Monte Carlo Methods for Managing Uncertain User Interfaces

JULIA SCHWARZ

December, 2014
CMU-HCII-14-111

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

THESIS COMMITTEE

Scott Hudson (Co-Chair), Carnegie Mellon University
Jennifer Mankoff (Co-Chair), Carnegie Mellon University
Niki Kittur, Carnegie Mellon University
Andy Wilson, Microsoft Research

Submitted in partial fulfillment of requirements for the degree of Doctor of Philosophy.

This work was supported by the National Science Foundation and a Microsoft PhD Fellowship. Any findings or recommendations expressed in this document are the opinions of the author and do not necessarily reflect the views of the sponsoring organizations.

Keywords

Human-Computer interaction, User Interfaces, User Interface Systems, Probabilistic Modeling, Probabilistic User Interface Systems.

ABSTRACT

Current user interface toolkits provide effective techniques for acting on user input. However, many input handling systems make the assumption that all input events are certain, and are not built to handle ambiguity such as multiple possible inputs from a recognizer. This is unfortunately at odds with recent interaction trends towards voice, gesture and touch, all of which come with a great deal of uncertainty.

This dissertation presents a new user interface architecture that treats user input as an uncertain process, approximates the probability distribution over possible interfaces using Monte Carlo sampling, and enables interface developers to easily build probabilistic user interfaces without needing to think probabilistically. This architecture is embodied in the JULIA toolkit: a JavaScript User interface Library for tracking Interface Alternatives. To demonstrate the versatility and power of this architecture, the dissertation presents a collection of applications and interaction techniques built using the JULIA toolkit. This architecture provides the foundation for a new era of nondeterministic user interfaces that leverage probabilistic models to better infer user intent.

TABLE OF CONTENTS

1 INTRODUCTION

Current user interface toolkits provide effective techniques for modeling, tracking, interpreting, and acting on user input. However, many modern input handling systems make the assumption that all input events have precise properties. This type of input handling works very well for input mechanisms such as keyboards and mice, however it works less well when the input to the system is less precise, such as with recognition-based inputs (voice and gesture) and touch. In these cases, recognition algorithms and sensing techniques often output a series of guesses indicating possible inputs the user may have provided. However, modern input systems are not built to easily handle multiple *possible inputs*, and as a result many systems use heuristics to convert these multiple possibilities into a single, precise value. Unfortunately, this greatly reduces the information the user interface toolkit has about a user's input, making the system error-prone. Maintaining information about alternate input interpretations and likelihoods throughout the input handling process could allow user interface toolkits to incorporate more information when making decisions.

1.1 Motivating Examples

Consider the example of touch input where a user touches not a single point, but an entire area. Research by Moscovich shows that using the entire touch area for input can lead to interaction improvements (Moscovich 2009), and it is likely that further extending this representation to a distribution of likelihoods across *possible touch locations* could be even more beneficial, as is hinted by Weir in (Weir et al. 2012). Although the capacitive sensor arrays on most touch input devices are capable of detecting the entire area that a finger touches, this area is often boiled down to a single point (usually the centroid of the touch). This information reduction can cause problems in certain situations. Consider the scenario in Figure 1.1 where a user intends to press one of two buttons, however the center of his touch lies directly between the two. While a close inspection of the picture would indicate that the red button is more likely to be the intended target, a conventional input system would actually do nothing because the centroid of the touch lies directly between both buttons and hits neither target. In contrast, a user interface toolkit that looked at the entire area of the touch and considered the likelihood of each button press relative to the other would make a better decision.

As another example, consider the following scenario: a person is trying to tell a voice-controlled system to share a picture she just took with a friend. This type of scenario is increasingly common, especially with wearable interfaces such as Google Glass (Google 2013). The user says “send picture to John”, and the voice recognition returns with three possible commands:

1. Send picture to “Don”, with probability 0.5
2. Send picture to “John”, with probability 0.3
3. Send picture to “Ben” with probability 0.2

All three contacts, “Don”, “John” and “Ben” are in the person’s contact list, however she sends emails to John far more frequently than to Don or Ben. A naïve algorithm would take the most likely recognized input that matches a contact “Don”, and send an email, failing to take into account the frequency with which Amy sends emails to John. A more sophisticated approach would be to take into consideration the frequency of emails sent when deciding who to send an email to. In fact, it is likely that many well-built input systems do exactly this. Unfortunately, this currently requires modification of either the voice recognition engine or the user interface logic, both of which are undesirable. A better approach would be to build in this sort of consideration of what a user might do (which we refer to as *prior likelihoods*) directly into the user input handling system, removing the need for developers to write custom code.

Our final example comes from touch interaction. The specific scenario here is inspired from an interaction in the application “Paper” from 53Designs (53Designs 2012). Consider a painting application that supports two separate two-fingered gestures: pinch to zoom, and rotating two fingers either clockwise or counterclockwise to undo/redo editing operations (Figure 1.2). Both actions (actions are operations that update interface state) cannot be executed at once. Therefore, when a user touches down with two fingers, the

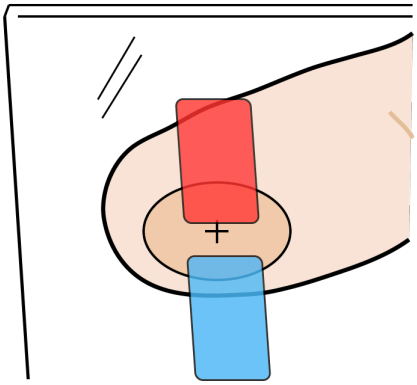


Figure 1.1: A user touches between the red and blue button, the cross indicates the centroid of the touch, which is used to determine the finger’s target by conventional input systems. He intends to press the red button, and more of his finger overlaps the red button than the blue. However, the centroid of his touch is directly between both buttons.

state of the system is uncertain: it is unclear whether the user intends to zoom or undo an action. When a user moves slightly, a naïve program may immediately make a decision and change its state to zooming or undo, based on whether the user's fingers moved slightly apart or slightly in a clockwise direction. A more intelligent algorithm would wait longer until there is sufficient evidence to confidently say that a user is intending to either zoom or undo. Although some programs may implement logic to delay a decision until the program is confident about a user's decision, this requires significant custom logic to be implemented in the user interface, and as a result many interfaces do not do this, favoring the immediate (but error-prone) decision instead.

Examples of uncertainty in user interfaces go beyond just voice and touch input, however. A vision recognition system that recognizes how many fingers are lifted when a user opens his or her hand may return the following results: five fingers with confidence 0.5, four fingers with confidence 0.4, and three fingers with confidence 0.1. Consider now what happens in an interface that has only four options, and asks a user to raise some number of fingers to select an option. A naïve system would automatically take the input with the highest confidence (5 fingers), and fail, as there are only four options available.

There is evidence beyond these simple examples to suggest that maintaining information about input alternatives could serve to help user interface toolkits make more informed decisions about what to do given some input from the user (Mankoff 2001). Not only does tracking input alternatives (and their likelihoods) enable user interface toolkits to use more information when making decision, it also allows user interface toolkits to recognize when input is truly ambiguous and delay action.

However, while a user interface toolkit that handles uncertain inputs may have several benefits, it might also have the drawback of requiring application developers to think about likelihoods when they are developing applications. For example, when a component of the user interface (which I will refer to as an *interactor*) such as a button receives an input event that has a probability of 0.5, the interactor must keep track of this probability when making future decisions. This adds a great deal of complication to the logic of

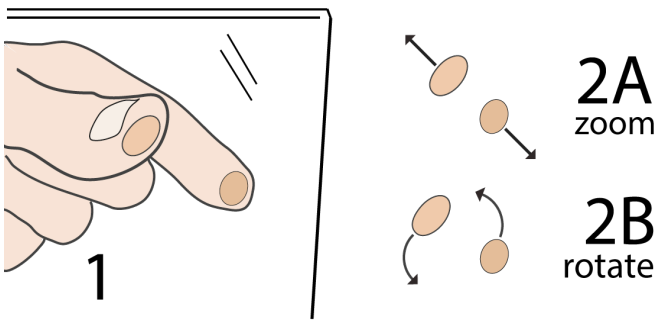


Figure 1.2: Two finger gesture example. 1. User presses down with two fingers. It is now ambiguous whether they want to execute a zoom (2A) or rotation gesture (2B).

interactors, making these interactors more difficult to develop. This dissertation presents a user interface toolkit that maintains information about input uncertainty throughout the input dispatch process (determining which interactors should handle input) without requiring developers to think probabilistically.

1.2 Contribution

New interaction modalities such as touch input, voice recognition, and free space gesture hold the promise of enabling more natural interaction. These recognition-based inputs also come with a fair amount of uncertainty, which current input toolkit are not built to handle. This dissertation presents a new user interface architecture that treats user input as an uncertain process, approximates the probability distribution over possible interfaces using Monte Carlo sampling, and provides tools for interface developers to easily build probabilistic user interfaces. Importantly, alternate interfaces (and their likelihoods) are managed by the architecture itself, allowing for interface developers to reap the benefits of probabilistic interfaces without needing to think probabilistically. This new user interface architecture is embodied in the JULIA toolkit: a JavaScript User interface Library for handling Input Alternatives. The specific contributions of this dissertation are as follows:

- A new architecture for modeling and dispatching uncertain user input, using Monte Carlo sampling to approximate the probability distribution over possible interface states.

