniques. In each section, after discussing the identified problem area, and a new mediation technique that addresses it, specific toolkit mechanisms necessary to support these solutions will be considered. Briefly, the areas addressed are incorrect alternatives in choice mediators (identified as a problem in Chapter 2, p. 27), choice mediators that occlude other screen elements (identified as a problem in Chapter 2 on p. 27), target ambiguity (first introduced in Section 1.1.3 and identified as a problem area in Chapter 1, p. 9), and rejection errors (first introduced in Section 1.1.2, and identified as a problem area in Chapter 1, p. 9).

### 6.1 Adding Alternatives

#### 6.1.1 Problem

One problem with choice mediators is that they only let the user select from a fixed set of choices (See Section 2.2.2 for evidence of this). If no choice is right, the mediator is effectively useless. Normally, the user switches to a repetition technique at this point, ideally one that lets her precisely specify the input she intended. For example, she may switch to a soft

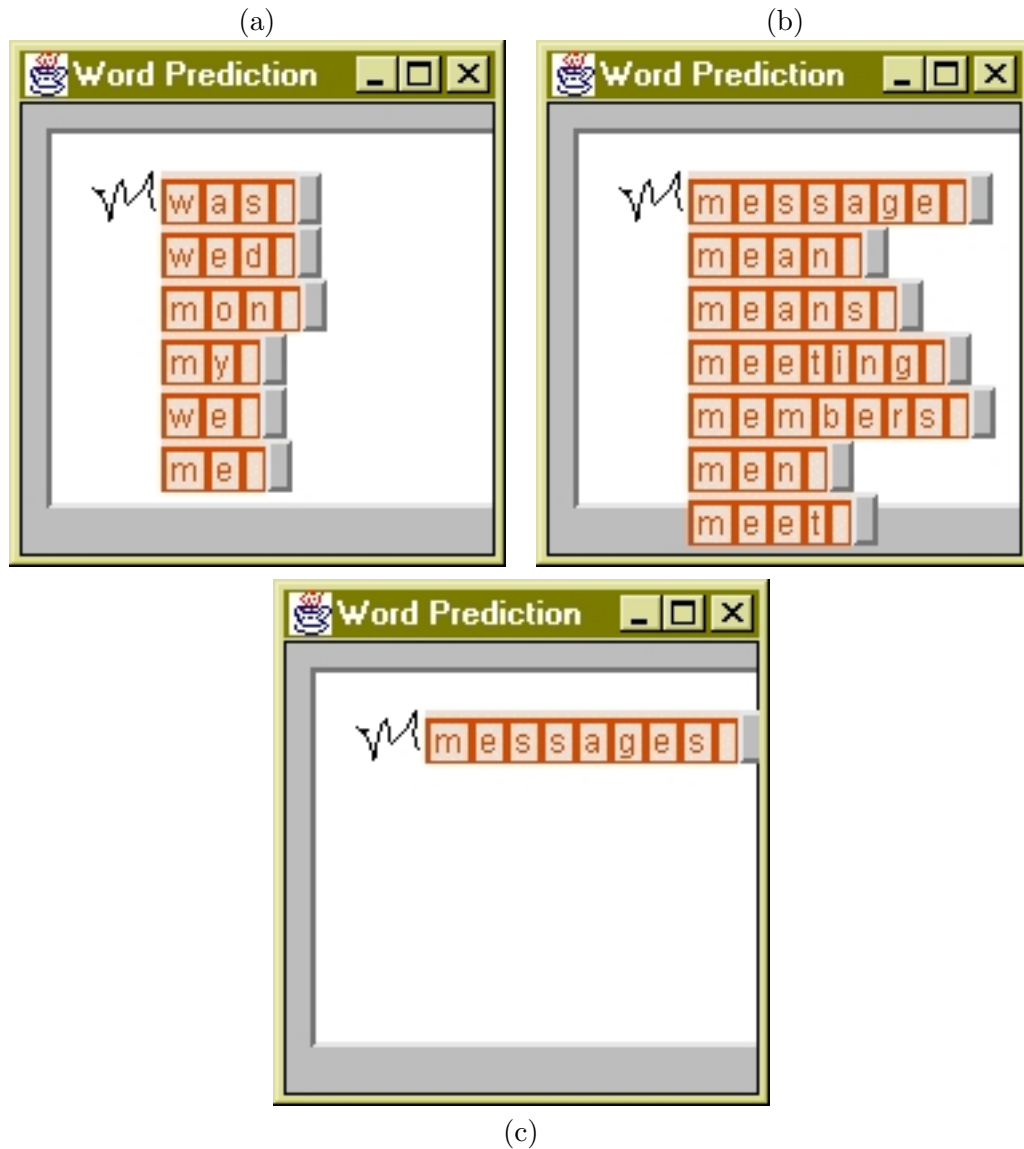


**Figure 6-1:** A menu of predicted URLs in Internet Explorer™

keyboard if her handwritten text is misrecognized. The problem with making this switch is that it may impose cognitive load [21].

Our goal is to allow the user to specify new interpretations, as well as to select from existing ones, using the same mediator. We can support a smooth transition from selection of an existing choice to specification of a new one by extending an  $n$ -best list to include some support for repetition. As an example, we created a mediator that fits these requirements for the word prediction application described in detail at the end of Chapter 4.

Word prediction is an extremely error-prone form of input in which a recognizer tries to predict what text the user is entering. Originally created to help disabled users in dictation tasks [42, 5, 26, 54, 55, 57], it has since been applied successfully to other domains, such as Japanese character entry [53]. In general, due to a great deal of ambiguity, word-prediction of English can typically only support input speeds of less than 15 words per minute (wpm) [5]. However, accuracy increases greatly with word length or limited input grammars. Thus, domains with long words (such a URL entry) and constrained grammars (such as programming) have benefited from word prediction. Examples include Internet Explorer™(Figure 6-1), Netscape™, and Microsoft Visual Studio™.



**Figure 6-2:** A choice mediator which supports specification. The user is writing ‘messages’. (a) The user sketches a letter ‘m’ which is interpreted as either a ‘m’ or a ‘w’ by a character recognizer. A word predictor then generates options for both letters. The user clicks on the ‘e’ in ‘me.’ (b) This causes the mediator to filter out all words that do not begin with ‘me.’ The word ‘message’ is now the top choice, but it needs to be pluralized. The user clicks on the space after ‘message’ indicating that the mediator should generate a word beginning with ‘message’ but one character longer. (c) The resulting word.

### 6.1.2 Solution

Figure 6-2 shows an example of the mediator we developed. In our application, the user can sketch Graffiti<sup>TM</sup> letters, which are recognized as characters (by [47]). A word predictor then recognizes the characters as words. When the graffiti letters are ambiguous, the word-predictor returns words starting with each possible letter. Once mediation is completed, the text edit window is passed the correct choice. When the user begins a word, there are hundreds of possibilities. Only as she enters additional letters do the possibilities shrink. It is not possible for the mediator to display all of these choices. Our goals in this mediator are:

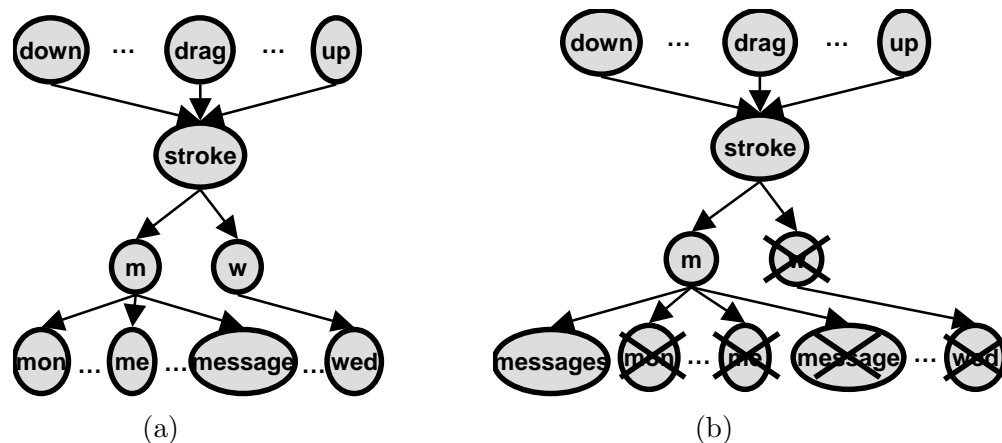
**Provide choice at the word level:** The user can select from among choices as he did in a standard choice mediator, by clicking on the grey button to the right of a word.

**Allow users to specify length:** By clicking on the space at the end of a word, the user causes the mediator to add a character to that word.


**Allow users to specify individual characters:** The user can right-click on a character to cycle through other possible characters. This can be used to generate words not returned by the word-predictor.

**Allow users to control filtering:** By clicking on a character, the user specifies a prefix. The mediator reflects this by displaying only words starting with the same prefix. Although word-predictors support dynamic filtering, in most cases a prefix can only be specified by entering each letter in the prefix in turn. If the user filters repeatedly on the same prefix, the mediator will display a new set of words each time. If the user filters on a new prefix (for example by clicking on the first or second letter of a previously specified prefix), the system will be updated to reflect this.

**Allow users an escape:** If the user sketches a new letter, only the current prefix (the last prefix selected during filtering) will be accepted.



**Figure 6-3:** (a) The original event hierarchy in Figure 6-2 (a-c) and (b) the final hierarchy after mediation, with the new word added.

Suppose the user enters an  (for which the graffiti recognizer returns ‘m’ and ‘w’). The word predictor returns words starting with ‘m’ and ‘w’ (derived from a frequency analysis of a corpus of e-mail messages), of which the mediator displays the top choices: was, wed, mon, ... (Figure 6-2(a)). The user, who intended ‘messages’, filters for words starting with ‘me’ by clicking on the ‘e’ in ‘me.’ The resulting list (Figure 6-2(b)) includes ‘message’, but not ‘messages.’ The user indicates that a word one character longer than ‘messages’ is needed by clicking on the space at the end of the word, and ‘messages’ appears as the top choice (Figure 6-2(c)). The user selects this choice by clicking on the grey button to its right.

### 6.1.3 Toolkit support for adding alternatives

This example dynamically generates new interpretations during mediation. In the example of Figure 6-2, a new interpretation (‘messages’) is created. All of the other interpretations are rejected. Figure 6-3 shows the original (a) and changed (b) hierarchy. The new event (bottom left of (b)) is accepted immediately, since the user just specified that it was correct using the mediator. It is then dispatched, with the result that it is consumed by the text

edit window. No further mediation is necessary since neither the new event, nor the event hierarchy it is added to, is ambiguous.

This example was created by subclassing the default choice mediator included with OOPS. Details on how this was done can be found in Section 4.5.2.

#### **6.1.4 Reusability**

This mediator can be used with any recognizer that returns text, including speech recognition or handwriting recognition, since the alternatives generated by these recognizers tend to have some similarities to the intended word. For example, some of the characters in a written word may be recognized correctly while others are not. The general idea, adding interactivity supporting repetition to a selection task, can be applied to other domains as well. For example, the automatic beautifier (Pegasus) uses a choice mediator to display multiple lines [37]. Repetition could be added to this by allowing the user to drag the end point of a line around. This would also allow Pegasus to display fewer alternatives in cases where too many are generated, in which case the line could snap to hidden interpretations.

#### **6.1.5 Analysis**

Is this mediator really useful? Without word prediction, the user would have sketched eight characters. Given an 85% accuracy rate (typical for many recognizers), she would have to correct at least one letter (with a total of at least nine strokes). Here, the user has sketched one letter and clicked three times. Also note that as the Graffiti<sup>TM</sup> recognizer becomes more inaccurate (we add new letters to the set of potential alternatives), the number of required strokes to enter a word increases linearly with the number of possible characters. In contrast the number of necessary strokes without word prediction, and with repetition-based mediation, increases exponentially. Of course, a complete analysis would be needed in order to accurately determine the relative speeds of the two types of input. Only one data point, it should be noted that the mediator is built with appropriate defaults

so that if the user simply ignores it and sketches the eight characters, the result will be identical to a situation without word-prediction. The user may use repetition as if the mediator were not present, or select the correct character from the first letter of each word (mediating ambiguity from the graffiti recognizer) and ignore the words generated by the word predictor.

