

**Proposal of a model and architecture for handling ambiguity
in recognition-based input.**

A Proposal

by

Jennifer Mankoff

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in College of Computing

Georgia Institute of Technology
November 1999

Copyright © 1999 by Jennifer Mankoff

**Proposal of a model and architecture for handling ambiguity
in recognition-based input.**

Approved:

Gregory D. Abowd, Chairman

Scott E. Hudson

Elizabeth Mynatt

Mark Guzdial

Date Approved _____

Preface

Recognition technologies are being used extensively in both the commercial and research world. But recognizers are still error-prone, and this results in performance problems and brittle dialogues, which are a barrier to the acceptance and usefulness of recognition systems. Better interfaces to recognition systems, which can help to reduce the burden of recognition errors, are difficult to build because of lack of knowledge about the ambiguity inherent in recognition. We have extended a user interface toolkit to model and to provide structured support for ambiguity at the input event level. We have populated this toolkit with re-usable interface components for resolving ambiguity. The resulting infrastructure makes it easier for application developers to support error handling, thus helping to reduce the negative effects of recognition errors, and allowing us to explore new types of interfaces for dealing with ambiguity.

Contents

Preface	iii
List of Tables	vi
List of Figures	vii
Acknowledgments	ix
1 INTRODUCTION	1
1.1 Thesis statement	3
1.2 Contributions	4
1.3 Overview of proposal	4
2 BACKGROUND	6
2.1 Mediation: Selecting the correct interpretation of user input	7
2.1.1 Repetition	8
2.1.2 Choice	9
2.1.3 Automatic mediators	12
2.1.4 Deciding how to mediate	14
2.2 Toolkit-level support for recognition-based input	15
2.2.1 Toolkit input handling models	16
2.2.2 Toolkits supporting recognition	17
2.2.2.1 Unimodal toolkits	17
2.2.2.2 Multi-modal toolkits	18
2.3 Conclusions	19
3 UNDERSTANDING AND MODELING AMBIGUITY IN RECOGNITION	20
3.1 A model of ambiguity	22

3.1.1	A mathematical description of ambiguity in recognition	23
3.1.2	Evaluation of model	25
3.2	Conclusions	25
4	TOOLKIT EXTENSIONS	27
4.1	Toolkit details	28
4.2	Mediation details	30
4.3	Hiding ambiguity	31
4.4	Conclusions	32
5	DEMONSTRATION APPLICATIONS	34
5.1	Traditional mediators are relatively easy to build	34
5.1.1	Word-prediction	35
5.1.2	Our architecture supports a broad range of techniques	35
5.1.2.1	Broad range of input types	35
5.1.2.2	Three types of ambiguity	36
5.1.2.3	Types of mediation	36
5.2	Mediation in the face of complexity	36
5.2.1	The current application	37
5.2.2	The extended application	37
5.3	Conclusions	38
6	PLAN	39

List of Tables

1	A comparison of different systems that offer the user a choice of multiple potential interpretations of her input.	10
2	An example of a confusion matrix for the letter e. This matrix only shows the letter e itself and letters that were confused with “e”. So, when the user drew the gesture for “e”, it was recognized correctly in every trial (100% of the time). On the other hand, the gesture for “k” was only recognized correctly 72% of the time, and it was mis-recognized as “e” 8% of the time. Similarly, the gesture for “l” was mis-recognized as “e” 3% of the time. No other letters were mis-recognized as “e”. Thus, we know that when the recognizer returns an “e”, it could be a k or l, but most likely is an e.	14

List of Figures

1	An interface sketched in Burlap (a simplified version of Silk [25]). Is the sketch to the upper left a check box or radio button? Since the recognizer generated both interpretations, the system has brought up a multiple alternative display asking the user which interpretation of her input is correct.	2
2	Target ambiguity: Which radiobutton did the user intend to check?	20
3	Segmentation ambiguity: The user telling Burlap to group 3 radiobuttons together. When grouped, only one button can be selected at a time. The mediator shown has three menu items. “separate” indicates that the radiobuttons will not be grouped. “sequence:2” indicates that the first two radiobuttons will be grouped. “sequence:3” indicates that all three radiobuttons will be grouped.	21
4	Recognition: The user has drawn a box on the screen. This is done with a mouse down, a series of mouse drags, and a mouse up. (a) The input is recognized as a stroke, which is in turn recognized as a rectangle. (b) The input is recognized as a stroke (unambiguously), which is in turn recognized as either a rectangle or a circle (ambiguously). The ambiguous interpretations are shown with cross hatches. Light gray indicates the original user input.	22
5	A sketch in Burlap. The original mouse input, shown in light grey, is interpreted as strokes (unambiguously). The first stroke may be a rectangle or circle (recognition ambiguity). The second is text (unambiguous). The two strokes together may be either a checkbox or a radiobutton (recognition ambiguity). In addition, the strokes may individually be a button and a label. The button and label (highlighted in light grey) represent one way of segmenting the strokes (separately), and the checkbox and radiobutton (highlighted in dark grey) represent a different way (together). This is an example of segmentation ambiguity.	23

6	User input in a simple word-prediction application. As the user types, a recognizer tries to predict which word she is typing. In this figure, a menu of possible predictions is displayed near the user's cursor.	28
---	--	----

Acknowledgments

This work was supported in part by the National Science Foundation under grants IRI-9703384, EIA-9806822, IRI-9500942 and IIS-9800597. 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 [39]. Takeo Igarashi provided the drawing beautifier recognizer, Pegasus, that was also used for demonstrational purposes [21].

Chapter 1

INTRODUCTION

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. Recognition is particularly useful in settings where a keyboard and mouse are not available, such as very large or very small displays, and mobile and ubiquitous computing.

However, recognizers are error-prone, and this can confuse the user, cause performance problems, and result in brittle interaction dialogues. For example, Suhm found that the speed of spoken input to computers is 40 words per minute (wpm) on average because of recognition errors, even though humans speak at 120 wpm [43].

In order to ground our discussion of recognition systems, and to demonstrate the type of interactivity we hope to support, we will use Burlap, the application shown in Figure 1, as a running example throughout this document. Burlap is a simplified version of the sketch-based user interface builder SILK (Sketching Interfaces Like Crazy [24]), built with our toolkit. Like SILK, it allows the user to sketch interactive components (or what we will call interactors) such as buttons, scrollbars, and checkboxes on the screen. Figure 1 shows a sample user interface sketched in Burlap. The user can click on the sketched buttons, move the sketched scrollbar, and so on. For this to work, an (error-prone) recognizer is converting the user's sketches into interactive widgets. In Figure 1, the user has drawn a radiobutton that is easily confused with a checkbox. Rather than simply guessing which interpretation is correct, the system has brought up a menu asking the user which interpretation of her input is correct.

This menu is an example of a choice-based interface, one of two primary interfaces strategies that research has shown reduce some of the negative effects of recognition errors. Both choice and the other strategy, repetition, involve the user in the process of correcting errors. In repetition-based

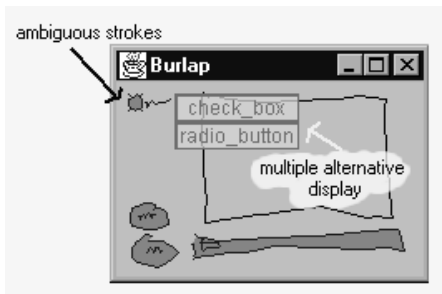


Figure 1: An interface sketched in Burlap (a simplified version of Silk [24]). Is the sketch to the upper left a check box or radio button? Since the recognizer generated both interpretations, the system has brought up a multiple alternative display asking the user which interpretation of her input is correct.

interfaces, the user repeats her input, either in the same or in a different modality. Choice interfaces display multiple potential interpretations of the user’s input, and allow her to select one.

For example, Suhm found that user satisfaction and input speed both increased when he added support for multi-modal error handling to a speech dictation system [43]. Suhm allowed users to repeat or correct their spoken input with pen gestures. Similar results were found in a study of Quickset, a multi-modal map based application [34]. Quickset first looked at input from both speech and pen input in order to combine results, and then let the user choose from among the alternatives [29].

In a broad sense, these interfaces allow the user to help the system pick the correct interpretation of her input. Recognizers will often generate multiple, ambiguous potential interpretations. Computer programs, not normally built to handle ambiguity, may select the wrong alternative. Choice and repetition-based interfaces help to avoid or correct these types of errors.

We can provide re-usable support for choice and repetition-based interfaces like those described in the literature by modeling and providing access to knowledge about the ambiguity resulting from the recognition process. 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.

When ambiguity is modeled explicitly in the user interface toolkit, it becomes accessible to

interface components. This means that components can give the user feedback about how their input is being interpreted before the correct interpretation has been selected. At the same time, by carefully separating and structuring different tasks within the toolkit, neither application nor interface need necessarily know anything about recognition or ambiguity in order to use recognized input, or to make use of predefined interface components for resolving ambiguity.