- A system for generating fluid interactive feedback which continuously communicates alternative (or future) input interpretations as a user is interacting and allows her to disambiguate intent.

- An API and library which allows user interface developers to easily build probabilistic interfaces. This includes a mechanism for specifying interactor behavior using probabilistic state machines, a rich set of classes for developing interactive feedback, and a built-in mechanism for adjusting likelihood of future actions based on past input.

- A large collection of interaction techniques and applications which demonstrate the versatility and power of the toolkit. These include demonstrations of how to leverage prediction to accelerate interaction, how to build interfaces that allow users to easily switch between multiple input interpretations, and how to perform Bayesian inference to adapt interface behavior.

1.3 Organization

The thesis is organized as follows: first, I cover prior work. I also give a brief primer on Monte Carlo modeling of probabilistic phenomena, which is at the core of the system. Next, I provide an architectural overview of the JULIA toolkit. I then explain in detail each component of the toolkit, and provide a collection of applications and interaction techniques developed using the toolkit.

1.4 Audience

This dissertation is intended primarily for developers of user interface toolkits and user interfaces. It assumes a basic knowledge of what makes up a user interface toolkit, as well as basic knowledge of computer science terminology. The toolkit this dissertation describes serves three primary sets of stakeholders: *end users*, *application developers*, and *toolkit developers*. *End users* are the people that use applications developed using the JULIA toolkit. *Application developers* make build the applications that end users consume. Application developers build their applications out of pre-built user interface components such as buttons and scrollbars. We call these user interface components *interactors.* Application developers may also want to occasionally write custom interactors to fit their needs. Finally, *toolkit developers* build components of the toolkit. They develop logic to control the dispatch of user input, the interactors provided in the toolkit (e.g. buttons, sliders), and any other tools such as graphical user interfaces (GUIs) for designing the layout of applications. The primary requirement of the JULIA toolkit is to make it as simple as possible for toolkit developers and application developers to create understandable, beautiful, and easy to use interfaces for end users.

1.5 Terminology

Throughout this thesis, I use the term *likelihood* instead of *probability* when describing input alternatives. This choice of terminology is intentional; the values associated with a particular input alternative are an approximation of the probability that an input alternative is the 'correct' or 'intended' input. Because modeling every possible alternative, and its probability, is difficult (and potentially infeasible), we use the term likelihood to emphasize that this is an estimate of probability. These likelihoods aren't necessarily guaranteed to represent the true probability, rather they are an *estimate* of the probability, and we make our best effort to ensure that these *estimates* represent the true probability distribution of the system.

When describing an input event property that has several alternative values, each of which has a particular *likelihood,* I use the structure provided by *probability density functions* (PDFs) and *probability mass functions* (PMFs). A PDF is a function which describes

the likelihood that some continuous random variable has some observable value. Similarly, a PMF is a function which describes the relative likelihood that some discrete random variable is equal to some observable value. For example, a Boolean variable (which can have one of two values: True or False) can be represented as a PMF with probability of 0.2 of being true and 0.8 of being false. As another, more complicated, example, say we want to have a way to represent the location a user *intended to hit* given a touch centroid, and the major and minor axis of the touch. Then, for each pixel coordinate on a screen, there is some likelihood that the user intended to hit this particular pixel when they touched the screen. Many of these pixels will have a likelihood of 0, but some (near the center of the touch) will be higher. Because pixels are discrete values, a PMF describes the likelihood of each pixel being the users intended point of interaction. Most of the input events and other properties that the JULIA toolkit interacts with are represented with PMFs, with the exception of real-valued numbers.

Finally, I use the term *definite* to refer to a variable, property or entity that does not have any uncertainty associated with it. Conventional input events are *definite,* while probabilistic input events contain multiple *possible alternatives, labeled with likelihood.* I will also sometimes refer to *samples*. These *samples* are objects that do not contain any uncertain properties, but that are created by randomly sampling properties from objects that do have uncertain properties. These samples have a *sample weight* attached to them to indicate this sample's likelihood in the space of possible input states.

2 BACKGROUND

In a world of logic gates that almost never fail and machines that always follow a strict set of predetermined rules, uncertainty is a black sheep. Uncertain logic is not easy to think about or troubleshoot; yet despite this, uncertainty has been widely adopted within the broader field of computer science. Probabilistic programming is a burgeoning field in computer science, and researchers have developed entire programming languages designed to simplify probabilistic reasoning, e.g. Church (Goodman et al. 2012).

In the domain of user interfaces, uncertainty arises during three different phases of the input handling process: at the sensor level, during input interpretation, and during application action. At the sensor level, uncertainty arises when sensor values do not fully reflect actual input, and when noise threatens to drown out actual signal. During input interpretation, errors often arise because of misrecognition of input, misunderstanding of whether the user is trying to perform input, and errors regarding which target the user intends to interact with. Finally, during application action, uncertainty is often caused by systems that try to perform prediction, or when noisy inputs do not accurately communicate user intent.

There is a wide body of research that aims to improve interaction by handling uncertainty at each of these levels individually. One of the primary ideas behind the JULIA toolkit is that if we handled uncertainty in a comprehensive and general way *across* these levels, we could create better interfaces. In this chapter, I will first show how existing research improves interaction by handling uncertainty—often in a problem or modality-specific way—at each of the three levels mentioned above (2.1 “Interaction Techniques”). During this exposition, I will argue that the interaction techniques presented could be further improved by integrating information across different levels of the input dispatch process in a general way.

After discussing interaction techniques, I will describe existing systems that have been built with uncertainty in mind, and point to the gap that this dissertation work fills (2.2 “Systems and Toolkits”). There has been some work that tries to handle inputs with uncertainty, but little work models likelihood directly at the toolkit level. This dissertation models likelihood at the toolkit level, with the aim of abstracting away as much probabilistic reasoning as possible away from the developer. Table 2.1 summarizes the literature reviewed in this chapter. The JULIA toolkit provides a solution that carries uncertainty across all three levels of the input handling process, making it possible for improvements at each level of the input handling process to work in concert and provide a better user experience.

		Interaction techniques		Toolkits, systems, and frameworks	
		Directly tracks input Alternatives and/or likelihoods	**Circumventing uncertainty without directly modeling it**	**Directly computes likelihoods**	**Tracks alternatives without directly computing likelihoods**
Where uncertainty is handled	**Sensor**	"FingerCloud" (Rogers *et al*. 2010) "AnglePose" (Stewart & Murray-smith 2011) "User Specific Machine Learning Model" (Weir *et al*. 2012)	"Understanding Touch" (Holz & Baudisch 2011) "100,000,000 taps" (Henze *et al*. 2011) "Contact Area interaction" (Moscovich 2009) "Shift" (Vogel & Baudisch 2007)	"Continuous Uncertain Interaction" (Williamson 2006)	"An Architecture and Interaction Techniques for Handling Ambiguity in Recognition-Based input" (Mankoff 2001)
	Interpretation of input	"Dynamics and probabilistic text entry" (Williamson 2003) "Octopocus" (Bau & Mackay 2008) "SenseShapes" (Olwal & Feiner 2003) Learning to Predict Engagement with a Spoken Dialog System in Open-World Settings (Bohus & Horvitz 2009) "Mutual disambiguation of recognition errors" (Oviatt 1999a) "Audio feedback for gesture recognition" (Williamson & Murray-Smith 2002) "Multimodal Integration—A Statistical View" (Oviatt & Cohen 1999)	"Method, device, and graphical user interface providing word recommendations for text input" (Bellegarda *et al*. 2012) "Beating Fitts' Law" (Balakrishnan 2004) "Put that there" (Bolt 1980) "Fluid Sketches" (Arvo & Novins 2000)	"Continuous Uncertain Interaction" (Williamson 2006) QuickSet (Cohen *et al*. 1997) TYCOON (Martin *et al*. 1998)	"An Architecture and Interaction Techniques for Handling Ambiguity in Recognition-Based input" (Mankoff 2001) "A user interface framework for multimodal VR interactions" (Latoschik 2005)
	Application action	"Adaptive Interface Based on Personalized Learning" (Liu *et al*. 2003) "Lumiere" (Horvitz *et al*. 1998)	"Pegasus"(Igarashi *et al*. 1998)	"Mutual disambiguation of 3D multimodal interaction in augmented and virtual reality" (Oviatt 1999a) "XWand" (Wilson & Shafer 2003)	"An Architecture and Interaction Techniques for Handling Ambiguity in Recognition-Based input" (Mankoff 2001) "Principles of Mixed-Initiative User Interfaces" (Horvitz 1999)

Table 2.1 Overview of interaction techniques and toolkits that deal with uncertain input, organized first by where uncertainty is handled (sensor, input interpretation, application action), then by the type of contribution (interaction techniques, toolkits/systems), and finally by how uncertainty is handled (directly computes likelihoods or not).