## 6.2 Occlusion in Choice Mediators

### 6.2.1 Problem

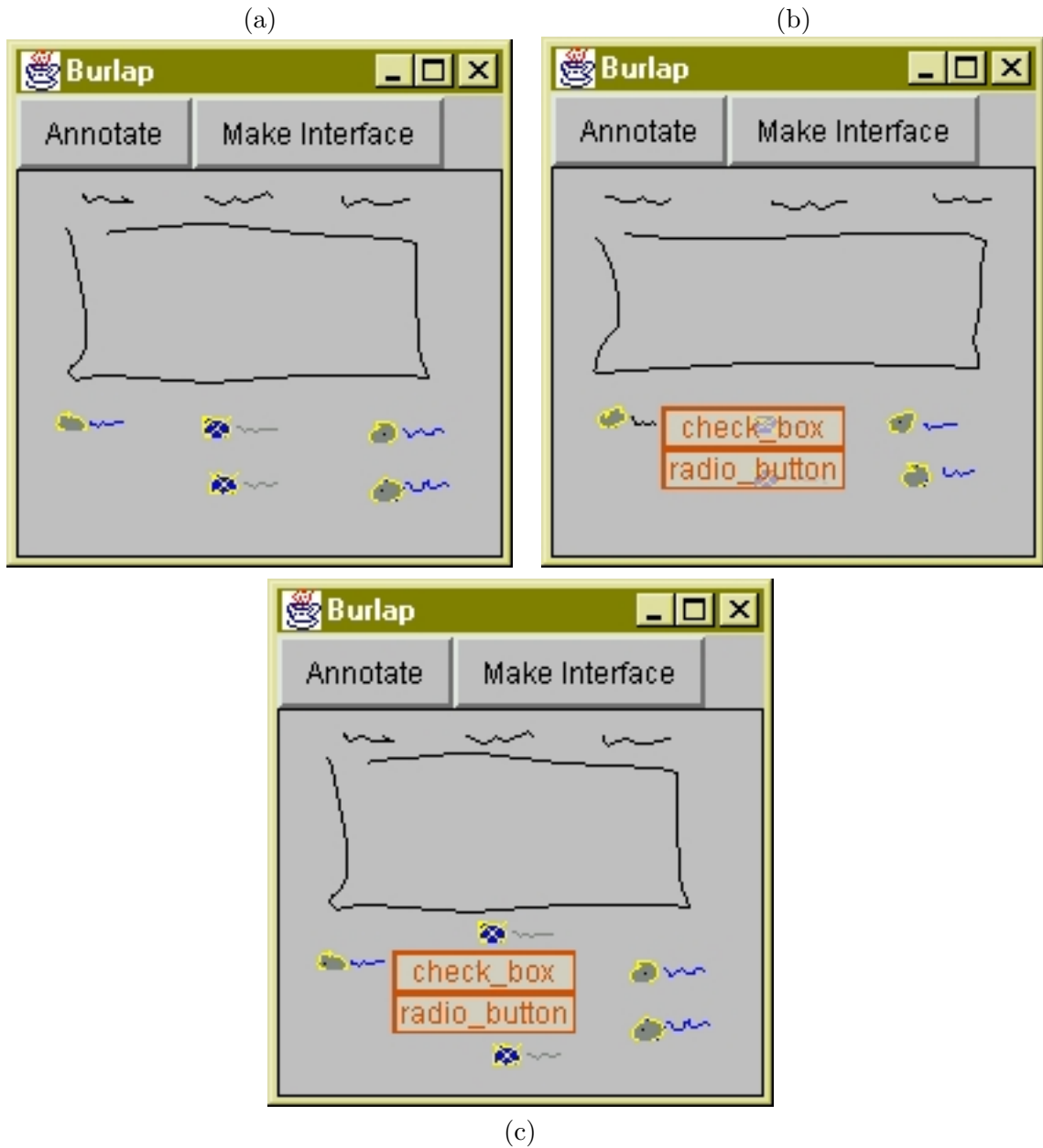
Choice mediators generally display several possible interpretations on the screen from which the user to selects. They are fairly large, and may obscure important information needed by the user to select the correct choice. Since they are also temporary, it does not make sense to leave screen space open just for them. An alternative is to dynamically make space for them. This approach was inspired by the work on fluid negotiation of Chang *et. al* [15]. They developed a series of techniques for making space for temporary displays in crowded interfaces. Some of the possible approaches they suggest include shrinking, fading, and call-outs. The temporary display then fluidly (dynamically) negotiates the best approach with the underlying document.

For example, consider Burlap, the application shown in Figure 6-4. The n-best list in Figure 6-4(b) is obscuring two buttons. Is the leftmost sketch a check box or a radio button? This type of ambiguity is not mediated in Burlap until the user tries to interact with a button. He may have drawn the hidden buttons some time ago. In order to be consistent, the user may need to see the buttons now to determine their status.

### 6.2.2 Solution

Our solution moves the sketches occluded by the mediator into a more visible location (Figure 6-4(c)). This approach is based on one specific technique from the set of fluid





**Figure 6-4:** An example of fluid negotiation to position a mediator in the Burlap application. (a) The user needs to mediate whether the object on the bottom left edge is a check box or radio button. (b) This mediator (an n-best list) occludes some of the sketched interactors. (c) This mediator repositions all interactors that intersect with it so as to remove any occlusion.

negotiation techniques developed by Chang *et al.* [15]. Because our mediator is used for input as well as output, and is the focus of the interaction, we chose a technique that only changes the underlying document (the sketched interface), not the size or shape of the mediator.

The mediator is based on a lens that uses the subArctic interactor tree to pick out all of the interactors that intersect its bounding box. It works by traversing the interactor hierarchy of the user interface in order to determine which interactors it overlaps. It then calculates the closest non-overlapping free space for each interactor, and causes the interactors to draw themselves in that free space. This is done by taking advantage of subArctic's ability to support systematic output modification [22], and does not require any changes to the interactors or the application.

### **6.2.3 Toolkit support for dealing with occlusion**

Occlusion is handled in a way that requires no changes to the underlying application. The only difference between the application shown in Figure 6-4(b) and (c) is the kind of mediator installed. The ability to swap mediators without modification to application or recognizer comes from the separation of concerns provided by OOPS.

The specifics of the interaction technique are made possible by subArctic's support for systematic output modification, rather than OOPS. However, if OOPS did not support mediation and recognition in a fashion that is completely integrated with subArctic's regular support for input and interactors, the mediator would be harder to implement.

### **6.2.4 Reusability**

The occlusion mediator can be used wherever an *n*-best list might be used, including word-prediction, speech and handwriting recognition in GUI settings, and the Remembrance Agent [71] (See Table 2-2). More generally, similarly to the fluid negotiation work, the lens responsible for moving screen elements out from under the mediator could be combined

with any mediator that is displayed on screen in a GUI. Of course, there are some situations where occlusion is appropriate and the mediator is not necessary, such as in the magnifier described in Section 6.3.


### 6.2.5 Analysis

Although we chose to move away interactors in the underlying documents, any of the other approaches suggested by Chang *et al.* would also be appropriate and should be explored. In the future, we would like to provide a complete implementation, including fluid negotiation for deciding which technique to use.

This mediator is modal, which means that the user is required to resolve the ambiguity before continuing his interaction. This limitation exists because our implementation for moving the interactors currently only handles output and not input.

## 6.3 Target Ambiguity

### 6.3.1 Problem

We have encountered three major classes of ambiguity in our work. These are recognition ambiguity (was that “sewage” or “assuage” or “adage”?); segmentation ambiguity (was that “sew age” or “sewage”?) and target ambiguity. Target ambiguity arises when the target of the user’s input is uncertain. For example, it is unclear if the circle around the word  is intended to include “is” or not.

We first introduced and described these classes of ambiguity in detail in Section 1.1.3. Even though target and segmentation ambiguity may also lead to errors, mediation has only addressed recognition ambiguity in almost all previous systems. Of the interface techniques described in our survey in Chapter 2, none dealt directly with segmentation ambiguity. In one case, researchers draw lines on the screen to encourage the user to segment his input appropriately, but no dynamic feedback about segmentation is given [27]. In [51], we

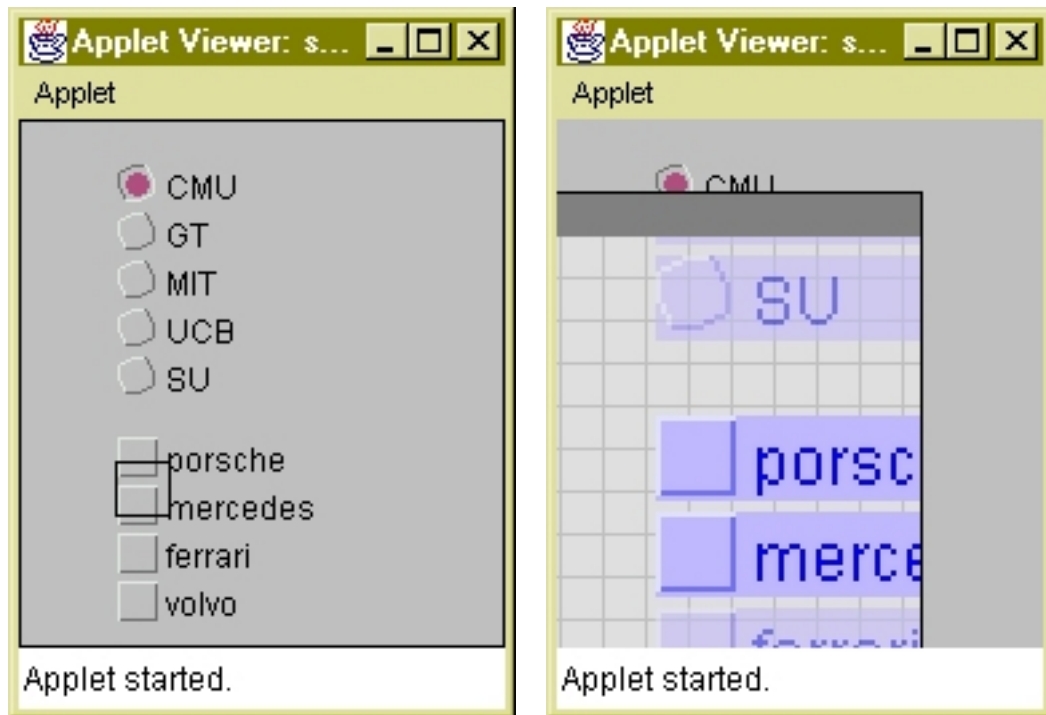
introduced a mediator of segmentation ambiguity. (See Figure 5-5, p. 120, for an example of this mediator in use). As with segmentation ambiguity, almost none of the systems in our survey dealt with target ambiguity. Where it existed, it was commonly tied directly in with recognition ambiguity. Here we demonstrate mediation of target ambiguity.

We are interested in using target ambiguity to model situations that, although they may seem ambiguous to the user, are commonly treated as straightforward by the computer. In particular, consider situations where mouse motion becomes difficult. The term “fat finger syndrome” is used to refer to the problems that arise when the user’s finger is larger than the button he wants to press (*e.g.* very small cell phones, touch screens). In addition, miscalibration on something like a digital white board can cause a mouse click to go to the wrong interactor. Also, certain disabilities and advanced age may make it difficult to control a mouse. Research shows that older users have trouble selecting common GUI targets [85] as do people with disabilities, such as cerebral palsy [40].

### 6.3.2 Solution

These problems can be addressed by treating the mouse as an ambiguous point, for example represented as an area. However, the resulting area may overlap more than one interactor (an example of target ambiguity). Both Worden and Buxton dealt with this by simply treating the mouse as a point in those circumstances, thus eliminating the ambiguity [85, 39].

In Figure 6-5, we mediate this kind of target ambiguity using a magnifier. Figure 6-5(a) shows the area associated with the mouse as a black box. The magnifier only appears when there is a need due to ambiguity. In this example, the box overlaps both the “Porsche” and “Mercedes” check boxes, so ambiguity is present. The resulting magnifier is shown in Figure 6-5(b). For context, we include an area four times the size of the mouse area. The magnified area is interactive and a user can click on interactors inside it just as he would on an unmagnified portion of the screen. As soon as the user completes his action, or the mouse leaves the magnifier, the magnified area disappears.



(a)

(b)

**Figure 6-5:** An example of mediation of ambiguous clicking with a magnifier. (a) The user is trying to click on a button. Which one? (b) A mediator that magnifies the area lets the user specify this (The ambiguous choices are displayed in darker colors)

### 6.3.3 Toolkit support for target ambiguity

Conceptually, target ambiguity arises when user input can go to multiple targets. In many cases, each target generates alternatives. Here a single recognizer generates alternatives for each target. This is because the area mouse is intended to function with interfaces that include old-style ambiguity-blind interactors, that may not receive or create ambiguous input and therefore cannot generate target ambiguity on their own.

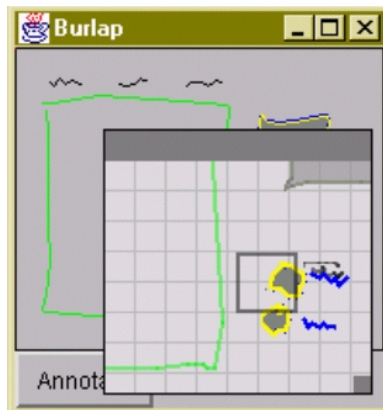
First, target ambiguity is generated by a recognizer that checks for multiple mouse targets within the mouse area (shown as a box in Figure 6-5). If only one target exists, the input is processed normally. If several targets exist, the results are passed to the mediator.

It is because of our general support of recognition that this is possible. For example, when the extended mouse area (but not the mouse itself) intersects a single interactor, this recognizer creates a new mouse event over that interactor as an interpretation of the raw mouse event it gets as input. This interpretation is dispatched and consumed by the interactor, which does not even know that a recognizer was involved. As far as the interactor is concerned, the user clicked on it.

Our mediator makes use of a lens that magnifies the area under the input [22]. In addition, the lens is responsible for adjusting any positional input it gets based on the new size and position of the pixels it is magnifying.

### 6.3.4 Reusability

The magnification mediator works with any interface built in OOPS. This includes animated (moving) interactors; Burlap (See Figure 6-6); and any other interface that uses mouse clicks. It is limited to mouse press and release events. There are some difficult issues that arise in the case of mouse drags and other mouse input. In theory it could be generalized to any positional input, not just mouse input.