Because the toolkit handles recognition and ambiguity at the input level, objects can receive recognition results through the same mechanism as mouse and keyboard events. An application can make use of recognized input without having to deal with it directly or to rewrite any interface components. Our approach differs from other toolkits that support recognized input such as Amulet [23] and various multi-modal toolkits [35, 33]. These toolkits provide varying amounts of support for ambiguity, but require the application writer to deal with recognition results directly. For example, consider a word-prediction application [1, 15]. As the user types each character, the system tries to predict what words may come next. Our input system automatically delivers any text generated by the word-predictor to the current text focus, just as it would do with keyboard input. The text focus may not even know that a word-predictor is involved. If there is more than one possible completion (the text is ambiguous), the toolkit will maintain the relationship between the possible alternatives and any interactive components using that input. The toolkit will automatically pass the ambiguous input to the mediation subsystem. This subsystem decides when and how to resolve ambiguity. If necessary, it will involve the user via interfaces, called mediators, that reflect the look and feel of a specific mediation strategy. For example, it may bring up an n -best list, or menu, of alternatives from which the user can select the correct choice.

1.1 Thesis statement

A graphical user interface toolkit architecture that models recognition ambiguity at the input level can make it easier to build effective interfaces to recognition systems. Specifically, we can provide re-usable interfaces for resolving ambiguity in recognition through mediation between the user and computer. In addition, we can apply our model and architecture to handle ambiguity in new, more complex settings.

1.2 Contributions

Our contributions include:

- A model for resolving ambiguity in recognition. We call this process mediation. By building a model of ambiguity and its resolution, we can identify a variety of types of ambiguity and mediation strategies.
- An architecture for integrating this model into a standard GUI toolkit. This architecture has been implemented as an extension of the subArctic toolkit [19, 9]
 - This architecture allows the application to make use of recognized input without having to deal with it directly or to rewrite any interface components. In general, it simplifies the process of building interfaces to recognition systems for application developers while still providing support for ambiguity and mediation.
 - It also makes it possible to build re-usable components and strategies for resolving ambiguity (called mediators). It allows the system to give the user feedback about how her input is being interpreted before it knows for sure exactly which interpretation is correct. At the same time, by carefully separating and structuring different tasks within the extended toolkit, neither application nor interface need necessarily know anything about recognition or ambiguity to use recognized input, or to make use of predefined interface components for resolving ambiguity. This allows us to experiment with a variety of new types of mediators.
- An investigation of mediation in some of the non-traditional settings derived from our model of ambiguity. Specifically, we will investigate ambiguity and mediation in a context-aware In/Out board. These investigations will be supported by a user study comparing different approaches to mediation and their effect on recognition accuracy, user satisfaction, and time to task completion.

1.3 Overview of proposal

Chapter 2 provides an introduction to research in recognition-based interfaces, and identifies two major areas of research that are addressed by this thesis (mediation, and toolkit input models). The

rest of the chapter surveys those areas in more depth. Chapter 3 develops a model of ambiguity and ambiguity resolution in recognition and identifies some common classes of ambiguity. This provides a theoretical framework for the toolkit architecture described in Chapter 4. Chapter 5 gives an overview of how our current and future work addresses the claims made in our thesis. Chapter 6 gives a timetable for completion.

Chapter 2

BACKGROUND

The task of building applications that make use of recognition can be divided into a series of related areas, each of which holds rich possibilities for research. We begin by giving an overview of these research areas. The remainder of this chapter will look in depth at the two major areas addressed by this thesis, mediation and toolkit-level support. Our survey of mediation was done in order to inform the design of the toolkit. It provides motivation and a categorization of the kinds of interfaces that should be supported by our toolkit. The section on toolkit-level support discusses systems that can be directly compared to ours. In particular, because our toolkit provides *input-level support for recognition*, we compare it to existing toolkit input models, and existing toolkits that support recognition.

Recognition First and foremost, these applications depend upon the existence of recognizers.

Recognition involves looking at user input or information sensed 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, word-prediction, and unification in multi-modal computing [35].

Research in recognition involves increasing the range of inputs that can be interpreted, and the range of interpretations that can be created. Recognition is generally a difficult process that may result in errors or may be ambiguous.

Another focus of recognition research is increasing recognition accuracy. The goal is to eliminate errors and ambiguity in recognition. Because ambiguity is inherent in human communication, the holy grail of eliminating errors is probably not achievable. In addition, when recognition technologies are used in noisy, varied environments, accuracy immediately goes down. In practice, researchers try to reduce errors instead. Error reduction is a tough problem, and big improvements (5–10%) before users even notice a difference [6], even for very

inaccurate recognizers (i 50% accuracy).

Mediation The errors that remain must be dealt with by the user of the application. We call the process of correcting and avoiding recognition errors *mediation*, because it generally involves some dialogue between the system and user for correctly identifying the user's intended input. In surveying existing interfaces to recognition systems, we found a plethora of approaches to mediation, described in more detail in Chapter 2, Section 1.

With so many mediation strategies available, it is difficult for designers to know which approach to use in which setting. For example, it does not make sense to check with the user constantly if the recognizer is usually right. Unfortunately, this recognition accuracy, user interruptability, and other context can change dynamically. One solution is to use recognition techniques to handle dynamic changes in the appropriate mediation strategy [17].

Toolkit-level support In order to make experimentation with new mediation strategies easier, we need toolkit-level support. Toolkit-level support also makes it easier for designers who are not experts to take advantage of standard strategies for handling recognition errors. This becomes even more important as recognition technologies are moving into the mainstream (consider the PalmPilotTM, and example of word-prediction in MS WordTM, NetscapeTM, and other applications). For example, the PalmPilotTM has no support at all for mediation. And developers who wish to use Pen for WindowsTM must choose from the one or two strategies provided with Microsoft's toolkit.

2.1 Mediation: Selecting the correct interpretation of user input

Mediation is the process of selecting the correct interpretation of the user's input. For our purposes, correct is defined by the user's intentions. Because of this, 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.

For example, the menu of choices shown earlier in Figure 1 was created by an interactive mediator that is asking the user to indicate whether she drew a radiobutton or a checkbox. It represents a choice-based mediation strategy. Behind the scenes, automatic mediation has already eliminated the possibility that one or more of those strokes was simply intended as an annotation not to be interpreted.

A survey of existing interfaces to recognition systems identifies three basic categories of mediation [26]. The first, and most common, is repetition. In this mediation strategy, the user repeats her input until the system correctly interprets it. The second 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.

2.1.1 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 then dictate the word again. This is the extent of the support for mediation in the original PalmPilotTM. Other devices provide additional support such as an alternative input mode (*e.g.* a soft keyboard), undo of the mis-recognized input, or partial repair of letters within a mis-recognized word.

Modality Repetition often involves a different modality, one that is less error-prone or has orthogonal sorts of errors. For example, in the Newton MessagePadTM, Microsoft Pen for Windows, and other commercial and research applications, the user may correct mis-recognized handwriting by bringing up a soft keyboard and typing the correct interpretation.

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 they modify their speaking voice to be clearer (by human standards) and therefore more difficult for the recognizer to match against normal speech [12]. On the other hand, less error-prone modalities are generally awkward, slow, or otherwise inconvenient (*e.g.* may require hands), or the user would have chosen them in the first place.

There are also examples of alternate modalities that are highly 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 using a pen, rather than bringing up a separate window or dialogue [43].

Undo Depending upon the type of application, and the type of error, repetition may or may not involve undo. For example, repetition is the only possible approach when the recognizer makes an error of omission (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 must undo or delete her input before repeating it (*e.g.* PalmPilotTM, “scratch that” in DragonDictateTM). In some applications, such as Burlap, it does not matter if there are a few extraneous strokes on the screen (they actually add to the “sketched” effect), so undo is unnecessary. In other situations, such as 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?”).

Partial 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.* note that users commonly correct messily written letters in handwriting, and support this even before the recognizer has interpreted the written word [20]. Another researcher from the same lab applied a combination of modality switch, partial repair, and undo to a speech dictation task [43]. He allowed users to cross out or write over mis-recognized letters with a pen.

2.1.2 Choice

Choice user interface techniques gives the user a choice of more than one potential interpretation of her input. One common example of this is an *n*-best list (a menu of potential interpretations) We have identified several dimensions of choice mediation interfaces (also called *multiple alternative displays*) including layout, instantiation time, additional context, interaction, and feedback [27]. Table 1 gives an overview of some commercial and research systems with graphical output that fall

System	Layout	Instantiation	Context	Interaction	Feedback
MessagePad TM [2]	linear menu	on click	original ink	drag-release	ASCII words
DragonDictate TM [8]	linear menu	speech command		speech command	ASCII words
Brennan&Hulteen[5]	speech	on completion	system state	natural language	pos&neg natural language evidence
Goldberg and Goodisman [13]	below top choice	on completion	none	click on choice	ASCII words
Word-prediction (Alm [1] & Greenberg [15])	bottom of screen (grid)	continuously		click on choice	ASCII words
Word-prediction (Netscape [7])	in place	continuously	none	return selects top keystroke, arrow requests more	ASCII words
Marking Menu [22]	pie menu	on pause	none	flick at choice	commands, ASCII letters
Beautification [21]	in place	on completion	constraints	click on choice	pictures (lines)
Remembrance Agent [37]	bottom of screen, linear menu	continuously	certainty, result excerpts	keystroke command	ASCII sentences
UIDE [44]	grid	on command	none	click on choice	thumbnails of results
Lookout [17]	pop up agent speech, ...	on completion	none	click ok	ASCII description of top choice

Table 1: A comparison of different systems that offer the user a choice of multiple potential interpretations of her input.

into this design space. Each system we reference implemented their solutions from scratch, but as Table 1 makes clear, the same design decisions show up again and again.