2.1 Interaction Techniques

The effects of misinterpretation of error-prone inputs often lead to significant user frustrations (Frankish *et al*. 1995). These problems are so common that many of them have specific names, for example the "Fat Finger Problem" and "Midas Touch". Table 2.2 presents a list of common problems caused by misinterpretation of uncertain inputs, which I will explain in subsequent sections. Much research has gone into developing interaction techniques to solve these specific problems. Some interaction techniques do this by explicitly modeling uncertainty (Weir 2012; Williamson 2003); others use clever tricks or heuristics (Henze *et al*. 2011; Vogel & Baudisch 2007; Balakrishnan 2004). As mentioned in the introduction of this chapter, I organize these problems according to where uncertainty arises in the input handling process: at the sensor level, during input interpretation, and during application action (Table 2.2).

Many interaction techniques focus on solving these problems by reducing uncertainty immediately when it arises, however in many cases interactions could be further improved by considering uncertainty across several stages. For example, information provided later on could help support initially less likely interpretations, or refute more likely interpretations. In the voice input example mentioned in Chapter 1, the information about Amy's frequently contacted friends could be useful when detecting which name she spoke, and vice versa. This section explicates the specific problems mentioned in Table 2.2, and reviews interaction techniques that solve these problems, with the aim of illustrating how incorporating information across different levels would serve to improve user experience.

Uncertainty from sensor noise	**Uncertainty during input interpretation**	**Uncertainty during application action**
"Fat Finger" problem Sensor Noise	"Midas Touch" Palm Rejection Targeting Misrecognition of Gestures and Voice	The Plight of Clippy Error Recovery

Table 2.2 Common problems caused by uncertainty, and where these problems arise.

2.1.1 Sensor Level

Sensor input lies at the foundation of interaction—errors at this level propagate to an alarming degree and are difficult to recover from. At the sensor level, uncertainty arises when sensor values do not reflect actual input, and when noise obscures or distorts the signal. I refer to these two problems as *sensor imprecision* and *sensor noise*, respectively. Many modern input handling systems do minimal work to handle this uncertainty; a classical example is touch input, where a touch area is boiled down to a single point.

In this section I will show several ways that researchers have found to mitigate errors due to uncertainty in sensor input. Unfortunately, without an ability to disseminate this information further in the input dispatch process, the utility of these techniques is limited.

2.1.1.1 Uncertainty Due to Sensor Imprecision

The most well-known problem caused by uncertainty due to sensor imprecision is the "Fat Finger Problem". This refers to the common problem that our fingers are unable to target small items on a touchscreen partially due to target occlusion, but also due to the fact that most touchscreens convert the entire finger's touch area into a single point when sending it to the system. Users have a very difficult time knowing exactly where their touch input will land, thus making it difficult to hit small targets.

There is a large body of work that aims to solve this particular problem, with two general approaches. One body of work *modifies touch location* to generate better guesses of the user's likely intent. In a probabilistic setting, many of these algorithms could be easily modified to output not a single guess, but a set of guesses and likelihoods. Second, several techniques aim to adjust interaction when input is uncertain. These techniques could greatly benefit from a fully probabilistic approach.

As an example of the first approach, some work aims to determine an offset function that maps the sensed touch location to an 'intended touch location' given extra information about the touch (such as pitch, roll and yaw). Holz *et al.* introduce adjustments to a touch point offset based off of information such as finger pitch and yaw, which they obtain through a fingerprint scanner(Holz & Baudisch 2010; Holz & Baudisch 2011) (Figure 2.1). Henze *et al.* take advantage of big data to determine an offset function by analyzing millions of taps obtained from a mobile phone game in "100,000,000 taps" (Henze *et al*. 2011). In both cases, the works presented output a single guess for an intended touch location. However, since the algorithms are fundamentally trying to predict where a user's intended touch would be, it would be natural for these approaches to instead output a set of possible values, which could then be propagated further into the input handling process, where additional information for resolving ambiguity may be available.

Another approach is to directly model touch input as an uncertain process and infer intended touch location given sensed location. Weir *et al.* show how to improve touch accuracy by modeling touch as an uncertain process and learning offset functions based on previous actions (Weir *et al*. 2012). Related to this, AnglePose performs probabilistic inference on the pose of the finger using a particle filter, and then uses this information to improve pointing accuracy (Stewart & Murray-smith 2011). Both of these techniques already treat input as uncertain. While currently these systems output a single guess, it would be natural for these approaches to output a set of guesses (and likelihoods) in a probabilistic setting.

Figure 2.1: Examples of existing interaction techniques for reducing touch uncertainty. Left: Holz determines touch offset using information about finger pitch, roll, and yaw. Right: Moscovich treats entire touch area as interaction region.

Moving away from offset functions and into interaction techniques, one approach to solve the fat finger problem at the sensor level is to change or augment touch interaction to help users target small objects. Contact area interaction (Moscovich 2009) builds interactors that use the entire touch area for interaction. This work treats the entire touch area as equally likely (Figure 2.1). This interaction is an example of something that could benefit from using a distribution over possible values.

Another interaction technique that would greatly benefit from having a distribution over possible touch values is the Shift interaction technique. When a user's touch is ambiguous between two small touches, Vogel *et al.* create an enlarged callout showing a copy of the area occluded by a touch, allowing the user to more precisely specify their intended touch location (Vogel & Baudisch 2007). One of the nice features of the work is that this callout only pops up when input is ambiguous in between several buttons. This technique determines input as uncertain *via* a simple distance based heuristic, and would greatly benefit for more holistically derived likelihoods.

Moving away from touch interaction, uncertainty due to sensor imprecision arises in other input techniques where sensing is systematically imprecise, or where precision decreases in certain situations. For example, gaze tracking using low-cost commodity webcams (Sewell & Komogortsev 2010) suffers from imprecision due to user head movement and difficulty in identifying irises under varying light conditions. Information that causes these systematic errors may be obtained separately, and used to adjust confidence in eye tracking. Location sensing using a magnetometer, as demonstrated in (Harrison & Hudson 2009), changes in precision based on the magnet's distance to the sensor due to the quadratic drop-off of the magnetic field strength. Adjusting system parameters based on sensor precision in this domain is promising, but unexplored.

2.1.1.2 Uncertainty Due to Sensor Noise

A second major issue that occurs at the sensor level is sensor noise. Many modern sensors such as accelerometers, microphones, and gyroscopes hide their true signal amidst a haze of background data. To handle this noise, two approaches are common. First, many systems aim to reduce noise using probabilistic methods. These methods lend themselves

well to outputting a series of guesses, rather than one concrete value. A second approach is to adjust the dynamics of a system based on the certainty of sensor values, something that would greatly benefit from a probabilistic approach.

Two approaches are common for reducing sensor noise. First, *filtering* uses probabilistic methods on a continuous stream of data. A common such filter is called a Kalman Filter (Welch & Bishop 2006, Kalman 1960), which first tries to predict the next value of the signal, and then filter out or modify values which diverge widely from the prediction, hence are unlikely to be correct. Second, *sensor* fusion combines signals from multiple sensors to produce a more accurate, unified signal. This approach combines sensor readings from multiple sources, weighing them by the reliability of the source. As a result, the combined signal tends to be more reliable than each signal individually. A common application of sensor fusion lays in GPS/INS, the use of Global Positioning System (GPS) signals to correct for errors in Inertial Navigation Systems (INS). INS measurements are often accurate only for a short period of time, but often drift over time. GPS signals are used to correct for this drift using a Kalman Filter. The idea central to sensor fusion can be used in user interfaces not just to improve accuracy of individual sensors, but also to improve overall interaction. Now, the signals are guesses about user input during different phases of input handling (sensor level, input interpretation action). As with sensor fusion, combining information at these different levels reduces overall error in an input handling system.

A second approach to handling sensor noise is to adjust the dynamics of a system based on the noise level of incoming sensors. The idea behind this is drawn from horseback riding—a horse senses the certainty with which its handler controls it. When the handler's actions are confident, the horse obeys, however, when the handler is tentative (for example, if the rider is injured somehow), the horse makes its own decisions. Using this same idea, FingerCloud (Rogers *et al*. 2010) tracks the (uncertain) 3D position of a finger above a mobile device to control mobile map navigation. In FingerCloud the uncertainty of finger position increases with the height from the display, and the system adapts its controls accordingly, allowing for application-controlled behavior when the finger is far from the display, and more fine-grained control when the finger is closer (and thus its position more certain). FingerCloud shows an example of something that is currently not possible in input systems, because applications have no way of knowing how uncertain the input they are handling is. Exposing information about uncertainty across different levels would enable FingerCloud-like interactions for all types of interaction.