**Figure 6-6:** An example of the magnification mediator being used in Burlap

---

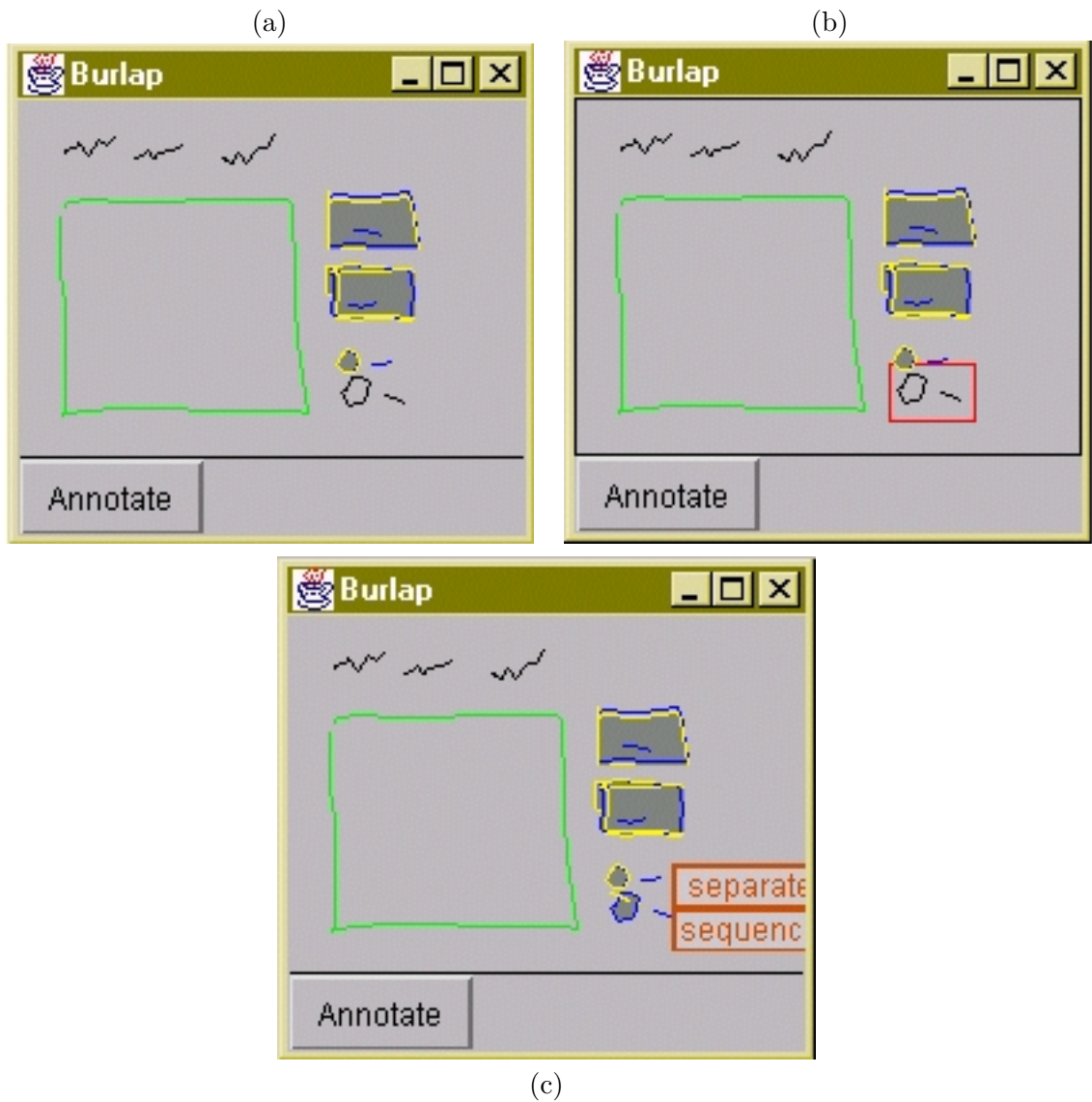
### 6.3.5 Analysis

As mentioned above, our design is intended to be an extension of previous work done by Buxton [39] and by Worden [85]. By using an automatic mediator that embodied the policies in Buxton's or Worden *et al.*'s work, we could create an identical interaction. For example, an automatic mediator that simply rejected both interpretations when the area overlaps multiple interactors would mimic Worden *et al.*'s policy: in those cases, mouse input would be delivered as if the mouse were an unambiguous point. However, our intent was to show that these automatic policies could be replaced by interactive mediation. Although we succeeded in this intent, additional evaluation would be necessary to show that our approach was an improvement and not simply more confusing or difficult to use.

## 6.4 Rejection Errors

### 6.4.1 Problem

The last problem area addressed in this chapter is rejection errors. Rejection errors occur when some or all of the user's input is not interpreted at all by the recognizer (See Section 1.1.2 for more details on this problem). For example, in Figure 6-7, the user has



**Figure 6-7:** An example of repetition through rerecognition. (a) An interface sketched in Burlap. In the lower right is an unrecognized radio button. (b) The user has selected the radio button, indicating that rerecognition should occur. (c) Rerecognition is completed, and the result is further ambiguity. One of the possible interpretations generated was a radio button. If that is correct, should the new button be in sequence with the old one, or should it be separate?



sketched a radio button in the lower right, but those strokes have no associated interpretation: they were not recognized. Remember that we define error from the end user's perspective, and she expected those two strokes to be grouped and recognized as a radio button. Thus, a rejection error has occurred.

The rules used for recognition in Burlap are taken from [46], and are based upon the shape and size of individual strokes drawn by the user and the graphical relationships between sets of strokes (such as distance, orientation, *etc.*). When the user draws something too big, too far apart, or so on, it is not recognized. In general, the causes of rejection errors are very recognizer-dependent. As an example, a rejection error may occur in speech recognition because the user speaks too quietly.

#### **6.4.2 Solution**

One solution to this problem is unmediated repetition. The user can simply try again. However, research in both pen [24] and speech [25] recognition has found that rerecognition accuracy rates are no better than recognition accuracy rates.

We address this with a mediator that supports guided rerecognition. The user can initiate rerecognition by selecting the strokes that should have been recognized using the right mouse button, as in Figure 6-7(b). By doing this, the user is giving the mediator important new information. In Burlap's case, the new information is that the selected strokes should be grouped (segmentation information), and interpreted as an interactor. The mediator passes this information on to the recognizer, which can use it to eliminate options, such as interactors that have a different number of strokes.

#### **6.4.3 Toolkit support for guided rerecognition**

In this example, the user is providing segmentation information to the recognizer with the help of the mediator. Although the unistroke gesture recognizer used in Burlap (a third party recognizer [47]) does not support guided rerecognition, the interactor recognizer,

which we built, does. When the user selects some strokes, the mediator passes the selected strokes to the interactor recognizer, using the `resegment(Set, Object)` method described in Chapter 3, p. 68. The recognizer generates a new set of interpretations based on those strokes. Because the recognizer now knows the number of strokes, it can quickly narrow the possible interactors and generate alternatives.

Since the recognizer generates new events during mediation, those events must be dispatched, potentially resulting in further interpretations. These new ambiguous events will be mediated by OOPS (Figure 6-7(c)), which tells any currently visible mediators to update themselves to show the new interpretations. Any events that were already accepted or rejected in the hierarchy remain that way.

#### 6.4.4 Reusability

The same mediator could be applied in other graphical settings with segmentation issues, such as handwriting recognition. This mediator must be told which recognizer it should work with (in the case of Figure 6-7, Burlap's interactor recognizer). Normally, it will automatically cache any events interpreted by that recognizer. Alternatively, it may be given a filter that knows enough about the domain to cache the correct events. In this example, we use a filter to cache stroke events (necessary because the interactor recognizer is part of a two-phase recognition process involving another recognizer that does not support guided rerecognition). Once the user specifies an area, any cached events inside that area are sent to appropriate recognizer to be rerecognized.

Note that this mediator works not only with rejection errors but also substitution errors. For example, when the user's input is misrecognized and none of the suggested options are correct, new options may be generated by the technique described above.

### 6.4.5 Analysis

Anecdotally, the approach we have described works so well that there is generally no need for the ability to delete misrecognized interactors in Burlap. The support for substitution errors allows misrecognized interactors to be converted to the correct interpretation.

The biggest flaw in the mediator is that selection is done modally by pressing the right mouse button. However, although this may be unintuitive, it helps to guarantee that rerecognition events will not be confused with new strokes intended to be recognized on their own. Many existing systems use this technique for the same reasons [68, 46].

## 6.5 Summary

The work described in this chapter addresses a variety of problems from several domains involving recognition, taken from some of the “red-flag” issues raised by our background work. All of the mediators presented here were enabled by the OOPS toolkit. They were built with the intent to be re-used in many situations.

Although these techniques all fall within the broad domain of pen input, we feel that, like other mediators, they may also apply in other domains. As an example, consider a speech only phone-based system.

**Adding Alternatives:** This problem area represents a tradeoff between repetition and choice. This tradeoff is critical in speech-based applications for two reasons. One, it takes time to present lots of choices, often too much time from the user’s perspective. Two, when users repeat their input, it tends to be harder to recognize. Thus, providing a mediator that allows the user to request additional choices without repeating her input would be a useful compromise. Although this solution would take a very different form in the audio world, the same techniques of filtering and specification apply.

**Occlusion:** This problem area is basically a visual issue, as is our solution. However, in certain domains, it might be necessary to mediate while the user is simultaneously

listening to other audio (for example if the user is asking questions about a news broadcast). In these cases, the mediator can easily be brought to the foreground. More interesting is the question of how the other audio should be moved to the background without being lost completely. Studies have shown that pitch is one key factor in allowing us to track multiple voices (the “cocktail party” effect) [10]. Alternatively, spacialized audio could be used to create exactly the effect that we create visually above.

**Target Ambiguity** As with pen input, situations may arise in speech where the user’s input is well understood, but the target is not known. In fact, the classic example of target ambiguity comes from speech in a multimodal setting: Bolt’s “Put That There” example [8, 84]. Word’s like “that” and “there” may refer to multiple targets. The equivalent of our magnification mediator in an audio-only setting would be a mediator that determined the set of valid targets based on other context, and then dynamically create a menu that contains only those “targets.”

**Rejection errors** Rejection errors may occur in noisy environments such as mobile speech-based applications (errors of all types skyrocket in noisy environments [28]). Raw audio is difficult for users to navigate, and reselecting previously spoken audio would be extremely difficult since the user would have to somehow find and specify where that audio fell in a continuous stream of input. However, in multi-modal settings, this approach is more feasible. Oviatt showed that even in noisy settings, mutual disambiguation of spoken and pen-based input helps to reduce the impact of substitution errors in speech recognition considerably [67]. In cases where no audio is “heard” at all (rejection errors), it might be possible to grab the audio stream around a significant pen event and apply more aggressive noise reduction algorithms to extract recognizable speech.

The techniques presented in this Chapter show that OOPS makes it possible to build a variety of techniques that go beyond the current state of the art in correction strategies. In the process of doing this, we have created a uniquely varied library of re-usable mediators that go beyond what is available with other toolkits, an important contribution in and of its own right.

## Chapter 7

# GENERALIZING THE ARCHITECTURE: A NEW DOMAIN

The goal of this chapter is to address questions about the generality of the architecture described in Chapter 3. In this chapter, we show that the architecture illustrated by OOPS in the previous two chapters does not require the subArctic toolkit and is not limited to pen or speech input. Although the first instance of that architecture, OOPS, was implemented on top of the subArctic toolkit, and has been mostly used to demonstrate mediation techniques supporting pen input, we believe that the architecture is applicable to a broad range of systems and for a broad range of input types. In order to demonstrate this, we have combined our architecture with that of the Context Toolkit [75, 18, 20], to allow humans in aware environments to mediate errors in sensed information about them and their intentions. The context toolkit supports the development of context-aware applications, applications that make use of sensor data and other implicitly gathered context to tailor their behavior to the user's context.

We begin by showing why mediation and ambiguity are pertinent to the area of context aware computing. The following section explains how we were able to modify the Context Toolkit to support our notion of ambiguity and mediation (creating CT-OOPS). We then discuss how mediation must be handled in a context-aware setting. We end with an example of an application built using CT-OOPS. We describe the application and show how it was built and how an interaction involving mediation takes place in that application. The description of CT-OOPS concludes with a discussion of some additional issues that should

be addressed in future work. The last section of the chapter is devoted to a discussion of how the architecture presented in this thesis could be combined with other toolkits besides subArctic and the Context Toolkit.

## 7.1 Motivation for Mediation in Context-Aware Computing

A context-aware environment senses information about its occupants and their activities, and reacts appropriately to this *context*, by providing context-aware services that facilitate everyday actions for the occupants [18]. Researchers have been building tools and architectures to facilitate the creation of these context-aware services since about 1994 [76, 19, 13, 63, 77], by providing ways to acquire, represent and distribute sensed data more easily (See [18] for an overview of these architectures). We added the ability to deal with ambiguous context, in an attempt to make context-aware applications more realistic. Part of addressing this realism is dealing with the inherent imperfections of sensors.

Imperfectly sensed context produces errors similar to recognition-based interfaces, but there are additional challenges that arise from the inherent mobility of humans in aware environments. Specifically, since users are likely to be mobile in an aware environment, the interactions necessary to alert them to possible context errors (from sensing or the interpretation of sensed information) and to allow for the smooth correction of those errors must occur over some time frame and over some physical space. Additionally, context is sensed implicitly without user involvement. Context may not even be used by the user about whom it is sensed. Thus, errors may be “corrected” by a user who does not know what the correct interpretation is. Another consequence of implicit sensing is that the user will not be involved in catching errors early or trying to structure input in a way that is easy to interpret. This means that more errors are likely to arise than in the more controlled settings typically used in desktop recognition applications.