Layout describes the position and orientation of the alternatives on the screen. The most common layout is a standard linear menu [2, 8, 37]. Other menu-like layouts include a pie menu [22], and a grid [1, 15, 44]. We also found examples of text floating around a central location [13], and drawings in location that the selected sketch will appear [21].

Another variation is to display only the top choice (while supporting interactions that involve other choices) [13].

Instantiation time refers to the time at which the multiple alternative display first appears, and the action that causes it to appear. Variations in when the display is originated include: on a mouse click [2] or other user action such as pause [22] or spoken command [8]; based on an automatic assessment of ambiguity [17], continuously [1, 15, 7, 37]; or as soon as recognition is completed [13, 21].

Contextual information is any information related 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 [37], how they were determined [21], and the original input [2].

Interaction, or the details of how the user indicates which alternative is correct, is generally done with the mouse. For example, Goldberg and Goodisman suggest using a click to select the next most likely alternative even when it is not displayed [13]. Or systems may allow the user to implicitly confirm the indicated choice simply by continuing their task (*e.g.* by drawing a new stroke) [13, 21]. In cases where recognition is highly error-prone, the user must select something to confirm, and can implicitly contradict the suggested interpretation [1, 15, 16]. In non-graphical settings interaction may be done through some other input mode such as speech.

Feedback is the method of displaying the interpretations. This generally correlates closely to how they will look if they are selected. Text is used most often [1, 2, 8, 13, 15, 7, 37], but some interpretations do not map naturally to a text-based representation. Other variations include drawings [21], commands [22], icons [44], and mixtures of these types. In addition, feedback may be auditory. For example, Brennan & Hulteen use natural language to “display” multiple alternatives [5]

Two systems that differ along almost all of the dimensions just described are the Newton MessagePadTM, and the Pegasus drawing beautification system [21]. The Newton MessagePadTM uses a menu layout, which is instantiated when the user double clicks on an incorrectly recognized word. The original handwritten input is displayed at the bottom of the menu and an alternative is selected by clicking the mouse on the choice. This system also supports repetition in an alternate modality (a soft keyboard).

In contrast, the Pegasus drawing beautification system recognizes user input as lines [21]. This allows users to more easily and rapidly sketch geometric designs. Layout is “in place,” i.e. lines are simply displayed in the location they will eventually appear if selected. Instead of waiting for an error to occur, the multiple alternative display is instantiated as soon as recognition is completed. It is essentially informing the user that there is some ambiguity in interpreting her input, and asking

for help resolving it. As in the previous example, the user can select an alternative by clicking on it. However, interaction differs in that the top choice will automatically be selected if the user continues to draw lines. This system also shows the constraints used to generate each choice as additional context.

By identifying this design space, we can begin to see new possibilities. For example, although continuous display of alternatives has been used in text-based prediction such as Netscape's word-prediction and the Remembrance agent [7, 37], to our knowledge it has not been used to display multiple predicted completions of a gesture in progress.

Further research in multiple alternative 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 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.

2.1.3 Automatic mediators

Automatic mediators select an interpretation of the user's input without involving the user at all. Three classes of automatic mediators are commonly found in the literature, and described below.

Thresholding Many recognizers return some measure of the probability that each interpretation is correct. This probability represents the confidence of the interpretation. The resulting probabilities can be compared to a threshold. When an interpretation falls below the threshold, the system rejects it. Many systems use a moving threshold, that is to say they always reject all interpretations except the top choice. Some systems may set the threshold to zero, meaning they do not reject anything (*e.g.*, word-prediction), and other systems try to determine a reasonable static threshold based on known information [36, 5, 3].

Rules Baber and Hone suggest using a rule base to determine which result is correct [3]. This can prove to be more sophisticated than either statistics or thresholding since it allows the use of context. An example rule might be:

When the user has just written ‘for (’, lower the probability of correctness for any alternatives to the next word they write that are not members of the set of variable names currently in scope.

Because rules often depend upon linguistic information, they benefit from knowledge about which words are definitely correct to use as “anchors” in the parsing process.

Historical Statistics When error-prone systems 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 makes mistakes. This task itself benefits from good error discovery. A historical analysis can help to increase the accuracy of both thresholding and rules. For example, Marx and Schmandt compiled speech data about how letters were mis-recognized into a confusion matrix, and used it to generate a list of potential alternatives for whatever the speech recognizer returned [28].

The example in Table 2 shows pen data for “e” from a confusion matrix generated by the author by repeating each letter of the alphabet 25 times in a PalmPilotTM. The first column represents the letter that was written; the other columns show which letters the PalmPilotTM GraffitiTM recognizer returned. Only letters that were mistaken for “e” are shown.

This approach may be used to enhance thresholding or rules. For example, a confusion matrix may be used to update certainty values before applying a threshold. Alternatively, it may be used to add additional possible interpretations before applying a rule or even an interactive method of mediation. We do this in a demo GraffitiTM application that uses a recognizer that only returns one interpretation.

This sort of matrix is called a confusion matrix because it shows potentially correct answers that the system may have confused with its returned answer. In this way, historical statistics may provide a default probability of correctness for a given answer when a recognizer does not. More sophisticated analyses can help in the creation of better rules or the choice of when to apply certain rules.

original	top guess	other guesses
<i>e</i>	<i>e</i> (100%)	
k	k(72%)	l(16%), <i>e</i> (8%), s(4%)
l	l(80%)	c(17%), <i>e</i> (3%)

Table 2: An example of a confusion matrix for the letter *e*. This matrix only shows the letter *e* itself and letters that were confused with “*e*”. So, when the user drew the gesture for “*e*”, it was recognized correctly in every trial (100% of the time). On the other hand, the gesture for “*k*” was only recognized correctly 72% of the time, and it was mis-recognized as “*e*” 8% of the time. Similarly, the gesture for “*l*” was mis-recognized as “*e*” 3% of the time. No other letters were mis-recognized as “*e*”. Thus, we know that when the recognizer returns an “*e*”, it could be a *k* or *l*, but most likely is an *e*.

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 error has occurred. Only when the user’s action is to notify the system explicitly of 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 create a new source of errors, leading to a cascade of errors.

2.1.4 Deciding how to mediate

The interactive and automatic mediators described above represent some very general mediation strategies within which there are a huge number of design choices. The decision of when and how to involve the user in mediation can be complex and application specific. User studies, and other standard HCI methods for gathering qualitative and quantitative data about user interfaces, can be a major source of guidance in interactions with error-prone computer programs

Although it is possible to ask the user direct questions about how they handle errors, this may miss the point since the best error handling happens with as little conscious attention as possible. An alternative is to compare task completion speeds with and without error correction support, and to test for satisfaction and frustration.

Suhm suggests normalizing the data based on the number of errors that occur to compare studies of different interfaces for error correction that can be used in the same application [41]. For systems that generate ASCII, he also devised a way to relate accuracy to words per minute [42].

In addition to helping with the design of better mediators, these studies can help to inform the

task of *meta-mediation* Meta-mediation involves deciding whether to do automatic mediation, and how and when to interrupt the user if automatic mediation is not possible.

The goal of meta-mediation is to minimize the impact of errors and mediation of errors on the user. This is partially an HCI design problem. Given knowledge about the accuracy of the recognizers in use and the user's task, an interface designer can select one or more mediators. However, the accuracy of recognition can change depending upon the user's input and its context. For example, systems tend to misunderstand a subset of possible inputs much worse than the rest, both in the realm of pen input [11] and speech input [28].

Horvitz uses a technique called *decision theory* to provide dynamic, system-level support for meta-mediation [17]. 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 interfaces.

2.2 Toolkit-level support for recognition-based input

Toolkits provide re-usable components and are most useful when a class of common, similar problems exists. Interfaces to error-prone systems would benefit tremendously from a toolkit providing mediators to be used and re-used every time an error-prone situation arose. However, it is not possible simply to add mediators to an existing user interface toolkit like one might add a new type of menu or button. In addition to mediators, a toolkit would need to keep track of multiple potential interpretations, and inform components when they were rejected or accepted. Without this capability the system has to commit to a single interpretation at each stage, throwing away potentially useful data. In addition, a toolkit that supports mediation needs to support recognized input as a first class input type similar to keyboard and mouse. In contrast, most toolkit input models assume a fixed set of input devices that are known in advance. this forces the application developer to handle the recognized input separately, and to find ways to extend existing components to handle recognized input rather than handling it through the same mechanisms as mouse and keyboard events.

After surveying traditional input models in graphical user interface toolkits, we discuss some of the changes made to these model in toolkits that support recognition-based input and mediation. Our

work goes beyond both traditional input models and toolkits that support recognition by integrating support for both recognition and mediation at an input level.

2.2.1 Toolkit input handling models

Toolkit input handling occurs at several levels of abstraction. At the first level of abstraction, we are interested in how the states of input devices, and changes to those states, can be monitored. At the second level of abstraction, we must consider how these low-level state changes are delivered to interfaces, and what syntactic or semantic translations take place.