The research above shows some ways that researchers have found to mitigate these errors, unfortunately these interaction techniques have no way to propagate information about uncertainty to other interactors in the dispatch system. For example, FingerCloud adapts its positioning controls based on uncertainty of measurement, but cannot actually adjust how buttons or other controls respond to the position given. Without an ability to

disseminate the information these new interaction techniques generate further in the input dispatch process, the utility of these techniques is limited.

2.1.2 Input Interpretation

During input interpretation, errors often arise due to misunderstanding of whether the user is trying to perform input (*when* the user is interacting), errors regarding which target the user intends to interact with (*what* the user intends to interact with), and misrecognition of input (*recognition error*). Much research has gone into solving each of these individual problems. In this section, I briefly touch on heuristics-based approaches towards handling many of these problems, however the solution space of this related work is vast, and beyond the scope of this thesis. There is, however, a sprinkling of solutions that handle this uncertainty by directly modeling input interpretation as uncertain. In this section, I will thoroughly investigate solutions that directly model uncertainty. Further, I illustrate how some of the interaction techniques which use heuristics to handle uncertainty could instead be modeled probabilistically to achieve better results.

2.1.2.1 *When* is the User Interacting?

Many new input techniques such as gaze-based input, free-space gesture, and voice hold the promise of providing natural interaction. However, because these natural interactions mimic what humans already do in real life, it is difficult for systems to determine when a user intends to direct their actions towards a system versus not. This problem manifests itself in many forms, depending on the input modality.

In gaze-based interfaces, the "Midas Touch Problem" refers to the problem that users have no easy way to distinguish looking towards an object from acting on it. One approach is to use gesture to switch input modes between targeting and action (Istance *et al*. 2008). Another approach is to perform selection by explicitly not looking at a target (referred to as anti-saccades) to perform selection (Huckauf *et al*. 2005). In many situations, just looking at gaze simply is not enough, systems need to be able to integrate information from other sensors (or other parts of input process) when making decisions about user action.

When performing in-air gestures or speaking, it is often unclear whether a user's actions are intended for the system. This challenge of determining intention to interact has been addressed by Bohus *et al.* (Bohus & Horvitz 2009), where Bohus looks at head position and orientation to determine whether a user intends to address a virtual agent. One great feature of this system is that it produces a likelihood for intention to interact, meaning that this information can be easily carried into the decision-making process. Furthermore, it could easily combine information from earlier on in the dispatch process when making decisions.

In the domain of touch and pen input, one major problem that arises is in ignoring palm input when a user is writing with a pen on a tablet. This problem is more generally referred to as palm rejection. One solution is to use non-capacitive sensor layer which does not respond to human touch. Unfortunately, adding a second sensing layer for touch input is expensive, and does not support a full range of interactions. Specifically, when pen interactions are combined with touch, e.g. as in Hinckley's "Pen + Touch" (Hinckley *et al*. 2010), a system must determine which touches are intentional vs. not. One approach is simply to disable all touch inputs when a pen is present, but again this does not align well with the natural interactions provided by Pen + Touch. Many projects reject palms based on touch size (Zeleznik *et al*. 2010; Murugappan *et al*. 2012; Hinckley *et al*. 2010). Another technique is to 'hold' a surface in place and prevent its manipulation by touch (Zeleznik *et al*. 2010). A third approach is to define 'rejection regions' where all touch input is ignored. Several patents exist to address the issue (Yeh & Chen 2011; Hinckley *et al*. 2012). One interesting approach presented in UnMousePad (Rosenberg & Perlin 2009) is to classify touches as either point touches (a pen) or as area touches (palm). The approach presented by UnMousePad could be especially beneficial when combined with other information about application context, etc. to make palm detection more accurate.

2.1.2.2 *What* is the User Interacting With?

In addition to the challenge of determining *whether* a user intends to perform input, the challenge of *what* the user is intending to target is an additional problem that arises with many new inputs. Even taking into account sensor inaccuracies, a user may often simply miss a button or other interactive element such that even the most accurate sensor would not be able to detect the intended target.

There are many approaches to solving these problems that use heuristics and clever interaction techniques. A few also use probabilistic inference. Starting off with the most widely used example, many mobile operating systems combine touch input with character-level word inference to assist users when typing (MacKenzie, 2002). For example, iOS enlarges the hit area of keyboard keys according to which keys have been pressed already (Bellegarda *et al*. 2012). In a similar vein, Williamson incorporates a language model in his gestural text entry system to improve text entry accuracy in in "Dynamics and Probabilistic Text Entry" (Williamson 2003). In this application, Williamson uses a language model to allow for text entry using continuous movements. The application changes its dynamics to reduce effort in entering likely text. Researchers have long worked on techniques for artificially facilitating pointing at small and distant targets in user interfaces. Balakrishnan provides a good overview of the vast amount of research in this area (Balakrishnan 2004). In his overview, Balakrishnan found that while many of the techniques provided work well for distant, sparse targets, they do not work well in the common situation where multiple targets are located in close proximity. This very problem is an area where the probabilistic toolkit could provide great value, as it could easily enable a combination of information from many sources to make intelligent decisions.

2.1.2.3 Recognition Error

Another major challenge in input interpretation lies in recognition-based input. Recognition-based inputs are inputs resulting from complex, inference-based algorithms, many of which output a list of guesses along with confidences. Due to the complex nature of these inputs, and because they are frequently improperly handled, these inputs are a large source of errors in user interfaces. Two of the most common recognition based inputs are speech recognition inputs and gesture inputs.

Speech recognition is a notoriously difficult problem that has been challenging researchers for decades. Researchers have come up with many ideas to improve speech recognition accuracy. The classic choice in speech recognition is the Hidden Markov Model. In (Rabiner 1989), Rabiner explains how Hidden Markov models are combined with information about the grammar of a language, the user's current task, and word sequence models to help disambiguate multiple likely recognized words. There is a large body of work in the speech recognition community related to leveraging other inference tools, as well as other information about grammar and voice to improve recognition accuracy, however this vast space of literature is beyond the scope of this review.

A second approach to improving speech recognition in the field of human computer interaction is to leverage information from other input sources to help disambiguate recognition results. An inspiration for this work comes from the "Put That There" system developed in 1980, when Bolt combined speech input with voice to provide fluid, natural interaction (Bolt 1980). The research in this area is so large that it has a name for itself—multimodal interaction (though, as pointed out in (Oviatt 1999b), it is important to note that multimodal input is not confined to just improving speech and pointing systems). The challenge of multimodal input is to combine multiple sources of uncertain input to provide mutual disambiguation. In "Mutual disambiguation of recognition errors in a multimodal architecture" (Oviatt 1999a), Oviatt shows how combining inputs from multiple sources can lead to improvement in voice recognition, particularly for people with hard-to-recognize accents. Wu and Oviatt further present statistical methods for reducing recognition error by fusing probabilities from multiple sources in (Oviatt & Cohen 1999). Additionally, in (Eisenstein & Davis 2005), Eisenstein shows how hand gestures can be used to segment spoken sentences during dialogue. Eisenstein also shows how gesture can be used to improve coreference resolution in natural language processing in (Eisenstein & Davis 2006). Note that while multimodal input may also serve to provide natural interactions, (Oviatt 1996) provides a good example of this, the main focus of the work as covered in this review is in how fusing multiple input sources can serve to reduce ambiguity. Multimodal input is, of course, not just limited to improving voice recognition. In (Eisenstein & Davis 2004), Eisenstein *et al.* use voice to improve recognition of visual gestures. Additionally, Kettebekov *et al.* illustrate how prosodic synchronization (segmenting gestures based on the rhythm of speech) can be used to improve gesture segmentation in continuous gesture scenarios (Kettebekov *et al*. 2005).

The work I just described illustrates the benefits of fusing multiple input sources in improving voice recognition accuracy. Now, just imagine how much more of an improvement in interaction we could get if we could not only combine information about inputs, but also take into account a user's preferences, their previous actions, and other contending inputs when making decisions about interaction.

Speech recognition isn't the only type of input that presents challenges for researchers, however. In fact, gesture recognition is a much more commonly encountered challenge. Detection of pre-defined gestures is increasingly common on mobile phones and other touch inputs, as users try to do more and more things on their mobile devices. While gesture languages with only a few inputs generally perform accurately, recognition rates drop significantly as more gestures are added to the language. Fortunately, many gesture recognition programs provide gesture alternatives, and likelihoods for these alternatives. Two such systems are Gesture Studio (Lü 2013), and the $1 Recognizer (Wobbrock *et al*. 2007). Several interaction techniques to mitigate these problems present themselves. Octopocus (Bau & Mackay 2008) provides a dynamic guide for gesture-based interfaces which adjusts its feedback according to gesture probabilities. Also, in "Fluid Sketches" (Arvo & Novins 2000), Arvo *et al.* provide continuous feedback about recognized gestures to users. This dynamic feedback is crucial and unfortunately largely absent from many user interfaces. In addition to visual feedback, some researchers have also worked to provide auditory feedback according to probabilities of various gestures (Williamson & Murray-Smith 2002). This works shows an additional benefit of systematically tracking probabilities: the opportunity for rich and expressive feedback about gesture recognition results.