## 7.2 Combining the Toolkits

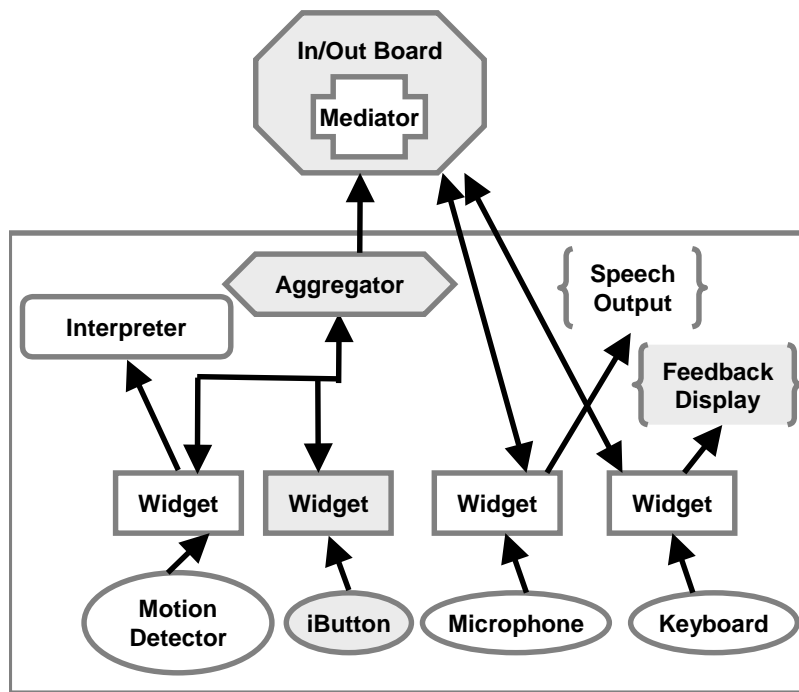
In order to explore mediation in the realm of context-aware computing, we chose to re-implement the architecture presented in this thesis on top of an existing context-aware computing toolkit, the Context Toolkit [75, 18, 20]. The Context Toolkit is a platform for building context-aware applications, and as such it solves many difficult problems relating to the acquisition, distribution, and use of context. However, it does not address the problem of imperfect recognition, for which the architecture presented in this thesis is well suited. We have combined the two architectures to explore the uses of ambiguity and mediation in context-aware computing.

The next section describes the Context Toolkit [18]. Once we have introduced the Context Toolkit, we go on to show how we were able to modify the Context Toolkit to incorporate support for ambiguity and mediation, by incorporating the architecture that was developed for this thesis [19]. As in the case of subArctic, our main modifications involved adding ambiguity to those aspects of the context toolkit that transmit and interpret information about sensed data (the event dispatching portion of the toolkit). Our other main additions are a mediation subsystem and a small number of example mediators.

### 7.2.1 The Context Toolkit

The Context Toolkit is a software toolkit that allows others to build services that support mobile users in aware environments. It allows the addition of context or implicit input to existing applications that do not use context. The Context Toolkit consists of three basic building blocks: *context widgets*, *context aggregators* and *context interpreters*. Figure 7-1 shows the relationship between sample context components in a demonstration application, the In/Out Board [75]. The In/Out Board is an application designed to monitor who is currently at home for use by residents and other authorized people. It is important that





**Figure 7-1:** Example Context Toolkit components. Arrows indicate data flow. This example is taken from the In/Out Board application, but it can easily be generalized to an arbitrary number of applications (outside the large rectangle), aggregators, widgets, sensors (the ovals at the bottom), widgets, interpreters, and output capabilities. The mediator (inside the In/Out Board application) is a component that is only found in CT-OOPS.

context gathered by this application be accurate, since it may also be used by other applications that need information about people’s whereabouts. The In/Out Board has been in use for about 4 years without support for ambiguity. It is a typical context-aware application. In our experiments, we installed an ambiguity-aware version of the In/Out Board in the Georgia Tech Broadband Residential Laboratory, Georgia Institute of Technology’s context-aware home [41].

Sensors used in the In/Out Board include a motion sensor, identity sensor (iButton), microphone, and keyboard. Context widgets encapsulate information produced by a single sensor, about a single piece of context. They provide a uniform interface to components or applications that use the context, hiding the details of the underlying context-sensing

mechanism(s) and allowing them to treat implicit and explicit input in the same manner. Context widgets allow the use of heterogeneous sensors that sense redundant input, regardless of whether that input is implicit or explicit. Widgets maintain a persistent record of all the context they sense. They allow other components to both poll and subscribe to the context information they maintain.

A context interpreter is used to abstract or interpret context. For example, a context widget provides action to motion data, but the interpreter converts this to identity and direction (in or out) based on historical information about when people tend to enter and leave the house.

A context aggregator is very similar to a widget, in that it supports the same set of features as a widget. The difference is that an aggregator collects multiple pieces of context. In this case, the aggregator is collecting context about identity from the iButton and motion detector widgets. An aggregator is often responsible for the entire context about a particular entity (person, place, or object).

Context components are intended to be persistent, running 24 hours a day, 7 days a week. They are instantiated and executed independently of each other in separate threads and on separate computing devices. This has consequences for the ambiguity-aware version of the Context Toolkit described below. The Context Toolkit makes the distribution of the context architecture transparent to context-aware applications, handling all communications between applications and components. Context, in the form of events, is *dispatched* by the toolkit to interested applications at appropriate times. In other words, this toolkit shares the same basic event-based architecture used in the GUI world. The implication of this is that the architectural changes for handling ambiguity described in this thesis can be applied to the Context Toolkit.

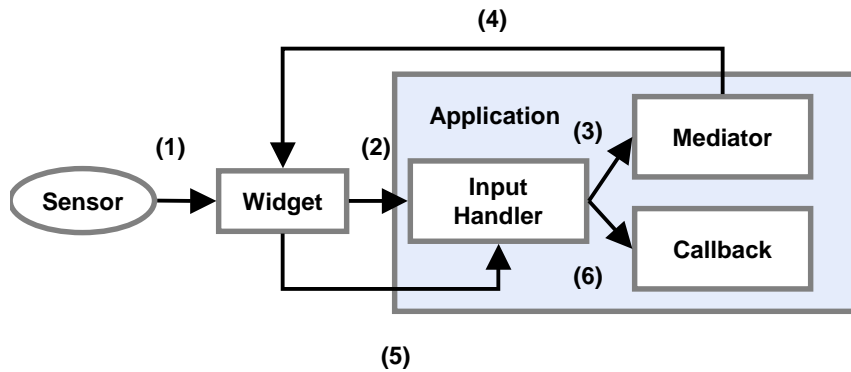
In order to highlight the similarities between event dispatch in the Context Toolkit and in the GUI world, we will describe a single interaction in the Context Toolkit: Initially, context, such as user motion in the In/Out Board, is implicitly sensed by a context widget.

Like input in a GUI toolkit, sensed context is split up into discrete pieces of information, analogous to input events. These pieces of information, or context, contain a set of attribute name-value pairs, such as the time and location that the motion occurred. Context events are sent to one or more interpreters. For example, a motion event occurring at 5pm on a weekday may be interpreted to be Jen because she always leaves the house at 5pm to walk her dog. Any interpretations are then sent by the widget to its subscribers. If those subscribers include other widgets or aggregators, the events will be propagated to *their* subscribers, and so on. So event dispatch is distributed in a multi-threaded fashion over all running widgets and aggregators (unlike the centralized, synchronous event dispatch system of a GUI toolkit).

### 7.2.2 Adding ambiguity to the Context Toolkit

In the modified Context Toolkit, a context interpreter (or widget) may create multiple ambiguous interpretations of context. Each interpretation contains information about the source of the interpretation (the event it is an interpretation of), and what produced it. The result is an ambiguous event hierarchy. The widget creating the interpretations sends just those events that match the subscription request, along with any ambiguous alternatives, to subscribers. The rest of the event hierarchy must be explicitly requested by subscribers. This minimizes network calls (which can be quite extensive and costly) as a hierarchy is generated and mediated. Providing each widget, aggregator and application with access to the entire graph for each piece of context and having to update each one whenever a change occurs impedes the system's ability to deliver context in a timely fashion.

Since other widgets and aggregators, as well as applications, may subscribe to a widget, all of these components were modified to include support for dealing with ambiguous events (and mediation of those events). A subscriber in the Context Toolkit receives data through an input handler, which is responsible for dealing with distributed communications. We modified this input handler to check incoming context for ambiguity, and, if necessary to



**Figure 7-2:** The flow of control for mediation in modified context toolkit (CT-OOPS). (1) A sensor senses some user input and passes it to a widget. (2) The widget sends one or more interpretations of the sensed input over the network to the input handler, a toolkit component inside the application. (3) Since the input is ambiguous, the input handler sends it to a mediator (a toolkit library component selected by the application developer). (4) When an event is accepted or rejected, the widget is notified. (5) The widget sends the final correct choice to the input handler. (6) The final choice is finally delivered to the main application code.

send the context to a mediator, instead of the subscribing component. Figure 7-2 illustrates how the event dispatch system in the modified context toolkit works. Thus, mediators intervene between widgets (and aggregators) and applications in the same way that they intervene between recognizers and applications in OOPS.

Once a mediator provides feedback to the user, the user responds with additional input. The mediator uses this information to update the event graph. It does this by telling the widget that produced the events to accept or reject them as correct or incorrect (4). The widget then propagates the changes to its subscribers (5). If ambiguity is resolved, the events are delivered as normal (6) and subscribers can act on them. Otherwise, the mediation process continues.

Because context widgets are persistent, and used by multiple applications, we chose to support both sets of guarantees originally mentioned in Chapter 3 for components unaware of ambiguity. An application can request to receive only unambiguous information, and

install mediators that handle any ambiguity (or to wait for some user of some other application interested in the same data to mediate the ambiguity). This is the sole approach supported by OOPS (Chapter 4). Alternatively, an application may receive information regardless of ambiguity, without waiting for mediation to be completed. An application that has no basis for accepting or rejecting ambiguous events might choose this alternative in order to receive context immediately. However, that application will still have to choose which events to use. Finally, if an application wishes, it may receive ambiguous information and then be updated as mediators resolve that information, similarly to the dispatch cycle described in Chapter 3:

## **7.3 Mediating Simple Identity and Intention in the Aware Home**

### **7.3.1 The modified In/Out Board**

Continuing our example, we will describe some details of interaction with the In/Out Board in an ambiguous world. In the base application, occupants of the home are detected when they arrive and leave through the front door, and their state on the In/Out Board is updated accordingly. Figure 7-3 shows the front door area of our instrumented home, taken from the living room. In the photographs, we can see a small foyer with a front door and a coat rack. The foyer opens up into the living room, where there is a key rack and a small table for holding typical artifacts found near a front door. To this, we have added two ceiling-mounted motion detectors (one inside the house and one outside), a display, a microphone, speakers, a keyboard and an iButton dock beside the key rack.

When an individual enters the home, the motion detectors sense his presence. The current time, the order in which the motion detectors were set off and historical information about people entering and leaving the home is used to infer who the likely individual is,

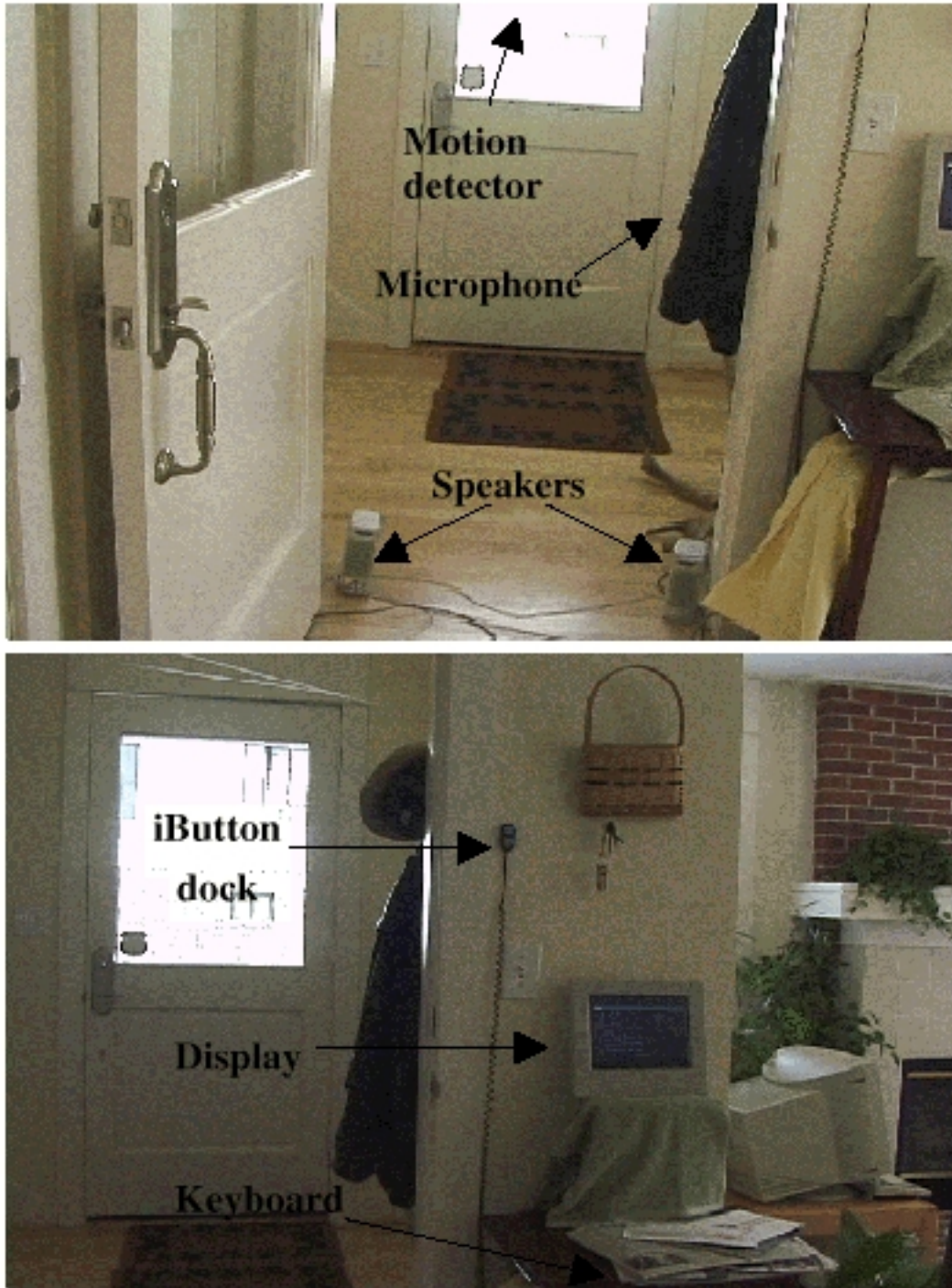
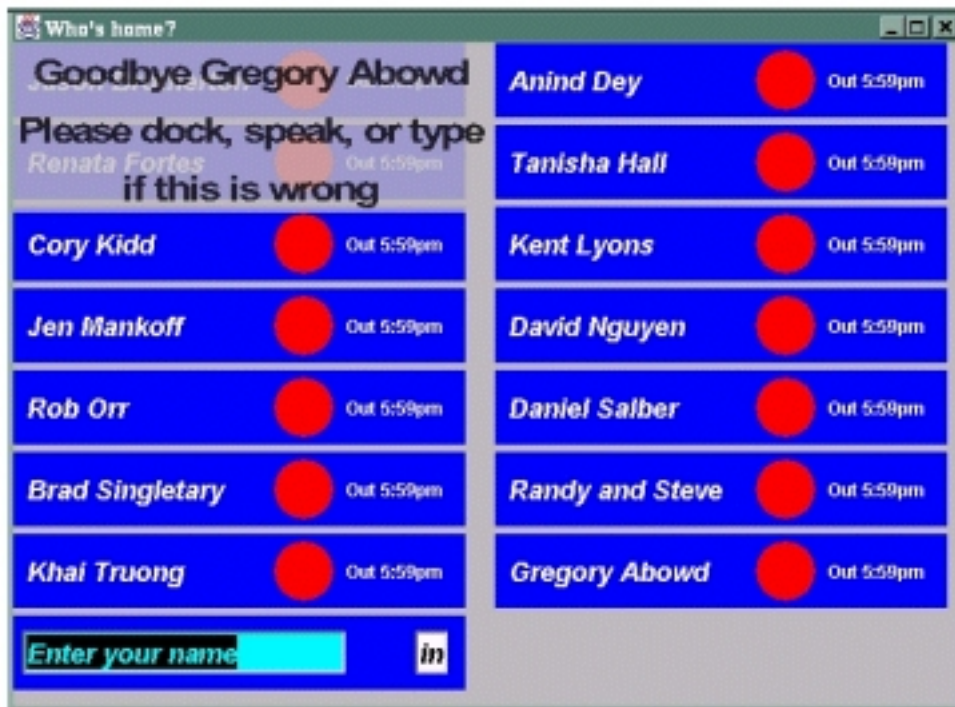