Historically, one of the earliest proposed models for handling user input was the graphics kernel standard (GKS) model proposed by Foley *et Al.* [10]. This model deals with the first level of abstraction. It was actually more sophisticated in some ways than the current standard. Every time the state of a device changed, an event was created. In addition, the value of a device could be queried or polled, and there was a special form of event delivery called “prompt and reply” in which the user was asked to supply the program with a particular value for some input device. The system would automatically prompt the user for this value (*e.g.* a mouse position or typed word) and notify the interface when the value had arrived.

The most common input handling model in user interface toolkits today is the event-based model. It is essentially a subset of the GKS model. In this model, user input is seen as changes in the state of input devices such as mouse motion and key presses. This input is split into a series of discrete events. No polling is supported.

Where today’s toolkits diverge is at the second level of abstraction. The decision of when and where to deliver these events may be made a range of methods including finite state automata in special input handlers [16, 31], constraints, and pre- and post- conditions.

These approaches are distinguished by whether or how events are translated from lexical occurrences into syntactic or semantic ones. For example, both subArctic [19, 9], and Garnet [31], have special objects that use a state machine to handle input events and then call *methods* that reflect actions to be taken based on the meaning of a series of events. For example, dragging and clicking is handled by a state machine internal to an event handler. The event handler keeps track of mouse press, drag and release events and calls methods on an application or interface component such as `click` or `drag_start`.

Contrast this to models that create higher level events from lower level events. The higher level events are essentially the syntactic or semantic “meaning” of the low-level events. For example, one interpretation of a **mouse-press** followed by a series of **mouse-drags** and a **mouse-release** is a **stroke**. This relationship is reflected in the inheritance hierarchy in object-oriented systems [14, 32].

2.2.2 Toolkits supporting recognition

The input models described above are generally confined to standard keyboard/mouse-based input. However, some toolkit designers have taken the step of providing explicit support for recognizing input. In addition to supporting recognition, some existing toolkits provide support for mediation. At the toolkit level, mediation can be described as the process of resolving ambiguity in recognition. Ambiguity arises when a recognizer returns more than one potential interpretation of the user’s input.

Very few toolkits provide a principled model for dealing with ambiguity, and none integrate this into the toolkit input model shared by on-screen components. Lack of support for ambiguity forces designers to resolve ambiguity earlier than necessary, or to build one-off solutions for retaining information about ambiguity. This lack of support, combined with recognition results not being integrated into the input model, makes it difficult to build re-usable solutions for mediation.

We will split this discussion into a discussion of unimodal and multi-modal toolkit architectures. In general, unimodal architectures have focussed more on the issue of how to handle recognition seamlessly at an input level, while multi-modal toolkits focus on the problem of handling a variety of inputs and dealing with ambiguity.

2.2.2.1 Unimodal toolkits

One of the earliest GUI toolkits to support gesture was the Arizona Retargetable Toolkit (Artkit) [16], a precursor to subArctic [19, 9]. Artkit grouped related mouse events, sent them to a recognition object, and then sent them to components or other relevant objects. Artkit took the important step of integrating the task of calling the gesture recognition engine, into the normal input cycle, an innovation repeated later in Amulet [23]. In addition, objects wishing to receive recognition results did this through the same mechanism as other input results. However, Artkit did not support ambiguity, the recognizer was expected to return a single result.

Support for ambiguity was addressed theoretically in the work of Hudson and Newell on probabilistic state machines for handling input [18]. This work is most applicable to handling visual feedback. For example, if there is uncertainty as to whether the user is selecting checkbox A or checkbox B on a touch screen, a probabilistic state machine would simplify the task of highlighting both buttons. The user could move her finger until only one box was selected. Without support for ambiguity, one, or possibly both, checkboxes would be selected and the user would have to uncheck the wrong one and check the other. This theoretical approach is intended to be integrated into the event handlers that translate input events from lexical into semantic events. However, it does not address how ambiguity should be passed between event handlers, nor how mediation should be supported, both of which are dealt with by our toolkit.

2.2.2.2 Multi-modal toolkits

In addition to toolkit support for building recognition-based interfaces, it is useful to consider toolkit support for handling multi-modal input [34, 33]. 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. Bolt used a combination of speech and pointing with a mouse in his seminal paper on multi-modal input [4]. In contrast, the Quickset system, which uses the Open Agent Architecture, combines speech with gesture recognition and natural language understanding [34].

These toolkits focus on providing support for combining input from multiple diverse sources of input. Unlike the unimodal toolkits described above, the result is generally passed to the application to handle directly rather than integrated into the same input model as mouse and keyboard input as in our toolkit.

Explicit support for ambiguity was addressed in the Open Agent Architecture. Oviatt's work in mutual disambiguation tries to use knowledge about the complementary qualities of different modalities to reduce ambiguity [34]. In addition, McGee, Cohen and Oviatt experimented with different confirmation strategies (essentially interactive mediation) [29]. Although this work includes support for ambiguity, this support is not integrated into a GUI toolkit input model in a transparent fashion. As a result, the application must resolve ambiguity at certain fixed times or explicitly track the relationship between ambiguity and any actions. This makes it hard to build generic, re-usable mediators.

2.3 Conclusions

There is a large variety of interfaces for mediating recognition errors to 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.

However, we found few examples of toolkit-level support for these types of applications, particularly at the graphical user interface level (multi-modal toolkits do not usually deal directly with interface components). Since most of the mediation strategies found in the literature can be placed in one of three categories, repetition, choice, or automatic mediation, there is a lot of potential for providing re-usable support for them.

A key observation about these mediation strategies is that they can be described in terms of ambiguity. They may let the user choose from multiple ambiguous interpretations of her input, or replace one interpretation with a new one. Although a model of ambiguity does not handle errors of omission where the recognizer did not realize that the user was creating new input, it can describe the kinds of errors that happen once input gets to a recognizer. The next chapter describes our model of ambiguity in depth.

Chapter 3

UNDERSTANDING AND MODELING AMBIGUITY IN RECOGNITION

The purpose of this chapter is to explore ambiguity as a way of understanding and modeling error-prone recognition-based input. In particular, ambiguity can describe a situation in which there are multiple potential ways to interpret the user's input. Recognizers will often return multiple interpretations with different probabilities. Until now, all of our examples have dealt with this type of ambiguity, often called *recognition ambiguity*. For example, Burlap depends upon a recognizer that can turn strokes into interactors. The recognizer is ambiguous; it returns multiple guesses in the hope that at least one of them is right and can be selected by the user via some interactive mediation strategy. However, there are other types of ambiguity that can lead to errors that could be reduced by involving the user. In particular, both *target ambiguity* and *segmentation ambiguity* are fairly common.

Target ambiguity arises when the target of the user's input is unclear. For example, the checkmark in Figure 2 crosses two radiobuttons. Which should be selected? Another classic example of target ambiguity comes from the world of multi-modal computing. If the user of a multi-modal



Figure 2: Target ambiguity: Which radiobutton did the user intend to check?

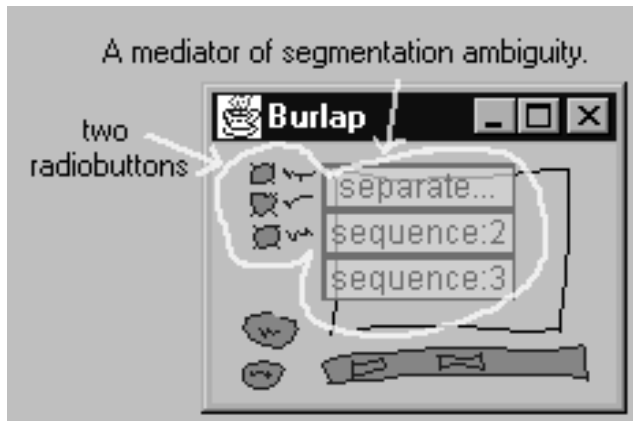


Figure 3: Segmentation ambiguity: The user telling Burlap to group 3 radiobuttons together. When grouped, only one button can be selected at a time. The mediator shown has three menu items. “separate” indicates that the radiobuttons will not be grouped. “sequence:2” indicates that the first two radiobuttons will be grouped. “sequence:3” indicates that all three radiobuttons will be grouped.

systems says, ‘put that there,’ what does ‘that’ and ‘there’ refer to? Target ambiguity also arises when the scope of an input (such as a selection gesture) is unclear, or when on-screen components overlap and the user clicks in the overlapping area.

A third type of ambiguity, segmentation ambiguity, arises when there is more than one possible way to group input events. Did the user really intend the rectangle and squiggle (which represents text) to be grouped as a radiobutton, or was she perhaps intending to draw two separate interactors, a button, and a label? If she draws more than one radiobutton, as in Figure 3, is she intending them to be grouped so that only one can be selected at a time, or to be separate? Should new radiobuttons be added to the original group, or should a new group be created? Similarly, if she writes ‘a r o u n d’ does she mean ‘a round’ or ‘around’? Most systems provide little or no feedback to users about segmentation ambiguity even though it has a significant effect on the final recognition results in domains such as handwriting and speech recognition.