2.1.3 Application Action

Once input is interpreted by a user interface toolkit (e.g. the system decided a specific button was pressed, or that the user said some word), an application must decide what specific actions to take (i.e. close a window, or initiate a Google search for the word "spork"), if any. The basic problem of any input system during the application phase is to determine what the user intends to do, given information about sensed input, and the interpretation of this input. During this phase, uncertainty is often caused by systems that try to guess what a user is trying to do, or when inaccurate input interpretations do not communicate user intent. Many of the interaction techniques regarding misinterpretation of noisy inputs have been covered in the above sections, therefore this section will focus on uncertainty that arises when systems try to predict a user's intent. Knowledge about alternative inputs is especially valuable during the action phase, which can integrate knowledge about the higher level application state, a user's previous actions, and information about alternate inputs to make better decisions.

2.1.3.1 Uncertainty Due to Prediction of User Intent

The former task of performing inference to automate tasks or provide suggestions to users is a wide area of research called Adaptive and Intelligent User Interfaces. An early and insightful example into the problem comes from the domain of user agents: programs that act as 'personal assistants' to the user. The most well-known user agent is Microsoft Clippy, whose unfortunately constant interruptions at inopportune moments led the agent to be widely disliked. The failure of Microsoft Clippy points to a common challenge when dealing with uncertainty: it is difficult to know a user's intended action given limited data such as where their mouse is and what they're typing. At a high level, this is the same problem that the probabilistic user interface toolkit aims to support: helping applications better make decisions about actions given noisy, uncertain input. The tradeoffs in this decision process are considered at length in Horvitz's "Principles of Mixed-Initiative User Interfaces" (Horvitz 1999). Mixed initiative user interfaces, as Horvitz refers to them, consider not only the user's uncertain internal state, but also the cost of interrupting the user, as well as the cost of continuing without interruption, when deciding on an action. Providing systematic tracking of uncertainty would allow systems to connect logic providing high level inference of user intention to lower level input dispatch, where uncertainty may also arise. In other words, maintaining information about alternatives and likelihoods at the input event level may serve to help programs such as user agents make better decisions.

The inference engine prototype that the Clippy office assistant was based off of came from the Lumiere project, which looked at user's actions and aimed to predict what they would do next (Horvitz *et al*. 1998). Five years later, Liu *et al*. aimed to improve upon this core idea by similarly looking at a user's interaction history and come up with formatting shortcuts in Microsoft Word based on a user's actions to help speed up the document editing process (Liu *et al*. 2003). Both these approaches are highly personalized, as they look at an individual's interaction history to improve their predictions. But when making decisions about actions, it is good to not only consider the user, but also the noise of the tools they are using, as well as the accuracy of recognition algorithms used to interpret this input.

Like the LookOut project in Horvitz's "Mixed Initiative User Interfaces" project, many other applications delay action under uncertainty (though many real world applications do not consider the cost of this delay). For example, some direct manipulation touch interfaces such as Paper by 53Designs(53Designs 2012) use two fingers to trigger multiple gestures. In the case of Paper, a two finger pinch zooms out, two finger expansion zooms in, and two fingers rotating initiates an undo mechanism. When a user touches her fingers down and moves initially, the application cannot be confident in what the user's intended gesture is. The application does not change modes until it is confident in the user's decision. This is a common problem in many modern interfaces, and each new program must implement its own custom logic to delay this action. The intention for the JULIA toolkit is

to facilitate this action delay naturally, without requiring the developer to even think about delaying of action. In the JULIA toolkit, this delaying of action until a system is sufficiently certain will just work.

This section has given an overview of the types of ambiguity present in inputs, current approaches to solve them, and illustrated how maintaining information about uncertainty throughout the input handling process could facilitate the fusion of all of these innovations. In the next section I cover the topics of feedback and feedforward: communicating uncertain system state to users.

2.2 Visualizing System State

Feedback is a crucial component to user interfaces: the visibility of system status is listed by Jacob Nielsen as one of the 10 criteria for heuristic evaluation of interface usability (Nielsen 1993). In today's world of complex, gesture-based interfaces where the interactions to trigger commands are unclear, gesture discoverability is equally important: users must understand which gestures are available (affordance), what actions their interactions may cause, what actions their interactions could lead to (feedforward). Naturally, a great deal of thought has already been put into methods for providing feedback and feedforward in the research literature, as well as in real-world applications.

In this section I will explore existing work relating to showing feedback about uncertainty in user interfaces, both to motivate the importance of such a component in the JULIA toolkit, and also to show how such uncertainty might be handled in my toolkit.

2.2.1 Feedback and Feedforward

Feedback in the context of Human Computer Interaction refers to any information returned from the computer to the user in response to action (Wensveen *et al*. 2004). Figure 2.2 illustrates a diagram showing the role of feedback in human computer interaction. Many systems give feedback about changes of state within a system (i.e. a button going from start state to down state). In the real world many environments provide *a priori* information about how an artifact can be used. This is defined by Norman as affordances (Norman, 1988), and by (Wensveen *et al*. 2004) as 'inherent' feedback. One challenge for providing affordances in, for example, gesture-based interfaces on surfaces is that it is difficult to provide affordances for a wide range of gestures beyond swipes and taps. Such challenges also exist in other domains such as in-air gesture. To compensate for this some interfaces provide suggestions for how to continue gestures when they are started. This is called feedforward. Examples of feedforward techniques are the GestureBar (Bragdon *et al*. 2009) and Interface Ghosts (Vanacken *et al*. 2008). Bau provides an excellent overview of feedback and feedforward mechanisms in his dissertation work (Bau 2010). He contrasts two approaches for feedforward and feedback: a conventional approach (Principle

1, Figure 2.2) where feedforward is given, input executed, and feedback about that input is given; and the 'interaction streams' approach, where feedforward and feedback are given in a constant feedback loop. Examples of interaction streams are in Octopocus (Bau & Mackay 2008) and Fluid Sketches (Arvo & Novins 2000). Interaction streams not only support the learnability of interfaces but also allow for a feedback loop for users to refine their gestures.

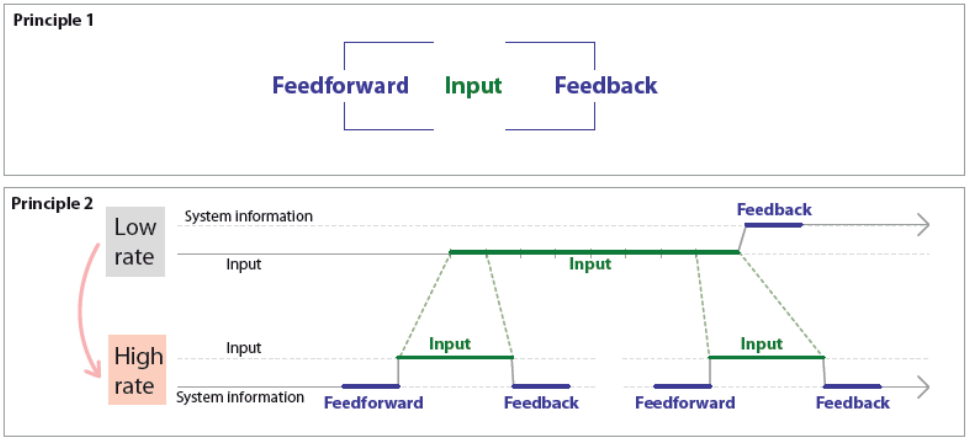


Figure 2.2: The role of feedback and feed forward in interfaces, as presented by Bau *et al.*(Bau 2010).

2.2.2 Existing Feedback Mechanisms for Uncertain Input

When input interpretation is ambiguous, users must understand not only how to provide this disambiguation, but also whether they have succeeded in their disambiguation during their continuous interaction. Techniques such as dialogue boxes are not sufficient. The disambiguation interface must be a continuous interaction stream: a rapid-fire conversation between human and computer, ensuring that the computer fully understands correctly the user's intention.

2.2.2.1 Input Interpretation

Examples of such fluid feedback mechanisms exist. In "Predictive Uncertain Displays" (Williamson 2006), Williamson uses inference to continuously update a visual display of a user's interpreted action. In fact, feedback is central to Williamson's thesis that user interaction can be thought of as a continuous control process (Williamson 2006). Williamson further explores this same idea in the domain of audio feedback (Williamson & Murray-Smith 2002), where the authors give audio feedback about tilt-based gestures. The feedback explored in this domain all relates to the idea that a continuous feedback loop in interfaces with uncertainty is paramount.

Ripples is an example of an excellent feedback system that is being used in a real interface, (Wigdor *et al*. 2009). Here, it is the input interpretation, and the existence of input that is ambiguous. Touch interfaces sometimes seem unresponsive or error prone due to a lack of basic feedback about how a touch gets interpreted, e.g. as a tap, swipe, or not at all. Small, high fidelity feedback provides nuggets of valuable information to the user about a system's interpretation of the user's input, so that he/she can understand why their input is getting misinterpreted.