Figure 7-3: Photographs of the In/Out Board physical setup.



**Figure 7-4:** The In/Out Board with transparent graphical feedback.

and whether he is entering or leaving. A nearest-neighbor algorithm is then used to infer identity and direction of the occupant. This inference is indicated to the person through a synthesized voice that says “Hello Jen Mankoff” or “Goodbye Anind Dey”, for example. This inference is ambiguous, and the interpreter was modified to generate multiple possible inferences in our ambiguity-aware version of the Context Toolkit.

The first mediator we added shows a transparent graphical overlay (see Figure 7-4) that indicates the current state and how the user can correct it if it is wrong: using repetition in an alternative modality (speaking, docking or typing). Other mediators handle each of these, which can be used either alone, or in concert. After each attempt at mediation, additional feedback is given indicating how the new information is assimilated. There is no pre-defined order to their use. If the inference is correct, the individual can simply continue on as usual and the In/Out Board display will be updated with this new information.

Figure 7-1 shows the block diagram of the components in this system. The original version of the In/Out Board (the components shaded in grey) used only an iButton widget that did not generate ambiguity. Similarly to applications built in OOPS, the new version of the application was only modified to install the appropriate mediator, which has the role of intercepting and resolving any ambiguous information. Additionally, the widgets were modified to be able to generate ambiguous as well as unambiguous context information, and the other widgets shown in Figure 7-1 were added. When any of these widgets capture input, they produce not only their best guess as to the current situation, but also likely alternatives as well, creating an ambiguous event graph. Because of the separation of concerns provided by the context toolkit, the application did not have to be modified in order to receive data from these additional widgets.

As in the original implementation, the modified In/Out Board subscribes to unambiguous context information. When ambiguous information arrives, it is intercepted by a mediator that resolves it so that it can be sent to the application in its unambiguous form. The mediator uses this ambiguous information to mediate (accept and reject) or refine alternatives in the graph.

The mediator creates a timer to create temporal boundaries on this interaction. The timer is reset if additional input is sensed before it runs out. As the mediator collects input from the user and updates the graph to reflect the most likely alternative, it provides feedback to the user. It does this in two ways. The first method is to use a generic output service provided by the Context Toolkit. This service uses IBM ViaVoice<sup>TM</sup> to produce synthesized speech to provide feedback to the user. The second method is application-specific and is the transparent graphical overlay on the wall display shown in Figure 7-4. The transparent overlay indicates what the most likely interpretation of the user's status is, and what the user can do to change his status: *e.g.* "Hello Anind Dey. Please dock, type, or speak if this is wrong." As the timer counts down, the overlay becomes more transparent and fades away, at which point the top choice is accepted.



When all the ambiguity has been resolved by the user in the event graph, or the time has expired, the overlay will be faded completely and the correct unambiguous input is delivered to the In Out board (which, aside from installing mediation at instantiation time, was totally uninvolved in the process and unaware of the ambiguity). The display updates itself with the new status of the occupant. Also, the final decision is delivered back to the interpreter so it has access to the updated historical information to improve its ability to infer identity and direction.

### 7.3.2 Issues and problems

This is a work in progress. Although we have demonstrated the utility of the architecture in a new setting, there are some difficult issues remaining to be solved. The first of these deals with the design of mediation in a context-aware setting, while the remaining issues have to do with the underlying architectural support for ambiguity and mediation.

**Implicit mediation** In our example application, we chose to supply redundant mediation techniques, in an attempt to provide a smooth transition from implicit to explicit input techniques. For example, the motion detector is completely implicit, while docking, or typing, are very explicit.

However, the problem of implicit mediation is an especially difficult one, because much of the work in mediation of ambiguity is based on the premise that users are available for involvement. Recent work has shown that orthogonality in input modalities such as speech and pen can be used to resolve ambiguity more effectively without involving the user [67]. Any solution to the problem of implicit mediation will probably have to combine more effective approaches to automatic mediation with well-designed support for transitioning towards explicit input. Even more than in an application such as Burlap, issues of timing come in to play. Interrupting the user to resolve a trivial piece of sensed input would be inappropriate. Additionally, it is critical that mediation

design takes advantage of knowledge about how likely it is that sensed input has been misinterpreted.

**Handling multiple applications** Unlike traditional GUI applications, multiple context-aware applications may be using the same data about a user interaction. These applications may be separated in time and space, and may involve diverse users. Our architecture uses a naïve strategy that supports ambiguity resolution on a first-come first-serve basis. In some cases, this could create a situation in which someone is mediating data that is not about them. For example, if Jen is accessing Gregory’s calendar, she may end up being asked to mediate ambiguity about his location. Depending on Jen’s relationship to Gregory and knowledge of his schedule, this may be inappropriate. A better solution would be to use ownership and policies to decide who gets to mediate data. However, this means that Jen may have to wait indefinitely long to see the unambiguous data about Gregory’s schedule if her application is unwilling to receive ambiguous data. Clearly, more sophisticated strategies are needed. For example, applications could come with a default automatic mediator that makes a local choice about which answer to provide, without affecting the publicly available ambiguous information.

**Fusion** Fusion involves taking input from multiple ambiguous sources about the same input event (intended action by the user) and combining them to get one set of ambiguous possible actions. Although it is rare to have redundant sensors in the GUI world, it is quite common in a robust sensing environment. Thus, a complete solution to the problem of ambiguity in the context-aware computing world would benefit from structured support for the fusion problem, a problem that has been extensively addressed in other research areas such as robotics [12].

## 7.4 Other Settings For The Architecture

We believe that any computer system that involves alternative forms of input is likely to suffer from the problem of dealing with ambiguous input. This chapter shows that our architecture may indeed be added to other event-based toolkits. Our goal is to show that there is nothing unique to either the Context Toolkit or subArctic that is required in order to add support for ambiguity and mediation. One piece of evidence for this claim is that the Context Toolkit and subArctic have so little in common.

The architecture presented here has two main components, the input-handling subsystem and the mediation subsystem. In addition, it has one major data structure, the ambiguous event hierarchy. We will address each of these in turn, starting with the ambiguous event hierarchy and the mediation subsystem.

### 7.4.1 The ambiguous event hierarchy

The ambiguous event hierarchy is created from scratch and maintained by the modified architecture. Similar to the library of mediators provided with each toolkit, this data structure is independent of any given toolkit. It is the job of maintaining this hierarchy that requires direct modification to a toolkit (see below).

### 7.4.2 The mediation subsystem

Like the ambiguous event hierarchy, the mediation subsystem is implemented entirely independently and in fact interacts with the other modifications through the ambiguous event hierarchy. In other words, it is completely separate from any idiosyncrasies of the unmodified original toolkits.

### **7.4.3 The input handling subsystem**

The majority of the direct toolkit modifications must occur in the input handling subsystem of the toolkit being modified. Here, access to source code is very important because there may be hidden assumptions and constraints in how a toolkit handles input. Since most modern toolkits separate input handling from output and other issues, there is generally a clearly defined portion of those toolkits that should be modified.

Exactly how those modifications are done is specific to the toolkit being modified. As stated earlier, modern toolkits use events to describe discrete pieces of input. Assuming an object oriented toolkit, these events are generally instances of an event class. Events are dispatched to interactors and other input handlers.

Given the basic input architecture described in the previous paragraph, support for our architecture can be added. The input system must be extended to allow input handlers to return information about any interpretations they create when they receive an event. The event class must be extended to represent interpretations of events as well as raw input events, and to keep pointers to other events in the hierarchy. Finally, the event dispatch algorithm itself must be modified to dispatch any interpretations of events recursively, and to store those interpretations in the ambiguous event hierarchy. Outside of those changes, the only other addition needed is some sort of “ambiguity manager” that watches for ambiguity and requests mediation from the mediation subsystem when necessary.

### **7.4.4 Additional details**

More sophisticated changes such as the separation of feedback from action found in OOPS are not intrinsically necessary for the support of ambiguity, but can be helpful. These changes may be slightly more difficult to support, but are not limited to OOPS. In general, they require some management of the order in which events are dispatched and the locations

to which they are dispatched, but this is necessary for any level of sophistication in standard examples of input event dispatching.

In conclusion, there is nothing about subArctic or the Context Toolkit that is necessary for the support of ambiguity, recognition, or mediation. This chapter has shown that our architecture truly is general and generalizable to any setting that supports communication between a user and a computer.

## Chapter 8

# CONCLUSIONS

### 8.1 Contributions

Recall the thesis statement given at the beginning of this document:

**A graphical user interface toolkit architecture that models recognition ambiguity at the input level can make it easier to build interfaces to recognition systems. Specifically, such a toolkit can provide re-usable interfaces for resolving ambiguity in recognition through mediation between the user and computer.**

We have demonstrated our claims in this document as follows:

- Chapter 1 describes what a recognition error is in the eyes of other peers in the field, and shows that ambiguity is a reasonable way to model potential errors.
- Chapter 2 provides an in-depth survey of existing mediators verified by outside peers in various fashions (commercial value, user studies, publication, *etc.*). This helps to identify the current state of the art in interfaces to recognition systems (that is, what a toolkit such as OOPS should simplify). Since all of the examples given in Chapter 2 were one-off systems, we believe that a toolkit which streamlines the use, modification, and substitution of such mediators, must make these types of interfaces easier to build.

- Chapter 3 describes our architecture to support recognition, and shows that such an architecture did not previously exist. In particular, it shows that previous toolkits have little integrated support for recognition and none for mediation, and describes some of the steps that application developers go through in order to build interfaces that make use of recognizers. It highlights the key architectural algorithms necessary to make our architecture possible, and illustrates how flexible combinations of mediators and meta-mediators can be supported and be used to implement mediation strategies.
- Chapter 4 focuses on one particular instance of the architecture described in Chapter 3. This chapter gives specific details on the types of mediators and meta-mediators that should be supported (based on the survey presented in Chapter 2) and ends with a concrete illustration of how such a toolkit might be used.
- Chapter 5 shows that our architecture is sufficient to handle one of the more complex existing applications found in the literature. This demonstrates that our architecture is capable of at least supporting the state of the art in ad-hoc solutions.
- Chapter 6 shows that OOPS allows us to build one-off solutions to previously intractable problems identified in the survey of Chapter 2. This demonstrates the ability of the toolkit to act as a stepping stone for further explorations of the design space that go beyond the state of the art.
- Chapter 7 shows that our architecture is itself a re-usable solution that is not dependent on the toolkit we extended, nor limited to traditional GUI forms of recognition.

We believe that, taken as a whole, this document provides evidence that we have succeeded in verifying the thesis statement given above. That it is true that a toolkit such as OOPS, that models recognition ambiguity at the input level, can make it easier to build interfaces to recognition systems, in particular by support mediation of uncertain input in a re-usable, lightweight fashion.

## 8.2 Future Work

As in any thesis, we have had to focus the goals of this work on a few deep problems, leaving many interesting issues lying by the wayside. A few of them are listed below.