Choosing the wrong possibility from a set of ambiguous alternatives causes a recognition error. Many common recognition errors can be traced to one or more of the three types of ambiguity described above. For example, when a handwritten phrase such as “recognition error” is mis-recognized as, say “resigns in error”, it is a mixture of segmentation and recognition errors that

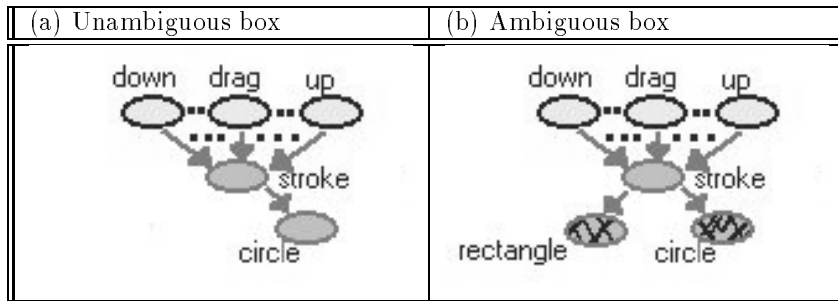


Figure 4: Recognition: The user has drawn a box on the screen. This is done with a mouse down, a series of mouse drags, and a mouse up. (a) The input is recognized as a stroke, which is in turn recognized as a rectangle. (b) The input is recognized as a stroke (unambiguously), which is in turn recognized as either a rectangle or a circle (ambiguously). The ambiguous interpretations are shown with cross hatches. Light gray indicates the original user input.

have caused the wrong result. Yet user interfaces for dealing with errors and ambiguity almost exclusively deal with recognition ambiguity while ignoring segmentation and target ambiguity. Of the systems described in Chapter 2, Section 1, none dealt directly with segmentation ambiguity. In one case, researchers draw lines on the screen to encourage the user to segment their input appropriately, but they give no dynamic feedback about segmentation [13]. None dealt with target ambiguity.

3.1 A model of ambiguity

In our definition, recognition involves looking at user input or information sensed about the user, and interpreting it. Most recognizers generate multiple potential interpretations, ranking them according to the probability that they are correct.

Based on this definition, we build a directed graph which links interpretations to the input from which they are generated. Figure 4(a) shows an example of this. The user has drawn a box on the screen with the mouse. This is done with a mouse down, a series of mouse drags, and a mouse up. That input is recognized as a stroke, which is in turn recognized as a rectangle.

The root nodes of this graph are the user's original inputs. A directed arrow from **A** to **B** means **B** is an interpretation of **A**. This graph is considered ambiguous when two or more interpretations are in conflict. Interpretations are in conflict when they depend upon the same source or upon conflicting sources. For example, in Figure 4(b), the stroke has two interpretations that are in

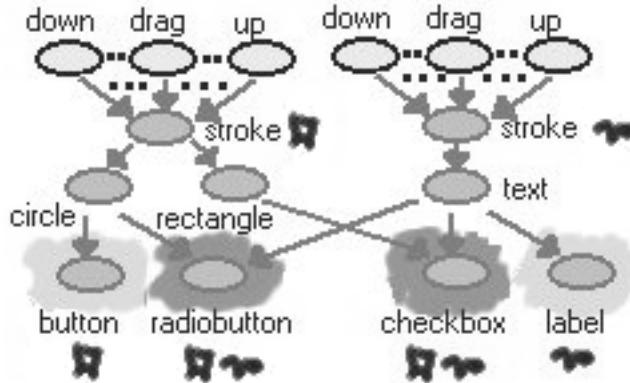


Figure 5: A sketch in Burlap. The original mouse input, shown in light grey, is interpreted as strokes (unambiguously). The first stroke may be a rectangle or circle (recognition ambiguity). The second is text (unambiguous). The two strokes together may be either a checkbox or a radiobutton (recognition ambiguity). In addition, the strokes may individually be a button and a label. The button and label (highlighted in light grey) represent one way of segmenting the strokes (separately), and the checkbox and radiobutton (highlighted in dark grey) represent a different way (together). This is an example of segmentation ambiguity.

conflict, the rectangle and the circle. This is an example of recognition ambiguity.

Figure 5 shows a more complicated example of ambiguity from Burlap. The user is sketching a checkbox. As in Figure 4(b) they start by drawing a box, which they follow with a squiggle indicating text. The first stroke has the same two interpretations. The second stroke has one interpretation, text. An interactor recognizer generates a radiobutton interpretation from the text and the circle, and a checkbox interpretation from the text and the rectangle. These are in conflict, and illustrate recognition ambiguity. In addition, the recognizer has interpreted the circle as a button and the text as a label. Each of these interpretations is in conflict with the checkbox or radiobutton because of segmentation ambiguity. The strokes can either be grouped together or separately, but not both.

3.1.1 A mathematical description of ambiguity in recognition

The following set of equations is a more formal definition of our model of ambiguity. We give the details of this definition in Appendix A. The most important functions and relations are described here.

Let N be the set of all possible inputs and interpretations. Let

$$g : N \leftrightarrow N \tag{1}$$

be a graph representing recognized input, where each node in the graph is either user input or some interpretation of a previous node.

Let

$$\mathbf{conflicts}(g)n = \{n' : N \mid ((\mathbf{ancestors}(g)n' \cap \mathbf{ancestors}(g)n) \neq \emptyset) \wedge (n' \text{ not } \mathbf{open}))\} \tag{2}$$

Where $\mathbf{ancestors}$ is all of the nodes which n is an interpretation of, and all of the ancestors or those nodes.

$$\mathbf{ancestors} : (N \leftrightarrow N) \rightarrow N \rightarrow \mathbb{P}N \tag{3}$$

$$\mathbf{ancestors}(g)n = (g^{-1})^+ \tag{4}$$

If any of the nodes in a graph have a non-empty set of $\mathbf{conflicts}$, those nodes (and therefore the graph) are considered ambiguous.

Whenever a new node n is added to g , that node n is considered \mathbf{open} . As we resolve ambiguity, we progressively \mathbf{close} nodes, either by $\mathbf{accepting}$ or $\mathbf{rejecting}$ them. When we close a node, we also mark whether it was accepted or rejected.

$$\mathbf{accept}(g)n = \mathbf{close}(g)n \wedge \mathbf{accept}(g)(\mathbf{sources}(g)n) \tag{5}$$

$$\mathbf{reject}(g)n = \mathbf{close}(g)n \wedge \mathbf{reject}(g)(\mathbf{interpretations}(g)n) \tag{6}$$

where

$$\mathbf{sources} : (N \leftrightarrow N) \rightarrow N \rightarrow \mathbb{P}N \tag{7}$$

$$\mathbf{sources}(g)n = \{n' : N \mid (n', n) \in g\} \tag{8}$$

and

$$\mathbf{interpretations} : (N \leftrightarrow N) \rightarrow N \rightarrow \mathbb{P}N \tag{9}$$

$$\mathbf{interpretations}(g)n = \{n' : N \mid (n, n') \in g\} \tag{10}$$

3.1.2 Evaluation of model

The relations described in the previous subsection allow us to identify situations that are ambiguous, and to resolve that ambiguity. In addition, the model is powerful enough to allow us to identify two of the three types of ambiguity described above.

Recognition ambiguity occurs when a node n has multiple interpretations (remember that when the interpretations are first created, they are **open** and therefore if there is more than one, they are ambiguous).

We can identify segmentation ambiguity as follows: If there are two nodes n and n' in a graph g , then segmentation ambiguity occurs when

$$\begin{aligned} & ((\mathbf{sources}(g)n \cap \mathbf{sources}(g)n') \neq \emptyset) \\ & \quad \wedge \\ & ((\mathbf{sources}(g)n \cup \mathbf{sources}(g)n') \neq ((\mathbf{sources}(g)n \cap \mathbf{sources}(g)n')))) \end{aligned} \quad (11)$$

Based on this definition, segmentation and recognition ambiguity may involve the same nodes. This is consistent with our examples.

Target ambiguity is harder to define, and we hope to extend this model to effectively handle it. At a system level, target ambiguity occurs when the same event may be dispatched to multiple components. Our model currently contains no information about how interpretations will be used, so it cannot differentiate between target ambiguity and recognition ambiguity.

3.2 Conclusions

By directly modeling recognition, we can build a history of how the user's input was interpreted and identify ambiguous situations, where different interpretations conflict. Our model encompasses recognition ambiguity, which is the kind of ambiguity normally handled by mediators found in the literature. However, it also allows us to identify an additional type of ambiguity which it may be useful to mediate: segmentation ambiguity. We hope to expand it to include target ambiguity as well.

This model also provides a way to define how ambiguity can be resolved. And it can guide us in developing toolkit-level support for ambiguity. It is not hard to see how one might represent a

hierarchical graph of events in an object-oriented system. It is possible to determine the presence of different types of ambiguity by looking at the graph structure, and we have defined functions on the graph for resolving ambiguity by rejecting and accepting events. The following chapter describes how we have integrated this model of ambiguity into the input system of an existing graphical user interface toolkit, subArctic [19, 9].

Chapter 4

TOOLKIT EXTENSIONS

We based our toolkit architecture on the model of ambiguity described in the preceding chapter. We extended an existing user interface toolkit to provide explicit support for tracking and resolving ambiguity (subArctic [19, 9]). This allowed us to make information about ambiguity accessible to interface components. We made our extensions at the input level, by extending the concept of hierarchical events [14, 32] to model ambiguity, and by providing explicit support for mediation (resolving ambiguity). The result is that interface components receive recognition results through the same mechanism as mouse and keyboard events. Since the toolkit dispatches input events when they are still ambiguous, consumers are expected to treat events as tentative, and to provide feedback about them with the expectation that they may be rejected. In particular, we require that feedback related to events be separated from the application actions resulting from those events.