Additionally, Li (Li 2009) describes the input SDK on the Android platform which gives feedback to the user about whether their current input is being interpreted as finger movement or as part of a gesture. Again, this illustrates feedback about interpretation ambiguity, rendering two possible bits of information that the user can adjust so that his input gets properly interpreted.

2.2.2.2 Future Input Interpretations

Moving on to providing information about possible input interpretations (in contrast to interpreted gestures), feedforward is an equally important player in the domain of ambiguous input. One clever idea for when and how to provide feedforward stems from the insight that people hesitate when they are uncertain of their actions. In accordance with this, Hover Widgets (Grossman *et al*. 2006) give guidance about possible gestures when the user hovers their pen over a tablet. This is a very clever idea that unfortunately does not get used nearly enough, perhaps because it is too difficult to implement in traditional UI toolkits.

Octopocus is another example of continuous feedforward. As we will see later, this technique overlays all possible gesture completions directly on the interaction, adjusting the opacity of each gesture according to its likelihood. Octopocus presents feedback and feedforward as well as information about gesture likelihoods in a constant loop. Similar techniques present themselves in the tabletop interaction space. ShadowGuides (Freeman *et al*. 2009) provides an interface for teaching touch gestures using the image of the hand as seen by the IR sensor on the Microsoft Surface. Finally, Ghosts in the Interface (Vanacken *et al*. 2008) provide tutorials similar to ShadowGuides when users begin executing actions.

2.2.3 Subjunctive Interfaces – Moving Beyond the Single State Document Model

The previous two sections discussed feedback techniques for communicating a system's interpretation of the user's input: what the user intended to do. Although the systems described may briefly track alternate interpretations of input, these input systems are still maintaining a single state, accomplishing one possible task. Therefore, to consider, *e.g.* alternate floor plans or different design choices, users must undo and redo their actions.

This is called the single state document model (Terry & Mynatt 2002), and makes working with alternate scenarios laborious.

A body of related work provides mechanisms for supporting multiple scenarios. In 2008, Aran Lunzer coined the term "Subjunctive Interfaces" to describe such interfaces that support the exploration of multiple scenarios (Lunzer & Hornbæk 2008). The term is based off of the Subjunct-TV presented in Godel, Escher, Bach (Hofstadter 1979), a TV which played the same football game in many different alternate universes, where turning the TV knob allowed a viewer to switch possible universes. A number of subjunctive interfaces were explored before the term was coined, however.

For example, Side Views (Terry & Mynatt 2002) is an interesting example of an early feedback technique which gave previews of the results of multiple alternative *actions* as opposed to *input interpretation.* Feedback about possible actions gets displayed in an interface as possible outcomes. These lenses into other 'possible worlds' are reminiscent of the Magic Lenses work pioneered by Bier and Stone in 1993 (Bier *et al*. 1993). Igarashi's work on interactive beautification (Igarashi & Matsuoka 1997) and suggestive interfaces (Igarashi & Hughes 2001) show possible drawings (*e.g.* different snap targets, alternate interpretations) in an interactive graphical editor. I will revisit these interactive beautification techniques in further chapters. In a similar vein to Side Views, Parallel Paths (Terry *et al*. 2004) presents a model of interaction that facilitates the generation, manipulation, and comparison of alternate solutions.

The body of work in subjunctive interfaces is especially interesting because it provides a rich set of interaction techniques for interacting with alternatives beyond even what is explored in this thesis. This thesis focuses on fusing and presenting multiple alternatives, one of which is eventually selected to disambiguate intentional input. In contrast, both Parallell Paths (Terry *et al*. 2004) and the RecipeSheet (Lunzer & Hornbæk 2008) provide facilities for manipulating alternatives interpretations directly. The ideas presented in subjunctive interfaces contain a wide range of possibilities for future work.

It is important to note, however, a key difference between the subjunctive interfaces and interaction techniques presented in the previous few sections. While the end results achieved by these methods look similar to what the JULIA toolkit can make, the contribution of the work is different. These interfaces and interaction techniques are contributing just that: interface designs and interaction techniques. The contribution of this thesis is in an architecture that can enable simple construction of all of the interaction techniques presented, and more.

Each implementation in each paper is distinct from the other, and in many cases it is difficult to generalize the implementation of one system to another. Additionally, though a few techniques (Octopocus, interactive beautification, *et al*.) do track likelihoods to determine what to display, tracking of likelihoods is not a core part of these systems. The

probabilistic toolkit does these very two things: it provides a software architecture for tracking interface alternatives, and their likelihoods. As will be seen in Chapter 10, the probabilistic input architecture enables the implementation of much of this prior work under one system.

While the work described above does not focus on the user interface system or architecture level, this is not to say that there is no prior work on toolkits for supporting ambiguous interpretation of input. In the next section I review toolkits that have begun to treat uncertainty in a principled way, and point to the research gap that the JULIA toolkit fills.

2.3 Systems and Toolkits

Input handling systems and user interface toolkits provide a general, reusable way to handle user input. By being general, toolkits are able to support many different types of interactions in a unified way, as well as mix and match different components without worrying how they have to work together. Toolkits typically automate tasks that the programmer would otherwise have to do him or herself. Also, since they are reused for many things, sophisticated/best-practice approaches can be used, which would probably not get built for every application. In addition to providing frameworks for handling many common tasks, toolkits also provide a library that the programmer can directly pull interactive objects from. Current input handling abstractions are by now highly evolved and work very well for what they are intended to do. However, they assume there is no uncertainty in the input, and this causes problems in the presence of uncertain, recognition-based inputs. When input is uncertain, disambiguation of uncertain inputs is often left to the developer. As a result, many applications end up growing in complexity to deal with this uncertainty.

Some systems and tools in the research literature aim to handle this ambiguity that arises during the input handling process. The systems and toolkits in this area fall into three broad categories. First, one set of systems aim to solve the specific problem of *input fusion*: providing disambiguation for multiple possible interpretations of inputs when multiple different input types (*e.g.* speech and gesture) are fused together. The second category of systems aims to provide a reusable structure for handling inputs in the face of alternatives for all input types. The final category of systems aims to treat user input as a completely uncertain process, arguing that the user's intention is a hidden state that can only be observed using noisy sensors, and providing a general method for processing user input by viewing user input as a control loop.

In this section, I will go over the systems and tools that represent more general solutions than the interaction techniques just described, point out the areas that the JULIA toolkit covers, as well as gaps in the research literature that the JULIA toolkit fills.

2.3.1 Disambiguation During Input Fusion

Multimodal input systems hold the promise of providing natural, efficient interaction. However, to implement these interactions, systems must be able to fuse inputs from different sources (*e.g.* speech and touch), understanding that different inputs are somehow related. This problem is commonly referred to as *input fusion,* and programs that perform this task are called *input fusion engines.* Lalanne provides a good overview of many of these systems (Lalanne *et al*. 2009). I will be focusing specifically on the problem of how these input fusion engines perform ambiguity resolution, and then describe the ambiguity resolution architecture of a few specific systems.

A fusion engine can be thought of as a black box that takes several streams of input and combines them. These multiple input streams often come from different data sources, for example one stream could be from voice, and another stream could be from touch. The fusion engine identifies inputs that co-occur in time (for example, pointing to an object and saying the word "on"), determines if two multimodal inputs are compatible, and outputs 'fused events' that represent interpretations of these fused inputs. During this type of fusion, two sources of ambiguity arise. First, it is unclear which inputs should be combined with one another. For example, if a user says "there", and points with two fingers, it is unclear whether the phrase "there" should be combined with one finger, another, or both. Second, individual recognizers may actually have recognition error. For example, the phrase "there" may accidentally be interpreted as "hare" "their" or "there". Three general approaches present themselves when aiming to resolve these ambiguities.

First, many engines use heuristic-based approaches such as preferring one modality over another (*e.g.* speech over gesture). Second is an interactive approach: when multiple interpretations are possible, the engine shows a list of selectable alternatives (called an n-best list). Finally, a small number of engines use a probabilistic approach, computing likelihoods of different interpretations, and resolving accordingly.

The earliest system with directly computed likelihoods was the Quickset system (Cohen *et al*. 1997). In this system, a multimodal integration agent combines inputs from multiple sources (voice and pen), getting likelihood scores for each recognized input. The integration engine then adjusts likelihoods, accounting for compatibilities of inputs specified by an input grammar. The recognized results are then sent to an application for further action. Shortly thereafter, The TYCOON system (Martin *et al*. 1998) provided a fusion method that also computed probabilities, however these probabilities were adjusted based on how closely events occurred in time. In the TYCOON system, decisions about ambiguities were made by looking at probabilities directly. Another system that directly used probabilities to determine actions was the mutual disambiguation architecture provided by Kaiser *et al.* (Kaiser *et al*. 2003). Kaiser's system provides fusion of three different inputs (speech, gaze, gesture). The fusion engine combines inputs by looking for events close in

time that satisfy an input grammar, and multiplies likelihoods of individually recognized results to determine the likelihood of the fused gesture.