### 8.2.1 Evaluation

It is our hope that the existence of a toolkit such as OOPS will encourage comparison of different mediation techniques in the same setting. Because OOPS makes it so easy to swap mediation techniques without modifying application or recognizer, it is now feasible to run large studies comparing many different mediation techniques.

In particular, we would like to investigate the following questions:

- At what point in time is it appropriate to do mediation? When should the system initiate mediation, and when should it be up to the user? What other heuristics can we identify in mediation design? For example, mediators tend to either default to do nothing or to accept the top choice [48]. Designers tend to tie this choice to recognition accuracy: what other factors should influence this choice?
- When is a repetition technique better than a choice technique, and vice versa?
- Are highly interactive mediators such as the one we presented in Section 6.1 actually any better than the simpler originals they are derived from?
- Past work has shown that users are reluctant to switch modality, even when mediation may be less error prone and quicker in the new modality [79]. Do certain mediation designs encourage change of modality better than others?
- Can mediation contribute to robustness of interaction by avoiding cascading errors, and other common problems in recognition?

In studying these questions, we must select specific domains and specific applications. Are the results of the user studies generally useful or are they only true for the domain



and application being tested? This is an issue germane to any usability investigation. We plan to investigate application areas for which the results will have important implications regardless of their generalizability.

In particular, we plan to investigate the domain of text entry, interesting because it is a crucial task for many people. Many circumstances can lead to situations in which a two-handed, traditional keyboard cannot be used (*e.g.* mobile computing, disability). Yet real-world input speeds seem to be limited to 30 wpm in the absence of a keyboard. We believe that by identifying optimum mediation techniques, we can improve this.

The question about robustness is particularly relevant to context-aware computing, and our initial forays into that domain have not led to any measurable changes in robustness. We hope that by focusing specifically on that problem we may be able to improve on this situation.

### **8.2.2 More examples**

By making existing mediation techniques accessible, OOPS has created a challenge for interface designers to discover new (and hopefully better) ways of doing mediation. At one extreme, the ideal mediator should be so well integrated with the application that the user is not even aware that mediation is going on. Such a mediator would probably only be useful in one specific setting, and must look and feel radically different from most of the mediators presented in this document.

At the other extreme, there might be a whole new class of mediation techniques that is as general as choice or repetition but perhaps more accessible, or flexible.

In addition, there are several issues remaining from our survey of mediation systems, including mediation strategies applicable to segmentation errors, non-GUI applications, and command recognition. All are difficult to mediate because they have no obvious representation (unlike, for example, the text generated in handwriting recognition).

### 8.2.3 New settings

Finally, we would like to see mediation applied to new settings. We have already taken steps in this direction by using a mediator in a context-aware application (see Chapter 7). Are there other examples of “non-recognition” ambiguous settings where mediation might be useful? How does the role of a mediator change in such settings? For example, we discovered a set of heuristics for mediation that is orthogonal to both classes of mediation in our context-aware work [19].

Some examples of different settings include ambient displays (in which a certain amount of ambiguity may even be intentional, and ambiguity exists in the output of the system as well as the input of the user), and search tasks (in which there are multiple correct answers and the user may not know them ahead of time).

Some examples of difficult areas for mediation include:

- Sensor fusion and other complex, multi-recognizer systems.
- Extremely ambiguous (very uncertain) settings such as word prediction. These are settings where recognition is more likely to be wrong than right.
- “Invisible” recognition, such as commands which may not have any visual effect on the screen.
- Non-interactive settings or distributed settings. As a first step, it is clear that maintaining ambiguity in such situations could be beneficial. Consider the eClass project, formerly called Classroom 2000 [69, 1]. eClass records audio in a classroom setting and does off-line transcription. If it maintained ambiguity about phonemes, it might be possible to increase search accuracy (search terms are translated into phonemes and matched against the transcribed lecture).
- Settings in which the ambiguity is about a person other than the one who is mediating.

- Search tasks in which there are multiple correct answers, and the user does not know which is correct.

### 8.3 Conclusions

This dissertation has set forth a problem in the form a thesis statement, and gone on to break that problem down and provide one possible solution. We have shown the relevance and importance of that solution in comparison to related work and illustrated the depth and breadth of the problem that it allows us to solve. Finally, we have put forth a list of some of the future work that remains.

One particular challenge remains: This work was originally begun with the intent to make computers more accessible to people who could not use standard input devices such as a mouse and keyboard. Although Burlap does not address this issue, some of the work presented here (in particular, the magnified mouse and the word predictor) begins to address these issues. The challenge is to address the difficult research questions described above in settings that also have social value, such as supporting people with disabilities, applications that may help to solve problems faced by real people, for which there are currently no good solutions.

## Appendix A

### The complete set of algorithms needed to handle ambiguous event dispatch

#### A.1 Maintenance Algorithms for the Event Graph

Events contain the following methods for collecting information from or updating the event graph: `is_ambiguous()`, `lstinline:accept()`, `reject()`, `leaf_nodes()`, `conflicting_siblings()` and `root_sources()`. Additional methods (not shown here) exist to return standard information about individual events such as its direct interpretations or sources.

**Source Code A-1:** The algorithm used to determine whether an event hierarchy is ambiguous (`is_ambiguous()`)

```
// Check if this event is ambiguous
boolean is_ambiguous() {
    // There are conflicting siblings
    if ( conflicting_siblings () not empty)
        return true;
    // There is at least one ambiguous interpretation
    for each interpretation
        if interpretation.is_ambiguous()
            return true;
    // An open event is by definition ambiguous
    return this is open;
}
```

**Source Code A-2:** The algorithm to accept events (accept())

```
// Accept events, propagating up hierarchy
accept() {
  // Accept sources
  for each source in sources
    source.accept();
  // Reject conflicting siblings
  for each sibling in conflicting_siblings ()
    sibling.reject();
  closure = accepted;
  // Notify recognizer that produced this event
  producer().notify_of_accept (this);
}
```

**Source Code A-3:** The algorithm to reject events (reject ())

```
// Reject events, propagating down hierarchy
reject () {
  // Reject interpretations
  for each interpretation
    interpretation.reject ();
  closure = rejected;
  // Notify recognizer that produced this event
  producer().notify_of_reject (this);
}
```

**Source Code A-4:** The algorithm used to retrieve the leaf nodes of an event hierarchy

```
once ambiguity has been resolved (leaf_nodes())
// Check if this event has any siblings that are ambiguous
Set leaf_nodes() {
  Set leaves
  if this.interpretations () is empty
    add this to leaves
  else
    for each interpretation in this.interpretations ()
      add interpretation.leaf_nodes () to leaves
  return leaves;
}
```

**Source Code A-5:** The algorithm used to determine whether an event has siblings that are ambiguous (`conflicting_siblings()`)

```
// Check if this event has any siblings that are ambiguous
Set conflicting_siblings () {
    Set conflicts ;
    for each sibling {
        if (( sibling is not related to this) &&
            ( sibling is open))
            conflicts .add(sibling);
    }
    return conflicts;
}
```

**Source Code A-6:** The algorithm that returns the root sources of an event (`root_sources()`)

```
Set root_sources() {
    Set s;
    if (sources is empty)
        add this to s;
    else
        for each source in sources
            add source.root_sources() to s;
    return s;
}
```

## A.2 Algorithms for Resolving Ambiguity in Mediators and Meta-Mediators

Each mediator and meta mediator differs from the others in how exactly it handles ambiguity, but we give a sample for each below (`mediate()` and `meta_mediate()`). More standard is the relationship between `meta_mediate()` and `continue_mediation()` in a meta mediator. `continue_mediation()` differs from `meta_mediate()` only in that it must explicitly pass control back to the mediation subsystem when ambiguity is resolved or it runs out of mediators. This is because it was not invoked by the mediation subsystem, but rather by a mediator that had previously deferred mediation. `handle_modification()` shows how a

meta mediator deals with a situation in which a deferred event graph has been modified by interpretation of a later event. This is essentially a case when mediation needs to be continued for external reasons (rather than because of the mediator).

**Source Code A-7:** The algorithm used by a mediator to mediate an event hierarchy

```
(mediate())
// in a mediator
mediate(event root, Set relevant_events) {
    if some event in root is correct
        event.accept()
    if some event in root is wrong
        event.reject ()
    if this is done and an event was accepted or rejected
        return RESOLVE
    if this is done and no changes were made
        return PASS
    if this is not done and we have to wait for some reason
        set up wait condition
        cache root
        return DEFER
}
// the wait condition has happened
wait_completed(event root) {
    // mm is the meta-mediator in which this is installed
    mm.continue_mediation(root, this)
}
```

**Source Code A-8:** The algorithm used by meta-mediators to send event graphs to mediators for resolution (meta\_mediate())

```
// in a meta mediator
meta_mediate(event root) {
    // selected mediators in order according to this meta mediator's policy
    for each mediator
        switch(val=mediator.mediate(root, this))
        case RESOLVE:
            if (not root.is_ambiguous())
                return RESOLVE;
        case PASS:
```

```

        continue;
    case DEFER
        cache mediator with root
        return val;
    }
    return PASS;
}

```

**Source Code A-9:** The algorithm used to continue deferred mediation (`continue_mediation()`)

```

// in a meta mediator
// pick up mediation where it stopped
continue_mediation(event root, mediator m) {
    if (not root.is_ambiguous()) return;
    for each mediator after m based on policy
        switch(val=mediator.mediate(root, this))
            case RESOLVE:
                // tell the mediation subsystem
                // to pick up from here
                // (See Source Code 3-8)
                mediation_subsystem.continue_mediation(root, this);
                return;
            case PASS:
                continue;
            case DEFER
                cache mediator with root;
                return;
        }
    // tell the mediation subsystem to pick
    // up from here (See Source Code 3-8)
    mediation_subsystem.continue_mediation(root, this);
}

```

**Source Code A-10:** The algorithm used by meta mediators to handle updates to an event

```

graph (handle_modification())
// in each meta mediator
handle_modification(event root, event newchild) {
    // retrieve the correct mediator from the cache of deferred mediators
    mediator m = cache.retrieve root
    if cancel_mediation_p(root)
        // start over, from scratch

```



```

        mediation_subsystem.resolve_ambiguity(root);
    else
        // update m
        m.update(root, newchild);
}

```

### A.3 The Algorithms Associated with Event Dispatch

Here we show the most complex form of event dispatch, which includes support for deferred mediation, and components unaware of ambiguity.

**Source Code A-11:** The modified event dispatch algorithm for components unaware of ambiguity (`dispatch_event()`)

```

dispatch_event(event e) {
    // Dispatch to components unaware of ambiguity. If any of them use the event, we are done.
    for each ambiguity-unaware component not in "wait"
        if (component.event_is_useful(e)) {
            component.use(e);
            return;
        }

    // If the event was not used, dispatch it to components aware of ambiguity (identically
    // to Source Code 3-5)
    generate_interps(e);
    if e.is_ambiguous() {
        mediation_subsystem.resolve_ambiguity(e);
    }
    complete_dispatch(e);
}

```

**Source Code A-12:** A helper algorithm for event dispatch (`generate_interps()`)

```

// Dispatch an event to recognizers, recursively.
// Returns the newly generated interpretations.
public Set generate_interps(event e) {
    Set local_new_interps;
    Set new_interps;
    for each recognizer

```

```

        if (recognizer.event_is_useful(e)) {
            recognizer.use(e);
            local_new_interps.add(
                e.new_interpretations());
        }
    for each interpretation in
        local_new_interps
    new_interps.add(
        generate_interps(interpretation));
    new_interps.addall(local_new_interps);
    return new_interps;
}

```

**Source Code A-13:** The final phase of the event dispatch algorithm (`complete_dispatch()`)

```

// separate out the final portion of dispatch so that when mediation is deferred, causing
// dispatch_event to exit, we can still complete the dispatch without redoing everything
complete_dispatch (event e) {
    // Dispatch any unambiguous leaf nodes to components unaware of ambiguity
    for each closed event ce in e.leaf_nodes()
        for each ambiguity-unaware component in “wait”
            if (component.event_is_useful(ce)) {
                component.use(ce);
                break;
            }
    }
}

```

**Source Code A-14:** The algorithm used by the event dispatch system to send event graphs to meta mediators for resolution (`resolve_ambiguity()`)

```

// in mediation subsystem
resolve_ambiguity(event root) {
    if (not root.is_ambiguous()) return;
    // select each meta mediator in the queue in turn
    for each meta-mediator
        switch(meta-mediator.meta_mediate(root))
        case RESOLVE:
            if (not root.is_ambiguous())
                return;

```

```

    case PASS:
        continue;
    case DEFER
        cache meta–mediator with root
        return;
}

```

**Source Code A-15:** The algorithm used when a mediator that has paused mediation wishes to pass control back to the mediation subsystem (`continue_mediation()`)

```

// in mediation subsystem
continue_mediation(event root, meta_mediator mm) {
    if (not root.is_ambiguous()) return;
    // pick up where we left off – go to the next meta mediator in the queue
    // just as if m had returned immediately
    for each meta mediator after mm
        switch(meta_mediator.meta_mediate(root))
            case RESOLVE:
                if (not root.is_ambiguous())
                    return ;
            case PASS:
                continue;
            case DEFER
                cache meta–mediator with root
                return;
    }
}

```

## A.4 Some Meta-Mediators

Traverses the interactor hierarchy of an interface in a depth-first search, looking for interactive mediators such as choice or repetition mediators. Any event always has a bounding box (which, by default, it inherits from its source). If any bounding box in the event hierarchy overlaps the bounding box of a mediator found in the traversal, that mediator is given a chance to mediate. The traversal continues until the entire interactor hierarchy has been searched or the event hierarchy is no longer ambiguous.