This means that interface components that know about ambiguity can give the user early feedback about what may be done with the recognized input before the correct interpretation is known. By separating feedback about events from action on those events, we can support more flexible strategies for resolving ambiguity because we do not have to resolve ambiguity as soon as it arises. For example, in Burlap, we use lazy mediation and wait to mediate a sketched widget until the user tries to interact with it.

On the other hand, traditional interface components such as the text area in Figure 6, do not need to know about ambiguity at all. This means that application designers can use existing components and third party software without rewriting everything, while still taking advantage of our support for ambiguity. From the perspective of traditional components such as the text area in Figure 6, input from a recognizer and the keyboard look identical. Not only does the text area receive text events correctly regardless of their source, but those events are automatically passed to the mediation subsystem to handle ambiguity. This is possible because ambiguity is handled by the mediation

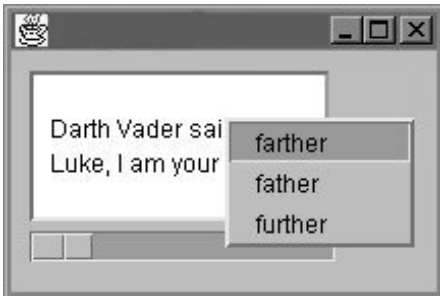


Figure 6: User input in a simple word-prediction application. As the user types, a recognizer tries to predict which word she is typing. In this figure, a menu of possible predictions is displayed near the user’s cursor.

subsystem, separately from the event dispatch cycle.

The mediation subsystem consists of a set of mediators, representing different mediation strategies. These mediator objects may ignore ambiguity (of types they are not designed to handle), directly resolve ambiguity (automatically, or via user interaction), or defer resolution of ambiguity until more information is available (or they are forced to make a choice). Ambiguous events are automatically identified by the toolkit and sent to the mediation subsystem. New mediators can be added by application components, or default mediators can be used.

4.1 Toolkit details

As described in Chapter 2, Section 2.1, existing user interface toolkits handle input from traditional input sources by breaking it into a stream of autonomous input events. Most toolkits dynamically gather a series of eligible objects that may consume (use) each event [10]. The toolkit will normally dispatch (deliver) an event to each such eligible object in turn until the event is consumed. After that, the event is not dispatched to any additional objects. Variations on this basic scheme show up in a wide range of user interface toolkits [19, 31, 30, 38].

In order to extend this model to handle recognized input, we first need to allow the notion of an input source to be more inclusive than in most modern user interface toolkits. Traditionally only keyboard and mouse (or pen) input are supported. We allow input from any source that can deliver a stream of individual events. In particular, we have made use of input from a speech recognizer

(each time text is recognized, a new event is generated) and from a context toolkit [40]. The context toolkit gathers information from sensors and interpreting, or abstracts it to provide information such as the identity of users in a certain location (*e.g.*, near the interface).

The next step in supporting recognized input requires us to track explicitly the interpretation and use of input events. This is done using hierarchical events, which have been used in the past in a number of systems (see for example Green's survey [14] and the work by Myers and Kosbie [32]). Hierarchical events contain information about how traditional input events (mouse, or pen and keyboard) are used, or in the case of recognition, interpreted. This is a translation of the model described in the previous chapter, and illustrated by the graphs in Figures 4 and 5, into an object-oriented architecture. Nodes in a graph represent events. Arrows into a node show what events it was derived from. Arrows out of a node show what events were derived from it. This event hierarchy allows us to model explicitly the relationship between raw events (such as mouse events), intermediate values (such as strokes), derived values (such as recognition results) and what the application does with those values (such as create a radiobutton).

Hierarchical events are generated by requiring each event consumer to return an object containing information about how it interpreted the consumed event [32]. For example, the circle in Figure 5 was created by a gesture recognizer that consumed a stroke and returned the circle as its interpretation of that stroke. Each derived interpretation becomes a new node in the graph. Nodes are dealt with just like raw input events: they are dispatched to consumers, which may derive new interpretations, and so on.

The changes we have described so far support recognized input in the absence of ambiguity. But, as we have shown, ambiguity is an inherent part of recognition. Hierarchical events can be used to model ambiguous, recognized input as in Figures 4(b) and 5. Just as in the unambiguous case, an ambiguous hierarchy is generated as events are dispatched and consumed. However, unlike most existing systems, an event is dispatched to each consumer in sequence even after the event has been consumed. If an event is consumed by more than one consumer, or a consumer generates multiple interpretations, ambiguity results.

We dispatch every new event (node in the graph) even if it is ambiguous. This is critical, because it allows us to defer the process of resolving ambiguity while giving the user early feedback about how events will be used if they are chosen. Consumers are expected to treat events as tentative,

and to provide feedback about them with the expectation that they may be rejected. In particular, we require that feedback related to events be separated from the application actions resulting from those events. For example, in Figure 6, the system is providing feedback about how the user’s input is being interpreted. If one of the words displayed is chosen, the system will act on that selection by inserting the word into the text entry window.

When ambiguity arises, it is resolved by the mediation subsystem of our extended toolkit. The mediation subsystem consists of a set of objects, called mediators, representing different mediation strategies. These mediator objects may ignore ambiguity (of types they are not designed to handle), directly resolve ambiguity, or defer resolution of ambiguity until more information is available (or they are forced to make a choice). The separation of feedback from action makes it possible to defer decisions about ambiguity when appropriate. This is important for providing robust interaction with ambiguous inputs. (As described later, a mechanism is also provided for existing components that are not ambiguity-aware to operate without this separation, and hence without change).

4.2 Mediation details

Because ambiguity arises when events are in conflict, it can be resolved by accepting one of the possible conflicting events, and rejecting the others. The accepted event is correct if it is the interpretation that the user intended the system to make. Otherwise it is wrong. It is the job of the mediation subsystem (described in the next section) to resolve ambiguity by deciding which events to accept or reject.

Ambiguous events are automatically identified by the toolkit and sent to the mediation subsystem. This system keeps an extensible list of mediators. The system provides some default mediators and inserts them into the list in order of priority. The application designer can add additional mediators to appropriate places in the list. The closer to the front of the list a mediator is, the earlier it will get a chance to mediate. When an ambiguous event arrives, the system passes that event to each mediator in turn until the ambiguity is resolved. At that point, no further mediators see the event.

If a mediator is not interested in the event, it simply returns PASS. For example, the n -best list mediator in Figure 3 only deals with segmentation ambiguity in radiobuttons. It simply ignores other

types of ambiguity by returning PASS. If a mediator resolves part of the event graph by accepting one interpretation and rejecting the rest, it returns RESOLVE. Finally, if a mediator wishes to handle ambiguity, but defer final decision on how it will be resolved, it may return PAUSE. Each mediator is tried in turn until one signifies that it will handle mediation by returning PAUSE or RESOLVE and the complete event graph is no longer ambiguous. The toolkit provides a default mediator which will always resolve an ambiguous event (by taking the first choice at each node). This guarantees that every ambiguous event will be mediated.

Not all mediators have user interfaces. Sometimes mediation is done automatically. In this case, errors may result (the system may select a different choice than the user would have selected). An example of a mediator without a user interface is the stroke pauser. The stroke pauser returns PAUSE for the mouse events associated with partially completed strokes, and caches those events until the mouse button is released. It then allows mediation to continue, and each cached mouse event is passed to the next mediator in the list. Feedback about each stroke or partial stroke still appears on the screen since they are delivered to every interested component before mediation. However, no actions are taken until the mouse is released (and mediation completed). At that point, components are told which of the events that they received were accepted or rejected.

Interactive mediators are treated no differently than automatic mediators. The mediators shown in Figure 1 and Figure 3 get events after the stroke pauser. When they see an event with ambiguity they can handle, they add the menu shown to the interface. They then return PAUSE for that event. Once the user selects an alternative, they accept the corresponding choice; and mediation continues.

The toolkit enforces the simple rules described in Chapter 3 about events that are accepted or rejected. All interpretations of rejected events are also rejected. All sources of accepted events are also accepted. Further, any event that conflicts with an accepted event is rejected.

4.3 Hiding ambiguity

We have described an architecture that makes information about ambiguity explicit and accessible. At the same time, we do not wish to make the job of dealing with ambiguity onerous for the application developer. The solution is to be selective about how and when we involve the application. This is possible because we separate the task of mediation from the application and from other parts

of the toolkit. Mediation, which happens separately, determines what is accepted or rejected. Note that mediation can be integrated into the interface even though it is handled separately in the implementation.

It is sometimes too costly to require interactors (components such as buttons and menus) to handle ambiguity and the corresponding separation of feedback and action. Some things, such as third party software, or large existing interactor libraries, may be hard to rewrite. Also, in certain cases, a rejected event may break user expectations (*e.g.*, a button which depresses, yet in the end does nothing).

We provide two simple abstractions that handle these cases. An interactor can be first-dibs, in which case it sees events first, and if it consumes an event no one else gets it. Thus ambiguity can never arise. Alternatively, a last-dibs interactor only gets an event if no one else wants it, and therefore is not ambiguous. Again, ambiguity can not arise. These abstractions can handle those situations where traditional interactors are mixed with interactors that are aware of ambiguity. This allows conventional interactors (*e.g.*, the buttons, sliders, checkboxes, *etc.* of typical graphical user interfaces) to function in the toolkit as they always have, and to work along side our extended interactors when that is appropriate. This property is important, because we do not wish to force either interface developers, or users, to completely abandon their familiar interfaces to make use of recognition technology.