One final approach that combines multiple inputs is the XWand system by Wilson and Shafer (Wilson & Shafer 2003). In this system, Wilson uses a Bayesian network to fuse multiple input sources, as well as a user's previous actions, to determine the user's most likely next action. In fact, XWand goes beyond other systems by bringing in a model of prior probabilities on user actions based on a user's previous actions. Unfortunately, the XWand system is custom-built for a specific task and does not provide a reusable component.

With the exception of the XWand, the output of all of these systems is essentially a recognized event combined with a likelihood. Additionally, uncertainty about inputs is not handled end to end—they provide only part of the solution. Most of these systems handle uncertainty primarily at the input interpretation level only, not action. Additionally, in these systems, it is still up to the application developer to manage the uncertainties. The goal of the JULIA toolkit is to handle these ambiguities under the covers, so that developers mostly do not need to worry about alternatives and likelihoods.

2.3.2 Systems for Disambiguation of Recognition-Based Inputs

Moving beyond input fusion and to the handling of ambiguous inputs (the outputs of input fusion), toolkit-level support for tracking ambiguous inputs was pioneered by the work of Mankoff and Hudson. Mankoff's dissertation presents a dispatch process and mediation system (Mankoff 2001) that maintains recognized alternatives for as long as possible, potentially deferring or mediating actions based on the alternatives present. Mediation refers to the process of deciding which action to execute, given a list of actions; it may be interactive (*e.g.* an N-Best list) or automatic (*e.g.* picking the alternative with the highest likelihood). Mankoff provides more detail about these mediation techniques in her 2000 paper about the OOPS toolkit (Mankoff *et al*. 2000). The OOPS toolkit also focuses on providing backwards compatibility with existing interfaces, and has the ability to model uncertainty through the input dispatch and handling process without requiring developers to change what they do. Additionally, the toolkit provides sophisticated notions for when to resolve uncertainty as opposed to when to defer or mediate, and provides a structured approach for how to do mediation.

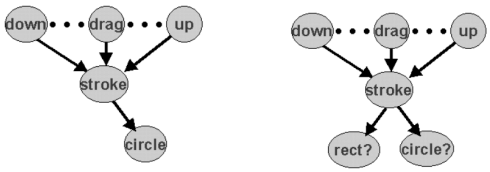


Fig 2.3: The OOPS toolkit tracks not only an event hierarchy, but also alternative interpretations. For example, while a traditional input toolkit would only take the most likely recognized input, a circle (left), the OOPS toolkit would track both the rectangle and the circle (right).

OOPS accomplishes this by tracking not only individual events, but event hierarchies, as well as possible alternative interpretations generated from the event hierarchy. An event hierarchy organizes individual, low-level events into higher level, recognized events. For example, a circle gesture is made up of a down event, followed by a series of move events, and finally by an up event. When input is ambiguous, OOPS performs mediation (either *via* an interface, or automatically), and delays any action until ambiguity is resolved. Developers are able to pick from a variety of mechanisms for resolving ambiguity (called *mediators*), and all alternative input tracking is performed at the toolkit level, so developers don't need to worry about ambiguity themselves.

While the OOPS toolkit covers a lot of ground, one drawback of the system is that it does not directly track likelihoods during the input handling process. While simply tracking input alternatives is good, there are certain advantages to directly tracking probabilities. For example, directly tracking probabilities allows for appropriate display of feedback, and also allows input systems to change their dynamics to provide 'interaction rails' for users, making it easier for them to provide accurate input.

2.3.3 Input as an Uncertain Process; Interfaces as Control Systems

The final system relevant to the JULIA toolkit has a fundamental goal which is similar to the goals of this toolkit: design and build a system which treats input honestly: as a fundamentally uncertain process. The system is described in the dissertation work of Williamson (Williamson 2006), which argues for a unique approach to input handling: that an interface can be described as the continuous control of a point in an 'intention space', and that the dynamics of an input system form a series of control loops that define the communication between the sensed input and the goal space by extending inference over time. Like the work presented here, Williamson uses Monte Carlo methods to implement the inference, feedback and control system for his interactions. The dissertation shows how dynamics of a system can be altered based on the likelihood of outcomes: buttons can be made easier or harder to press based on the system's belief of a user's intent. Additionally, Williamson provides a way to synthesize feedback about the (uncertain) state of a system, as does this toolkit (this work will be presented in Chapter 7). While William-

son's system precisely tracks probabilities, the system described is radically different from conventional user interfaces. This is not necessarily a bad thing, but does add significant barriers towards the practical adoption of uncertainty in interfaces. In contrast, the JULIA toolkit aims to provide some of the benefits of Williamson's work, while maintaining an appearance of having the event dispatch model that is so familiar to developers. While under the covers the JULIA toolkit may be performing complex inference, the interface to developers remains similar to the event dispatch model application developers are familiar with.

Table 2.3 provides an overview of the research gap that the JULIA toolkit fills, by showing how it supports all the features that related systems individually support. Not only does the toolkit provide the extensibility and usability of Mankoff's work, it also enables many of the interaction techniques described earlier to be integrated into one single ecosystem, removing the need for customized code and repetitive logic, and opening up probabilistic input handling to a wider audience.

	Provides Mechanism for Dispatching Inputs	Familiar to Developers	Tracks Alternatives	Tracks Likelihoods
JULIA	✓	✓	✓	✓
Conventional Toolkits	✓	✓		
OOPS	✓	✓	✓	
Continuous Uncertain Interaction			✓	✓
XWand			✓	✓
TYCOON			✓	✓
QuickSet			✓	✓

Table 2.3 Overview of toolkits and systems for handling uncertain inputs.

2.4 Monte Carlo Approach

My system relies heavily on Monte Carlo methods (Metropolis & Ulam 1949; Hammersley & Handscomb 1964), a broad class of algorithms which rely on repeated random sampling of a distribution to understand how this distribution changes when it goes through some process. For example, Monte Carlo methods are used to predict the output of wind farms, given (uncertain) weather predictions. Monte Carlo methods all share a property that probability distributions are approximated by a set of randomly selected samples (Figure 2.4). Note that each individual sample is definite, but collectively the samples represent a distribution over possible values and their likelihoods.

Given a set of samples that represent a distribution over alternatives for a single variable, we can now make predictions about what would happen to this variable when it goes through some process (which might itself have some uncertain components) by running each individual sample through this process and looking at the resulting set of samples, which represent the resulting distribution.

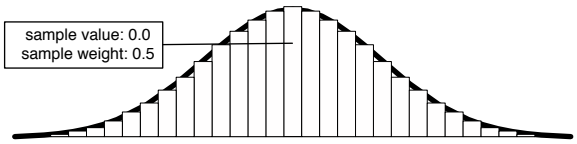


Figure 2.4 Illustration of how JULIA toolkit uses samples to approximate a probability distribution. The underlying probabilistic distribution (thick curve) is approximated by the set of samples, each of which has a weight, indicating likelihood. Note that these samples represent an "ideal" sampling where the distribution is sampled to match the distribution exactly. This is for illustration; only large sets of samples so closely match an underlying distribution.

For example, imagine that a touchscreen can not only sense touch position but also touch type: whether the touch is with a stylus, finger pad, nail, or knuckle (as is demonstrated in (Harrison *et al*. 2011)). In this case, not only is there ambiguity regarding the location of the touch, but also regarding the *type* of touch. Consider the scenario again in Figure 1.1, but this time that the touch has an additional ambiguous field: *touch type*. This touch type represents which part of the finger touches the screen, and can be one of pad, knuckle, or nail. Furthermore, assume that the red button responds only to knuckle taps, and the blue button only to pad taps. Now, consider what an input system would do with an input event at the location in Figure 1.1, which now has 60% probability of being a knuckle and 40% probability of being a pad.

One thing the system could do is pick the most probable touch type (in this case knuckle), and look at the centroid of the touch to determine which button to select. This is what a conventional system would do, yielding no result. Another approach would be to try to compute the likelihood of each button by manually computing the overlap area, accounting for the likelihood of the touch input at each point, then multiplying this by the likelihood of a touch being a particular type (pad or knuckle). However, running this overlap calculation for each button fails to take into account information about what *type* of touch input the other button is listening for. In other words, the fact that the blue button is not listening for knuckle affects the likelihood that the red button should be pressed, given that the likelihood of a knuckle is 60%.

To accurately track the likelihood of each button being pressed, a user interface toolkit (or application developer) would need to take into account the conditional probability of each button given the state of the other button and the properties of a touch event. While this simple example may be feasible to compute, this approach does not scale to the complexity and dynamic nature of modern interfaces. Modern interfaces have many

components which interact with each other, and components get added and removed during program execution, meaning that developers would need to specify dependencies not just between a myriad of existing components, but also between all possible component configurations. Without an automated way to specify all conditional dependencies, a purely analytic approach where probabilities are explicitly tracked is infeasible.