**Source Code A-16:** An alternative implementation of positional (`meta_mediate()`)

```
// in positional_meta_mediator, which has standard
// implementations of update, cancel, etc.

// returns PASS, RESOLVE, or DEFER
int meta_mediate(event root)
    // put all of the events in root's tree in s
    set s = root.flatten ()

    queue q = interactor_hierarchy.depth_first_traversal ()
    // location is a rectangular area of the screen,
    // or an interactor (which has a location
    // that may change dynamically with time)
    for each interactor in q that is of type mediator {
        // retrieve events in hierarchy inside location
        res = elements of s inside interactor.global_bounds()
        if (res is not empty)
            switch(interactor.mediate(root, res)) {
                case RESOLVE:
                    if (not root.is_ambiguous())
                        return RESOLVE
                case PASS:
                    continue;
                case DEFER
                    cache interactor with root
                    return DEFER;
            }
    }
    return PASS;
}
```

## VITA

Jennifer Mankoff received her PhD in 2001 from Georgia Institute of Technology. Her thesis research focuses on toolkit-level support for building interaction techniques that are applicable to the use of recognition-based input. Jennifer believes that recognition-based user interfaces face significant usability problems due to recognition errors and ambiguity. Because the use of recognition is becoming more common, the HCI community must address these issues. Her work is contributing to this effort. As of Spring 2001, she and her advisors, Gregory Abowd and Scott Hudson, investigated the foundations of this area and built toolkit-level support for building interfaces to recognition systems. She also investigated interface issues for situations when ambiguity is not normally addressed, such as context-aware and intelligent computing, and issues of segmentation ambiguity, and of target ambiguity. This work culminated in one journal paper [49] and two conference publications (CHI'00 [51] and UIST'00 [49]).

As a broader goal, Jennifer Mankoff is interested in making computers more accessible both in environments where keyboard and mice are not tenable, such as ubiquitous and mobile computing, and for people with special needs. With the help of Dr. Mynatt and Dr. Moore, she lead a group of one PhD student and four undergraduate students to develop a suite of user interface techniques and training techniques for a user with locked-in syndrome. This user communicates by means of a brain implant which controls cursor movement over a keyboard. She also designed two ambient displays as part of her investigations of alternative input and output modalities.

Jennifer's PhD was funded by two fellowships, the NSF Traineeship (Fall 1995–Spring 1997), and the IBM Fellowship (Fall 2000–Spring 2001), as well as a variety of NSF grants.

In addition she was offered (and declined) the Intel Fellowship for the 2000–2001 calendar year. She received her Bachelors degree at Oberlin College in 1995, where she graduated with High Honors. In the past seven years, she has interned at two companies (FX Pal (96) and AT&T Bell Labs (94) (now Lucent Technologies), as well as Argonne National Labs (93) and Carnegie Mellon University (99). She has served on the program committees for UIST 2001 and for the ASSETS 2000 conference on Assistive Technology, and reviewed papers for CSCW and UIST. She was an Invited Panelist at ASSETS'97, and participated in the CHI'00 doctoral consortium. Jennifer has also served locally as a member of her graduate student committee since her arrival at Georgia Tech. She helped to educate her fellow students on how to avoid Repetitive Strain injuries (RSI) and coordinated the RSI lending library.

In addition to her technology related interests, Jennifer pursues several hobbies. To date, she has trained two dogs for Canine Companions for Independence, an organization that provides dogs for work with people with disabilities. She was a member of the Emory Symphony Orchestra for two years and the College of Computing Virtual String Quartet for six years, and in the past taught both Viola and Piano to beginning musicians.

## BIBLIOGRAPHY

- [1] ABOWD, G. D., HARVEL, L. D., AND BROTHERTON, J. A. Building a digital library of captured educational experiences. In *Invited paper for the 2000 International Conference on Digital libraries* (Kyoto, Japan, November 2000).
- [2] AINSWORTH, W. A., AND PRATT, S. R. Feedback strategies for error correction in speech recognition systems. *International Journal of Man-Machine Studies* 36, 6 (1992), 833–842.
- [3] ALM, N., ARNOTT, J. L., AND NEWELL, A. F. Input acceleration techniques for severely disabled nonspeakers using a communication system. *Communications of the ACM* 35, 5 (May 1992), 54–55.
- [4] APPLE COMPUTER. Available at: [ftp://manuals.info.apple.com/Apple\\_Support\\_Area/Manuals/newton/NewtonProgrammerGuide20.pdf](ftp://manuals.info.apple.com/Apple_Support_Area/Manuals/newton/NewtonProgrammerGuide20.pdf)  
*Newton Programmer's Guide, For Newton 2.0.*
- [5] ARNOTT, J. L., NEWELL, A. F., AND ALM, N. Prediction and conversational momentum in an augmentative communication system. *Communications of the ACM* 35, 5 (May 1992), Available at: <http://www.acm.org/pubs/toc/Abstracts/0001-0782/129878.html>  
46–57.
- [6] BABER, C., AND HONE, K. S. Modelling error recovery and repair in automatic speech recognition. *International Journal of Man-Machine Studies* 39, 3 (1993), 495–515.
- [7] BAUMGARTEN, BARKSDALE, AND RUTTER. *IBM® ViaVoice™ QuickTutorial®*, 2000.
- [8] BOLT, R. A. “Put-That-There”: Voice and gesture at the graphics interface. *Computer Graphics* 14, 3 (July 1980), 262–270.
- [9] BOLT, R. A., AND HERRANZ, E. Two-handed gesture in multi-modal natural dialog. In *Proceedings of the ACM UIST'92 Symposium on User Interface Software and Technology* (Monterey, CA, 1992), ACM Press, pp. 7–14.
- [10] BREGMAN, A. S. *Auditory scene analysis: The perceptual organization of sound*. MIT Press, 1990.
- [11] BRENNAN, S. E., AND HULTEEN, E. A. Interaction and feedback in a spoken language system: A theoretical framework. *Knowledge-Based Systems* 8, 2-3 (1995), 143–151.
- [12] BROOKS, R. R., AND IYENGAR, S. S. *Multi-sensor fusion: Fundamentals and applications with software*, 1 ed. Prentice Hall, Englewood Cliffs, NJ, 1997.

- [13] BRUMITT, B. L., MEYERS, B., KRUMM, J., KERN, A., AND SHAFER, S. Easyliving: Technologies for intelligent environments. In *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2K)* (Bristol, United Kingdom, September 2000), Springer-Verlag, Available at: <http://www.research.microsoft.com/barry/research/huc2k-final.pdf> pp. 12–27.
- [14] BUSKIRK, R. V., AND LALOMIA, M. The just noticeable difference of speech recognition accuracy. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems* (Denver, CO, 1995), Interactive Posters, ACM Press, Available at: [http://www.acm.org/sigchi/chi95/proceedings/intpost/rvb\\_bdy2.htm](http://www.acm.org/sigchi/chi95/proceedings/intpost/rvb_bdy2.htm) p. 95.
- [15] CHANG, B.-W., MACKINLAY, J. D., ZELLWEGER, P. T., AND IGARASHI, T. A negotiation architecture for fluid documents. In *Proceedings of the ACM UIST'98 Symposium on User Interface Software and Technology* (San Francisco, CA, November 1998), Papers: Enabling Architectures, ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/uist/288392/p123-chang/p123-chang.pdf> pp. 123–132.
- [16] COHEN, P., JOHNSTON, M., MCGEE, D., SMITH, I., OVIATT, S., PITTMAN, J., CHEN, L., AND CLOW, J. QuickSet: Multimodal interaction for simulation set-up and control. In *Proceedings of the Fifth Applied Natural Language Processing meeting* (Washington, DC, 1997), Association for Computational Linguistics.
- [17] COHEN, P. R., CHEYER, A., ADAM, J., WANG, M., AND BAEG, S. C. An open agent architecture. In *Readings in Agents*, M. Huhns and M. Singh, Eds. Morgan Kaufmann, March 1994, pp. 1–8. Originally published in AAAI Spring Symposium, Technical Report SS-94-03.
- [18] DEY, A. K. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, Georgia Institute of Technology, 2000. Dr. Gregory Abowd, Advisor.
- [19] DEY, A. K., MANKOFF, J., AND ABOWD, G. D. Distributed mediation of imperfectly sensed context in aware environments. Tech. Rep. GIT-GVU-00-14, Georgia Institute of Technology, GVV Center, 2000.
- [20] DEY, A. K., SALBER, D., AND ABOWD, G. D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human Computer Interaction* 16, 2–3 (2001). Context Aware Computing anchor article.
- [21] DIX, A., FINLAY, J., ABOWD, G., AND BEALE, R. *Human-Computer Interaction*, 2 ed. Prentice Hall, 1998, ch. Chapter 6—Models of the user in design.
- [22] EDWARDS, W. K., HUDSON, S. E., MARINACCI, J., RODENSTEIN, R., SMITH, I., AND RODRIGUES, T. Systematic output modification in a 2D UI toolkit. In *Proceedings of the ACM UIST'97 Symposium on User Interface Software and Technology* (Banff, Canada, October 1997), ACM Press, pp. 151–158.



- [23] FOLEY, J. D., WALLACE, V. L., AND CHAN, P. The human factors of computer graphics interaction techniques. *IEEE Computer Graphics and Applications* 4, 11 (November 1984), 13–48.
- [24] FRANKISH, C., HULL, R., AND MORGAN, P. Recognition accuracy and user acceptance of pen interfaces. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems* (Denver, CO, 1995), Papers: Pen Interfaces, ACM Press, Available at: <http://www.acm.org/sigchi/chi95/proceedings/papers/crf.bdy.htm> pp. 503–510.
- [25] FRANKISH, C., JONES, D., AND HAPESHI, K. Decline in accuracy of automatic speech recognition as function of time on task: Fatigue or voice drift? *International Journal of Man-Machine Studies* 36, 6 (1992), 797–816.
- [26] GARAY-VITORIA, N., AND GONZALEZ-ABASCAL, J. Intelligent word-prediction to enhance text input rate (A syntactic analysis-based word-prediction aid for people with severe motor and speech disability). In *Proceedings of IUI'97 International Conference on Intelligent User Interfaces* (1997), Short Papers, Available at: <http://www.acm.org/pubs/articles/proceedings/uist/238218/p241-garay-vitoria/p241-garay-vitoria.pdf> pp. 241–244.
- [27] GOLDBERG, D., AND GOODISMAN, A. STYLUS user interfaces for manipulating text. In *Proceedings of the ACM UIST'91 Symposium on User Interface Software and Technology* (Hilton Head, SC, 1991), ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/uist/120782/p127-goldberg/p127-goldberg.pdf> pp. 127–135.
- [28] GONG, Y. Speech recognition in noisy environments: A survey. *Speech Communication* 16, 3 (April 1995), 261–291.
- [29] GREENBERG, S., DARRAGH, J. J., MAULSBY, D., AND WITTEN, I. H. Predictive interfaces: what will they think of next? In *Extra-ordinary Human-Computer Interaction: interfaces for users with disabilities*, A. D. N. edwards, Ed., Cambridge series on human-computer interaction. Cambridge University Press, New York, NY, 1995, pp. 103–139.
- [30] HALVERSON, C., KARAT, C., KARAT, J., AND HORN, D. The beauty of errors: Patterns of error correction in desktop speech systems. In *Proceedings of INTERACT* (1999), pp. 133–140.
- [31] HENRY, T. R., HUDSON, S. E., AND NEWELL, G. L. Integrating gesture and snapping into a user interface toolkit. In *UIST'90. Third Annual Symposium on User Interface Software and Technology. Proceedings of the ACM SIGGRAPH Symposium* (Snowbird, UT, October 1990), ACM Press, pp. 112–122.
- [32] HORVITZ, E. Principles of mixed-initiative user interfaces. In *Proceedings of ACM CHI'99 Conference on Human Factors in*

- Computing Systems* (Pittsburgh, PA, 1999), vol. 1, ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/chi/302979/p159-horvitz/p159-horvitz.pdf> pp. 159–166.
- [33] HUDSON, S., AND SMITH, I. *The subArctic User's Manual*. Georgia Institute of Technology, Available at: [http://www.cc.gatech.edu/gvu/ui/sub\\_arctic/sub\\_arctic/doc/users\\_manual.html](http://www.cc.gatech.edu/gvu/ui/sub_arctic/sub_arctic/doc/users_manual.html) September 1996.
- [34] HUDSON, S. E., AND NEWELL, G. L. Probabilistic state machines: Dialog management for inputs with uncertainty. In *Proceedings of the ACM UIST'92 Symposium on User Interface Software and Technology* (Monterey, CA, 1992), Toolkits, pp. 199–208.
- [35] HUDSON, S. E., AND SMITH, I. Supporting dynamic downloadable appearances in an extensible UI toolkit. In *Proceedings of the ACM UIST'97 Symposium on User Interface Software and Technology* (Banff, Canada, October 1997), ACM Press, pp. 159–168.
- [36] HUERST, W., YANG, J., AND WAIBEL, A. Interactive error repair for an online handwriting interface. In *Proceedings of ACM CHI'98 Conference on Human Factors in Computing Systems* (Los Angeles, CA, 1998), vol. 2 of *Student Posters: Interaction Techniques*, ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/chi/286498/p353-huerst/p353-huerst.pdf> pp. 353–354.
- [37] IGARASHI, T., MATSUOKA, S., KAWACHIYA, S., AND TANAKA, H. Interactive beautification: A technique for rapid geometric design. In *Proceedings of the ACM UIST'97 Symposium on User Interface Software and Technology* (Banff, Canada, 1997), Constraints, Available at: <http://www.acm.org/pubs/articles/proceedings/uist/263407/p105-igarashi/p105-igarashi.pdf> pp. 105–114.
- [38] JOHNSTON, M., COHEN, P., MCGEE, D., OVIATT, S. L., PITTMAN, J. A., AND SMITH, I. Unification-based multimodal integration. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics* (March 1997), Association for Computational Linguistics Press.
- [39] KABBASH, P., AND BUXTON, W. The “prince” technique: Fitts’ law and selection using area cursors. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems* (Denver, CO, 1995), Papers, ACM Press, pp. 273–279.
- [40] KEATES, S., CLARKSON, P. J., AND ROBINSON, P. Developing a methodology for the design of accessible interfaces. In *Proceedings of the 4th ERCIM Workshop on 'User Interfaces for All'* (1998), Papers: Design Methodology for Universal Access, ERCIM, Available at: <http://ui4all.ics.forth.gr/UI4ALL-98/keates.pdf> p. 15.