4.4 Conclusions

We have shown how the model of ambiguity described in Chapter 3 can be used to develop a toolkit architecture. This architecture extends the concept of hierarchical events. In addition, it tracks the relationship between events that are dispatched to user interface components and ambiguity, and notifies components when ambiguity is resolved. Ambiguity resolution is handled by a special subsystem for mediation. This subsystem is automatically invoked when ambiguity is present.

In general, we have built this toolkit with the goal of impacting the designer as little as possible while still supporting the standard solutions for handling ambiguity found in our literature survey. At the same time, we provided the appropriate hooks so that a designer wishing to experiment with more novel approaches to mediation can easily do so.

The next chapter discusses applications and mediators we have built and plan to build in order to validate the effectiveness of our toolkit architecture.

Chapter 5

DEMONSTRATION APPLICATIONS

We will demonstrate our toolkit architecture in two ways. First, we will show that we can make it *easier* to do things that have been done in the past, by implementing examples from our survey of interfaces to existing applications using recognition. Second, we will show *breadth of coverage*. We will go beyond examples from the literature to build an example of mediation in a new setting. To this end, we plan to build a context-aware In/Out board as a testbed for exploring these design decisions.

The next section describes how we have covered a broad range of traditional mediators and made it easier to use them. This is all about repeating the past. Section 5.2 talks about support for mediation in complex, non-traditional settings, and the demonstration application that we plan to build.

5.1 Traditional mediators are relatively easy to build

We define “traditional” in terms of the survey of background work presented in Chapter 2, which tells us that interfaces to recognition systems must support mediation, and that certain types of mediators (specifically *n*-best lists and repetition) are appropriate for a broad range of tasks. This suggests that we should provide defaults that support those types of mediation when the designer makes use of ambiguous input sources.

We validate this claim by showing that with minimal changes to an existing interface, a designer can add recognition-based input and include support for standard types of mediation.

In particular, we have taken a simple application for editing text and added a word-predictor and speech recognizer to it. Changes to the application involved only 2 lines of code. This assumes that the recognizers already exist and that a wrapper for each recognizer is provided with the toolkit.

We provide a default mediator for multiple text alternatives which the designer must instantiate.

5.1.1 Word-prediction

As stated above, we have implemented a simple demonstration application that makes use of word-prediction. Although this simple application demonstrates automatic support for mediation, we feel that our argument is more compelling in a real-world setting. To this end, we will demonstrate automatic support for mediation in a word-prediction application for use by people with disabilities. We have begun work on a re-usable multiple alternative display that provides support for individually varying the dimensions described in the review of choice-based mediation given in Chapter 2, Section 1.2 This allows us to also demonstrate the advantages of being able to easily experiment with a variety of mediation schemes. This application will be used to help train patients with severe motion limitations using neural implants to control a mouse cursor and select letters from an on-screen keyboard. It is most similar to work done by Alm *et al.* [1] and Greenberg *et Al.* [15].

5.1.2 Our architecture supports a broad range of techniques

We will show that our toolkit-level solution supports breadth of mediation in two orthogonal ways. First, we support a broad range of input types including dictation by pen and speech, pen commands, drawings, and contextual input. Second, in addition to mediating recognition ambiguity, we also mediate target and segmentation ambiguity.

5.1.2.1 Broad range of input types

In addition to many small applications, we have built one larger application to date. Burlap is a simplified version of the sketch-based user interface builder, SILK (Sketching Interfaces Like Crazy [24]). Burlap, like SILK, allows the user to sketch interactive components (or what we will call interactors) such as buttons, scrollbars, and checkboxes on the screen. Figure 1 shows a sample user interface sketched in Burlap. The user can click on the sketched buttons, move the sketched scrollbar, and so on. In this example, the user has drawn a radiobutton, which is easily confused with a checkbox. The system has brought up a mediator asking the user which interpretation is correct.

As described above, we also have demonstrations of input from a word-predictor (which we plan

to expand). In addition, we plan to build mediation of context into the In/Out board being used on a daily basis by members of this research group (described below in more detail).

Burlap includes input: from speech recognition (IBM ViaVoice); gesture recognition (3rd party software [25]); and context identity (also from 3rd party software [40]). In past work with a previous version of our toolkit, we built a simple application that supported drawing beautification [21].

5.1.2.2 Three types of ambiguity

Burlap includes examples of all three types of ambiguity. We have several examples of mediators for recognition ambiguity, and one for segmentation ambiguity. We plan to build an example mediator for target ambiguity into Burlap as well. Our other applications currently demonstrate recognition ambiguity exclusively.

5.1.2.3 Types of mediation

Burlap already includes choice-based mediators for both segmentation ambiguity and recognition ambiguity. We plan to add a mediator for target ambiguity. Burlap also contains several automatic mediators. Burlap supports repetition-based error correction in the case of errors of omission. We plan to add a re-usable mediator for repetition.

5.2 Mediation in the face of complexity

One true test of the effectiveness of this toolkit is whether it can be used in a complex application that combines input from many different recognizers and uses a variety of mediation strategies in concert.

We believe that issues of ambiguity and mediation arise in a much broader range of settings than are normally associated with recognition. In addition to speech and pen dictation, command, and multi-modal applications, we feel that any application that is trying to interpret information sensed about the user is subject to recognition errors and ambiguity.

One good example of this is ubiquitous computing in general and context-aware computing in particular. Context-aware applications try to take actions on the user's behalf based on information about identity, location, time, and activity sensed from the user's environment. But sensors, and

the task of correctly interpreting them, are error-prone and when we try to fuse data from multiple sensors the errors and ambiguity multiply.

We plan to apply these heuristics to a particular application, a context-aware In/Out board that is in daily use by our research group and other people in our building.

5.2.1 The current application

Currently the In/Out board currently senses user presence through one semi-accurate sensor, an IbuttonTM dock. The application interprets sensor data from this dock to determine when users enter or leave the building. This is ambiguous because users do not always remember to dock in or out, so it is not always clear when they dock whether they are going or coming.

However, the current application does no mediation, and the infrastructure it is based on, the context toolkit [40], contains no support for ambiguity or mediation.

Our first task is to do a simple study of the In/Out board to determine how accurate the current system is. This involves using a motion detector to timestamp and snap a picture of each person who enters or leaves. We will also use a questionnaire to determine user satisfaction with the system.

5.2.2 The extended application

We plan to add a mediator to the application to improve its overall accuracy both through additional sensing and by requesting more user input when necessary. Because the user is moving as sensing goes on, we will need to distribute this interface over space. A motion detector will be added to the system to facilitate this.

When someone crosses the motion sensor, we will attempt to infer who they are, and whether they are coming or going, from known calendar data. We will use spoken output and input to mediate this guess depending upon its predicted accuracy. Only if we are unable to determine the user's identity and activity will we request that they dock at the Ibutton.

We will need to re-implement much of the architecture described in this document to handle a distributed setting before we can build the extended application. Our research so far shows that this can be done with minimal changes to our model of ambiguity and our architecture for resolving ambiguity.

We will test the completed application in the same way as we plan to test the original application.

5.3 Conclusions

This thesis makes two main claims. First, that our architecture makes it relatively easy to build traditional mediators. Second, that our architecture supports a broad range of input types and ambiguity, and thus, that it allows us to experiment with mediation in complex, non-traditional settings.

We have already demonstrated the most of the first claim, and we will finish this work off by building a word-prediction system for use by a patient with disabilities.

We plan to demonstrate the second claim by building and testing context-aware In/Out board. An initial prototype with no mediation already exists. We plan to scale this up to include many more sensors and to test out a variety of mediators.

Chapter 6

PLAN

Recall the thesis statement:

By building an input level model of ambiguity and mediation, we can provide toolkit-level support for interfaces for handling recognition errors and ambiguity. Our solution makes it easier to build interfaces for recognition systems by providing a re-usable solutions for handling recognition-based input. In addition, it allows us to handle a broad range of ambiguous inputs, and to experiment with new types of mediators and new settings for mediation.

So the claims are:

1. An input level model of ambiguity allows us to identify a variety of types of ambiguity and mediation strategies and build a toolkit (validated by the existence of the toolkit, and traditional mediators built in the toolkit). DONE
2. Our toolkit provides re-usable solutions for mediation (validated by writing one mediator and using it in several applications). DONE for multiple alternative displays. Do for repetition, meta-mediation: May 2000
3. The toolkit and the re-usable mediators make it easier to build standard mediators for recognition systems (validated by building some simple applications that depend on the re-usable mediators. Also validated by building a standard type of mediated application, for this we use the word-predictor/training interface for Melody, Word-prediction: December 2000.
4. The toolkit lets us handle a broad range of ambiguous inputs. Broad includes both sources and classes of ambiguity as defined in the section on Ambiguity. (validated by having a broad

range of examples including: burlap, and simple word-prediction (DONE); the context-aware In/Out board application (August 2000)).