- [41] KIDD, C. D., ORR, R. J., ABOWD, G. D., ATKESON, C. G., ESSA, I. A., MACINTYRE, B., MYNATT, E., STARNER, T. E., AND NEWSTETTER, W. The aware home: A living laboratory for ubiquitous computing research. In *Proceedings of the Second International Workshop on Cooperative Buildings, CoBuild'99* (October 1999), Position paper.
- [42] KOESTER, H. H., AND LEVINE, S. P. Validation of a keystroke-level model for a text entry system used by people with disabilities. In *First Annual ACM/SIGCAPH Conference on Assistive Technologies* (Marina del Rey, CA, 1994), Augmentative Communication, ACM Press, pp. 115–122.
- [43] KURTENBACH, G., AND BUXTON, W. User learning and performance with marking menus. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems* (Boston, MA, 1994), Papers: Pen Input, ACM Press, pp. 258–264.
- [44] LANDAY, J. A. *Interactive Sketching for the Early Stages of User Interface Design*. PhD thesis, Carnegie Mellon University, 1996. Brad A. Myers, Advisor.
- [45] LANDAY, J. A., AND MYERS, B. A. Extending an existing user interface toolkit to support gesture recognition. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems – Adjunct Proceedings* (Amsterdam, The Netherlands, 1993), Short Papers (Posters): Interaction Techniques I, ACM Press, pp. 91–92.
- [46] LANDAY, J. A., AND MYERS, B. A. Interactive sketching for the early stages of user interface design. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems* (Denver, CO, 1995), Papers: Programming by Example, ACM Press, Available at: <http://www.acm.org/sigchi/chi95/proceedings/papers/jal1bdy.htm> pp. 43–50.
- [47] LONG, JR., A. C., LANDAY, J. A., AND ROWE, L. A. Implications for a gesture design tool. In *Proceedings of ACM CHI'99 Conference on Human Factors in Computing Systems* (Pittsburgh, PA, 1999), Papers: Alternatives to QWERTY, ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/chi/302979/p40-long/p40-long.pdf> pp. 40–47.
- [48] MANKOFF, J., AND ABOWD, G. D. Error correction techniques for handwriting, speech, and other ambiguous or error prone systems. Tech. Rep. GIT-GVU-99-18, Georgia Tech Gvu Center, 1999.
- [49] MANKOFF, J., ABOWD, G. D., AND HUDSON, S. E. OOPS: A toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces. *Computers and Graphics (Elsevier)* 24, 6 (December 2000), 819–834.
- [50] MANKOFF, J., HUDSON, S. E., AND ABOWD, G. D. Interaction techniques for ambiguity resolution in recognition-based interfaces. In *Proceedings of the ACM UIST 2000 Symposium on User Interface Software and Technology* (San Diego, CA, November 2000), ACM Press, pp. 11–20.

- [51] MANKOFF, J., HUDSON, S. E., AND ABOWD, G. D. Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. In *Proceedings of ACM CHI 2000 Conference on Human Factors in Computing Systems* (The Hague, The Netherlands, 2000), ACM Press, pp. 368–375.
- [52] MARX, M., AND SCHMANDT, C. Putting people first: Specifying proper names in speech interfaces. In *Proceedings of the ACM UIST'94 Symposium on User Interface Software and Technology* (Marina del Rey, CA, 1994), Papers: Speech and Sound, ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/uist/192426/p29-marx/p29-marx.pdf> pp. 29–37.
- [53] MASUI, T. An efficient text input method for pen-based computers. In *Proceedings of ACM CHI'98 Conference on Human Factors in Computing Systems* (Los Angeles, CA, 1998), Papers: In Touch with Interfaces, ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/chi/274644/p328-masui/p328-masui.pdf> pp. 328–335.
- [54] MCCOY, K. F., DEMASCO, P., PENNINGTON, C. A., AND BADMAN, A. L. Some interface issues in developing intelligent communications aids for people with disabilities. In *Proceedings of the 1997 International Conference on Intelligent User Interfaces* (1997), Papers: Applications, ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/uist/238218/p163-mccoy/p163-mccoy.pdf> pp. 163–170.
- [55] MCCOY, K. F., DEMASCO, P. W., JONES, M. A., PENNINGTON, C. A., VANDERHEYDEN, P. B., AND ZICKUS, W. M. A communication tool for people with disabilities: Lexical semantics for filling in the pieces. In *First Annual ACM/SIGCAPH Conference on Assistive Technologies* (Marina del Rey, CA, 1994), Augmentative Communication, ACM Press, pp. 107–114.
- [56] MCGEE, D. R., COHEN, P. R., AND OVIATT, S. Confirmation in multimodal systems. In *Proceedings of the International Joint Conference of the Association for Computational Linguistics and the International Committee on Computational Linguistics (COLING-ACL)* (Montreal, Canada, 1998).
- [57] MCKINLAY, A., BEATTIE, W., ARNOTT, J. L., AND HINE, N. A. Augmentative and alternative communication: The role of broadband telecommunications. *IEEE Transactions on Rehabilitation Engineering* 3, 3 (September 1995).
- [58] MEYER, A. Pen computing: A technology overview and a vision. *SIGCHI Bulletin* (July 1995).
- [59] *Microsoft Windows for Pen Computing Programmer's Reference*, 1992.

- [60] MYERS, B. A. A new model for handling input. *ACM Transactions on Information Systems* 8, 3 (July 1990), 289–320.
- [61] MYERS, B. A., AND KOSBIE, D. S. Reusable hierarchical command objects. In *Proceedings of ACM CHI'96 Conference on Human Factors in Computing Systems* (1996), Papers: Development Tools, ACM Press, Available at: <http://www.acm.org/sigchi/chi96/proceedings/papers/Myers/bam.com.htm> pp. 260–267.
- [62] NEBEL, B. How hard is it to revise a belief base? In *Handbook of Defeasible Reasoning and Uncertainty Management Systems, Volume 3: Belief Change*, D. Dubois and H. Prade, Eds. Kluwer Academic Publishers, 1998, Available at: [citeseer.nj.nec.com/nebel96how.html](http://citeseer.nj.nec.com/nebel96how.html) pp. 77–145.
- [63] NELON, G. J. *Context-aware and location systems*. PhD thesis, University of Cambridge, 1998. Available at <http://www.acm.org/sigmobile/MC2R/theses/nelson.ps.gz>.
- [64] Netscape communications corporation. Available at: <http://www.netscape.com> Product Web Page.
- [65] NIGAY, L., AND COUTAZ, J. A generic platform for addressing the multimodal challenge. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems* (Denver, CO, 1995), Papers: Multimodal Interfaces, ACM Press, Available at: <http://www.acm.org/sigchi/chi95/proceedings/papers/lmn.bdy.htm> pp. 98–105.
- [66] OVIATT, S. Mutual disambiguation of recognition errors in a multimodal architecture. In *Proceedings of ACM CHI'99 Conference on Human Factors in Computing Systems* (Pittsburgh, PA, 1999), Papers: Speech and Multimodal Interfaces, ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/chi/302979/p576-oviatt/p576-oviatt.pdf> pp. 576–583.
- [67] OVIATT, S. Multimodal system processing in mobile environments. In *Proceedings of the ACM UIST 2000 Symposium on User Interface Software and Technology* (San Diego, CA, November 2000), ACM Press, pp. 21–30.
- [68] PEDERSEN, E. R., MCCALL, K., MORAN, T. P., AND HALASZ, F. G. Tivoli: An electronic whiteboard for informal workgroup meetings. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems* (Amsterdam, The Netherlands, 1993), Meetings and Collaborative Writing, ACM Press, pp. 391–398.
- [69] PIMENTEL, M., ISHIGURO, Y., ABOWD, G. D., KERIMBAEV, B., AND GUZDIAL, M. Supporting educational activities through dynamic web interfaces. *Interacting with Computers* 13, 3 (February 2001), 353–374.

- [70] POON, A., WEBER, K., AND CASS, T. Scribbler: A tool for searching digital ink. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems* (Denver, CO, 1995), Short Papers: Pens and Touchpads, ACM Press, Available at: [http://www.acm.org/sigchi/chi95/proceedings/shortppr/adp\\_bdy.htm](http://www.acm.org/sigchi/chi95/proceedings/shortppr/adp_bdy.htm) pp. 252–253.
- [71] RHODES, B. J., AND STARNER, T. Remembrance agent. In *The Proceedings of The First International Conference on The Practical Application Of Intelligent Agents and Multi Agent Technology (PAAM '96)* (1996), Available at: <http://rhodes.www.media.mit.edu/people/rhodes/remembrance.html> pp. 487–495.
- [72] ROSENBERG, J., ASENTE, P., LINTON, M., AND PALAY, A. X Toolkits: the lessons learned. In *UIST'90. Third Annual Symposium on User Interface Software and Technology. Proceedings of the ACM SIGGRAPH Symposium* (Snowbird, UT, October 1990), ACM Press, pp. 108–111.
- [73] RUBINE, D. Specifying gestures by example. *Computer Graphics* 25, 4 (July 1991), Available at: <http://www.acm.org:80/pubs/citations/proceedings/graph/122718/p329-rubine/> 329–337.
- [74] RUDNICKY, A. I., AND HAUPTMANN, A. G. Models for evaluating interaction protocols in speech recognition. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems* (New Orleans, LA, 1991), Papers: Special Purpose Interfaces, ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/chi/108844/p285-rudnicky/p285-rudnicky.pdf> pp. 285–291.
- [75] SALBER, D., DEY, A. K., AND ABOWD, G. D. The Context Toolkit: Aiding the development of context-enabled applications. In *Proceedings of ACM CHI'99 Conference on Human Factors in Computing Systems* (Pittsburgh, PA, 1999), Papers: Tools for Building Interfaces and Applications, ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/chi/302979/p434-salber/p434-salber.pdf> pp. 434–441.
- [76] SCHILIT, B. N., ADAMS, N. I., AND WANT, R. Context-aware computing applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, December 1994), IEEE Computer Society.
- [77] SCHMIDT, A., AIDOO, K. A., TAKALUOMA, A., TUOMELA, U., LAERHOVEN, K. V., AND DE VELDE, W. V. Advanced interaction in context. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing (HUC '99)* (Karlsruhe, Germany, September 1999), Springer-Verlag, Available at: [http://www.teco.edu/albrecht/publication/huc99/advanced\\_interaction\\_context.pdf](http://www.teco.edu/albrecht/publication/huc99/advanced_interaction_context.pdf) pp. 89–101.

- [78] SPILKER, J., KLARNER, M., AND GÖRZ, G. Processing self corrections in a speech to speech system. In *Proceedings of COLING'00* (July 2000), ICCL.
- [79] SUHM, B. Empirical evaluation of interactive multimodal error correction. In *IEEE Workshop on Speech recognition and understanding* (Santa Barbara (USA), Dec 1997), Available at: <http://www.cs.cmu.edu/~bsuhm/> IEEE.
- [80] SUHM, B. *Multimodal Interactive Error Recovery for Non-Commercial Speech User Interfaces*. PhD thesis, Carnegie Mellon University, 1998. Brad A. Myers, Advisor.
- [81] SUHM, B., WAIBEL, A., AND MYERS, B. Model-based and empirical evaluation of multimodal interactive error correction. In *Proceedings of ACM CHI'99 Conference on Human Factors in Computing Systems* (Pittsburgh, PA, 1999), Papers: Speech and Multimodal Interfaces, ACM Press, Available at: <http://www.acm.org/pubs/articles/proceedings/chi/302979/p584-suhm/p584-suhm.pdf> pp. 584–591.
- [82] SUKAVIRIYA, P. N., FOLEY, J. D., AND GRIFFITH, T. A second generation user interface design environment: The model and the runtime architecture. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems* (Amsterdam, The Netherlands, 1993), Model-Based UI Development Systems, ACM Press, pp. 375–382.
- [83] Sun microsystems: Java awt™. Available at: <http://java.sun.com> Product Web Page.
- [84] THORISSON, K. R., KOONS, D. B., AND BOLT, R. A. Multi-modal natural dialogue. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems* (Monterey, CA, 1992), ACM Press, pp. 653–654.
- [85] WORDEN, A., WALKER, N., BHARAT, K., AND HUDSON, S. Making computers easier for older adults to use: Area cursors and sticky icons. In *Proceedings of ACM CHI'97 Conference on Human Factors in Computing Systems* (Atlanta, GA, 1997), Papers: Enhancing The Direct Manipulation Interface, ACM Press, Available at: <http://www.acm.org/sigchi/chi97/proceedings/paper/nw.htm>, <http://www.acm.org/pubs/articles/proceedings/chi/258549/p266-worden/p266-worden.pdf> pp. 266–271.
- [86] ZAJICEK, M., AND HEWITT, J. An investigation into the use of error recovery dialogues in a user interface management system for speech recognition. In *Proceedings of IFIP INTERACT'90: Human-Computer Interaction* (1990), Interactive Technologies and Techniques: Speech and Natural Language, pp. 755–760.