Bibliography

- [1] N. Alm, J. L. Arnott, and A. F. Newell. Prediction and conversational momentum in an augmentative communication system. *Communications of the ACM*, 35(5):46–57, May 1992. Referenced on pages: 3,10,11,35
- [2] I. Apple Computer. The Newton MessagePadTM. Referenced on pages: 10,11
- [3] C. Baber and K. S. Hone. Modelling error recovery and repair in automatic speech recognition. *International Journal of Man-Machine Studies*, 39(3):495–515, 1993. Referenced on pages: 12,13
- [4] R. A. Bolt. “Put-That-There”: Voice and gesture at the graphics interface. *Computer Graphics*, 14(3):262–270, July 1980. Referenced on pages: 18
- [5] S. E. Brennan and E. A. Hulteen. Interaction and feedback in a spoken language system: A theoretical framework. *Knowledge-Based Systems*, 8(2-3):143–151, 1995. Referenced on pages: 10,11,12
- [6] R. V. Buskirk and M. LaLomia. The just noticeable difference of speech recognition accuracy. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 2 of *Interactive Posters*, page 95, 1995. Referenced on pages: 6
- [7] N. C. Corporation. <http://www.netscape.com>. Web Page. Referenced on pages: 10,11,12
- [8] DragonDictate. <http://www.dragonsystems.com/products/dragondictate>. Product Web page. Referenced on pages: 10,11
- [9] W. K. Edwards, S. E. Hudson, J. Marinacci, R. Rodenstein, I. Smith, and T. Rodrigues. Systematic output modification in a 2D UI toolkit. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST-97)*, pages 151–158, New York, Oct. 14–17 1997. ACM Press. Referenced on pages: 4,16,17,26,27

- [10] J. D. Foley, V. L. Wallace, and P. Chan. The human factors of computer graphics interaction techniques. *IEEE Computer Graphics and Applications*, 4(11):13–48, Nov. 1984. Referenced on pages: 16,28
- [11] C. Frankish, R. Hull, and P. Morgan. Recognition accuracy and user acceptance of pen interfaces. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 1 of *Papers: Pen Interfaces*, pages 503–510, 1995. Referenced on pages: 15
- [12] C. Frankish, D. Jones, and K. Hapeshi. 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):797–816, 1992. Referenced on pages: 8
- [13] D. Goldberg and A. Goodisman. STYLUS user interfaces for manipulating text. In *Proceedings of the ACM Symposium on User Interface Software and Technology — UIST'91*, pages 127–135. ACM Press, 1991. Referenced on pages: 10,11,22
- [14] M. Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986. Referenced on pages: 17,27,29
- [15] S. Greenberg, J. J. Darragh, D. Maulsby, and I. H. Witten. *Extra-ordinary Human-Computer Interaction: interfaces for users with disabilities*, chapter Predictive Interfaces: what will they think of next?, pages 103–139. Cambridge series on human-computer interaction. Cambridge University Press, New York, 1995. Referenced on pages: 3,10,11,35
- [16] T. R. Henry, S. E. Hudson, and G. L. Newell. Integrating gesture and snapping into a user interface toolkit. In ACM, editor, *UIST. Third Annual Symposium on User Interface Software and Technology. Proceedings of the ACM SIGGRAPH Symposium, Snowbird, Utah, USA, October 3–5, 1990*, pages 112–122, New York, NY 10036, USA, Oct. 1990. ACM Press. Referenced on pages: 11,16,17
- [17] E. Horvitz. Principles of mixed-initiative user interfaces. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, volume 1, pages 159–166, 1999. Referenced on pages: 7,10,15

- [18] S. E. Hudson and G. L. Newell. Probabilistic state machines: Dialog management for inputs with uncertainty. In *Proceedings of the ACM Symposium on User Interface Software and Technology, Toolkits*, pages 199–208, 1992. Referenced on pages: 18
- [19] S. E. Hudson and I. Smith. Supporting dynamic downloadable appearances in an extensible UI toolkit. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST-97)*, pages 159–168, New York, Oct. 14–17 1997. ACM Press. Referenced on pages: 4,16,17,26,27,28
- [20] W. Huerst, J. Yang, and A. Waibel. Interactive error repair for an online handwriting interface. In *Proceedings of ACM CHI 98 Conference on Human Factors in Computing Systems (Summary)*, volume 2 of *Student Posters: Interaction Techniques*, pages 353–354, 1998. Referenced on pages: 9
- [21] T. Igarashi, S. Matsuoka, S. Kawachiya, and H. Tanaka. Interactive beautification: A technique for rapid geometric design. In *Proceedings of the ACM Symposium on User Interface Software and Technology, Constraints*, pages 105–114, 1997. Referenced on pages: 10,11,36
- [22] G. Kurtenbach and W. Buxton. User learning and performance with marking menus. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, volume 1 of *Pen Input*, pages 258–264, 1994. Referenced on pages: 10,11
- [23] J. A. Landay and B. A. Myers. 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, Short Papers (Posters): Interaction Techniques I*, pages 91–92, 1993. Referenced on pages: 3,17
- [24] J. A. Landay and B. A. Myers. Interactive sketching for the early stages of user interface design. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 1 of *Papers: Programming by Example*, pages 43–50, 1995. Referenced on pages: 2,35
- [25] A. C. Long, Jr., J. A. Landay, and L. A. Rowe. Implications for a gesture design tool. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, volume 1 of *Alternatives to QWERTY*, pages 40–47, 1999. Referenced on pages: 1,36

- [26] J. Mankoff and G. D. Abowd. Error correction techniques for handwriting, speech, and other ambiguous or error prone systems. Technical Report GIT-GVU-99-18, Georgia Tech GVU Center, June 1999. Referenced on pages: 8
- [27] J. Mankoff, G. D. Abowd, and S. E. Hudson. Interacting with multiple alternatives generated by recognition technologies. Technical Report GIT-GVU-99-26, Georgia Institute of Technology GVU Center, 1999. Referenced on pages: 9
- [28] M. Marx and C. Schmandt. Putting people first: Specifying proper names in speech interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology — UIST'94*, pages 30–37. ACM Press, 1994. Referenced on pages: 13,15
- [29] D. R. McGee, P. R. Cohen, and S. Oviatt. 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, Quebec, Canada, 1998. Referenced on pages: 2,6,18
- [30] S. Microsystems. Java awtTM. Available at <http://java.sun.com>. Referenced on pages: 28
- [31] B. A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990. Referenced on pages: 16,28
- [32] B. A. Myers and D. S. Kosbie. Reusable hierarchical command objects. In *Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems*, volume 1 of *PAPERS: Development Tools*, pages 260–267, 1996. Referenced on pages: 17,27,29
- [33] L. Nigay and J. Coutaz. A generic platform for addressing the multimodal challenge. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 1 of *Papers: Multimodal Interfaces*, pages 98–105, 1995. Referenced on pages: 3,18
- [34] S. Oviatt. Mutual disambiguation of recognition errors in a multimodal architecture. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, volume 1 of *Speech and Multimodal Interfaces*, pages 576–583, 1999. Referenced on pages: 2,18

- [35] S. L. Oviatt, D. McGee, M. Johnston, J. Clow, P. Cohen, and C. Slattery. Robust functioning through mutual disambiguation in a multimodal system architecture. *Journal Manuscript*, In Submission. Referenced on pages: 3
- [36] A. Poon, K. Weber, and T. Cass. Scribbler: A tool for searching digital ink. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 2 of *Short Papers: Pens and Touchpads*, pages 252–253, 1995. Referenced on pages: 12
- [37] B. J. Rhodes and T. Starner. Remembrance agent. In *The Proceedings of The First International Conference on The Practical Application Of Intelligent Agents and Multi Agent Technology (PAAM '96)*, pages 487–495, 1996. Referenced on pages: 10,11,12
- [38] J. Rosenberg, P. Asente, M. Linton, and A. Palay. X toolkits: the lessons learned. In ACM, editor, *UIST. Third Annual Symposium on User Interface Software and Technology. Proceedings of the ACM SIGGRAPH Symposium, Snowbird, Utah, USA, October 3–5, 1990*, pages 108–111, New York, NY 10036, USA, Oct. 1990. ACM Press. Referenced on pages: 28
- [39] D. Rubine. Specifying gestures by example. volume 25, pages 329–337, July 1991.
- [40] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In M. G. Williams, M. W. Altom, K. Ehrlich, and W. Newman, editors, *Proceedings of the Conference on Human Factors in Computing Systems (CHI-99)*, pages 434–441, New York, 1999. ACM Press. Referenced on pages: 29,36,37
- [41] B. Suhm. Empirical evaluation of interactive multimodal error correction. In *IEEE Workshop on Speech recognition and understanding*, Santa Barbara (USA), Dec 1997. IEEE. Referenced on pages: 14
- [42] B. Suhm, B. Myers, and A. Waibel. Designing interactive error recovery methods for speech interfaces. In *CHI 96 Workshop on Designing the User interface for Speech Recognition applications*. SIGCHI, 1996. Referenced on pages: 14
- [43] B. Suhm, A. Waibel, and B. Myers. Model-based and empirical evaluation of multimodal interactive error correction. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, volume 1 of *Speech and Multimodal Interfaces*, pages 584–591, 1999. Referenced on pages: 1,2,9

- [44] P. N. Sukaviriya, J. D. Foley, and T. Griffith. 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*, Model-Based UI Development Systems, pages 375–382, 1993. Referenced on pages: 10