When a stochastic process is too complex to be computed analytically, we often turn to Monte Carlo methods and approximate the probability distribution of the system using a collection of samples. Applied to our button press problem, this leads us to draw a set of event samples from this probabilistic input event, see what would happen to each event sample if it were dispatched as it would be in a conventional user interface toolkit and examine the results. This approach behind the JULIA toolkit, and is the key insight that allows us to turn the highly complex problem of analytically tracking the probability distribution over possible interfaces into a more practical empirical one.

Note that unlike some Monte Carlo techniques, the JULIA toolkit is not performing simulation. Rather, live user input is used to track the probabilities associated with ongoing interactions.

2.4.1 Particle Filters

The probabilistic input architecture could be seen as highly related to the particle filter approach for approximating nonlinear state estimate in stochastic systems (Gordon *et al*. 1993). Thus, there are several similarities between particle filters and the probabilistic input architecture approach, which I would like to highlight in the spirit of promoting understanding of the architecture, and particle filters at large.

Particle filters refer to a set of algorithms for approximating the likelihood distribution of some state space. This state space might be the position of a robot, the contour of a hand, or in the case of probabilistic input, the state of all interactors in an interface.

In a particle filter algorithm, many particles (samples) are used to represent guesses about the system's actual state. For example, a set of particles can be used to represent guesses about a robot's location in a building. At each time step, every particle updates its state based on input given to the system. In the case of a robot, these inputs are controls (drive forward, left, right). In addition to controls, the system also has measurements, in the case of a robot, this might be distance measurements to walls. At each time step, the system then assigns an importance weight to every particle according to the likelihood that the particle at that given state could produce the measurement observed. Particles are then resampled according to their importance weight, reducing the number of particles to a specified maximum quantity.

In a probabilistic input architecture, the “system state” we are approximating is the interface state. When a new input to the system arrives, we represent this input as a series of possible measures (event samples). We then update all interface particles according to these possible measurements. Every interface particle is paired with every possible measurement. As with the update model of a particle filter, state transitions may have probabilities associated with them. Each updated particle likelihood is then multiplied with the likelihood that the input on this updated particle acted on is the ‘correct’ input given the input event. In other words, each updated particle’s likelihood is multiplied with the event sample likelihood. The event sample likelihood is essentially the ‘measurement likelihood’ in a particle filter.

This analogy to particle filters places this dissertation in the larger context of state estimation of stochastic systems, and points to a body of work (optimizations, algorithms) that future explorations in the area of probabilistic input architectures may benefit from.

The next chapter gives an overview of the architecture that uses this Monte Carlo approach to approximate the probability distribution over possible interfaces based on uncertain user input.

3 ARCHITECTURAL OVERVIEW

Nearly all modern user interface toolkits implement user interfaces as a mostly independent collection of interactive objects (interactors) managed by an infrastructure for handling input, producing output, and numerous other tasks. The input-handling component of these systems is mature and works very well for its task. In the next subsection I will describe the structure of this process, then for the remainder of this section use this structure to provide an overview of the system being presented in this dissertation.

3.1 Conventional User Interface Architecture

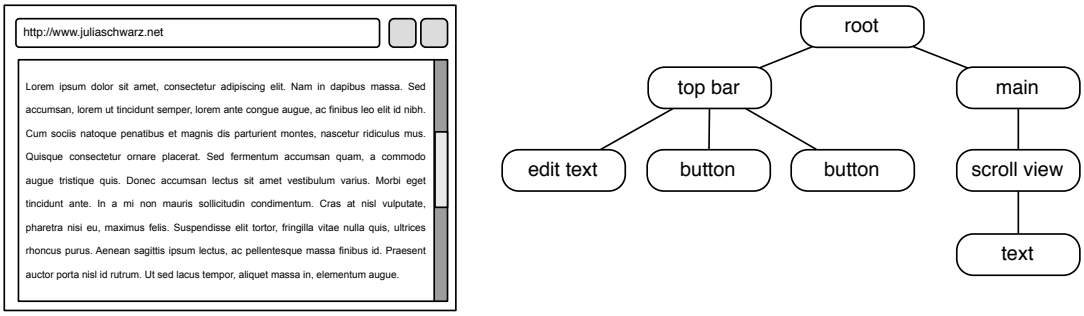


Figure 3.1 Example of interface hierarchy for a simple browser. Left: A sample rendering of a simple web browser. Right: Interactor hierarchy.

The conventional input handling structure can be thought of as providing four major capabilities: (1) *modeling* of inputs, by providing a way to record all the relevant details of what input happened (in the form of *event records*), (2) a process for *dispatch* of those events – deciding which interactor object(s) should receive a given input, (3) facilities for *modeling interactor state*, usually by using a state machine (4) *updating interactor state* based on given inputs, (5) *updating application state* based on inputs, and finally (6) *presenting feedback* to the user about interface state. By giving interactors structured, yet independent, control over how they respond to input, this framework gains uniformity and extensibility.

For the purpose of illustration, consider the scenario in Figure 1.1 under a conventional user interface toolkit structure. In this scenario, the user has just touched down between a red and a blue button. The touch event that would be delivered to the user interface

would be a touch down event, with some location (x,y). Many toolkits also expose the length of the major and minor axes of the touch ellipse. A conventional user interface toolkit would then proceed to dispatch this touch down event to the interactors in the interface.

The interactors in an interface are organized hierarchically, meaning that each element in an interface has a parent and a child. The root of an interface is usually its main window, which has a list of children. Each child may be an interactor such as a button, or a container, which contains other interactors (for example a scrollable view). This hierarchy is often referred to as the *interactor tree*. Figure 3.1 shows an example of an interactor tree for a simple interface. When a new input event comes in, it is recursively dispatched through this structure in a typically depth first manner. In our example from Figure 1.1, the containing window would send the event to each button, which is usually determined by the Z-ordering of the buttons.

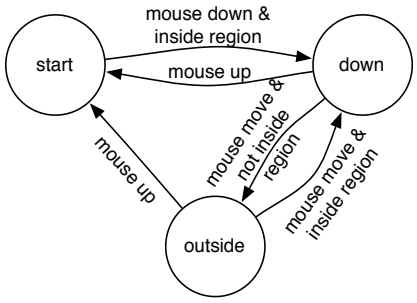


Figure 3.2 Example of state machine for a button

When an interactor receives an event, it may update its internal state based on this event. This is often done using a finite state machine. A finite state machine is a mathematical abstraction commonly used in computer programs. Figure 3.2 shows a state machine commonly used for buttons. It consists of a finite number of states, and rules that specify when to move between states, called *transitions*. A finite state machine can only be in one state at a time, called the *current state*. When a new input arrives, a finite state machine changes its current state according to the rules specified in its list of transitions. In the context of user interfaces, the inputs that a state machine interprets are user input events. For example, common inputs (as seen in Figure 3.2) are mouse down, mouse move, and mouse up. In addition to updating its internal state, an interactor may also update additional application state as a result of user input. The modeling of interactor state *via* a state machine, updating internal state as result of input, and potentially updating application state is managed by the toolkit.

In the example of a yes/no button receiving a touch down event, the button would change from the starting state to the down state. During this transition, the button object would also update other variables to make the button look depressed. In this thesis, I will refer to the sequence of operations performed to update interface and/or application state as *actions*.

After an interactor has executed an action to update its state, the input system needs to know whether to continue dispatching the input event to other interactors in the hierarchy, or stop. When an interactor declares that is has *handled* an event, event dispatch stops. Most buttons handle events, but other interactors such as mouse cursors may wish for dispatch to continue.

In our example, say the user's finger was over the red button. In this case, the red button would perform some action, it would mark the event as handled, and dispatch would stop. This simple structure provides effective techniques for handling inputs. The structure for probabilistic user interfaces is similar, however at every step the toolkit tracks not a single definite instance, but a set of instance samples which represent a distribution over possible interfaces.

3.2 Probabilistic User Interface Architecture

The probabilistic user interface architecture accomplishes the same tasks as a conventional input: input modeling, dispatch, state update, and rendering feedback. Figure 3.3 gives an overview of the input architecture. Developers represent interfaces using the same hierarchical structure as in conventional interfaces. Internally however, the JULIA toolkit maintains a distribution over possible interfaces, and their likelihoods, during each phase of the input dispatch process.

One key difference between the conventional and probabilistic architecture is that the JULIA toolkit does not immediately update interface state when an input is dispatched, but rather gathers all possible update operations (called action requests) and decides which actions, if any, to execute. A second key difference is that the JULIA toolkit includes a mechanism for specifying interactor behavior using probabilistic state machines, a powerful abstraction that easily enables behaviors such as tracking alternate input interpretation and incorporating Bayesian inference. These two differences add to both the complexity of the architecture and the power of the resulting interfaces that the probabilistic architecture enables.

To accomplish this set of tasks, the JULIA toolkit relies on the Monte Carlo approach described earlier to turn the problem of tracking a distribution over alternate inputs, states, and actions into a more manageable, deterministic form. This approach abstracts away as much probabilistic reasoning away from application developers as possible. In this chapter I will give a high level overview of each of the components in the JULIA toolkit.

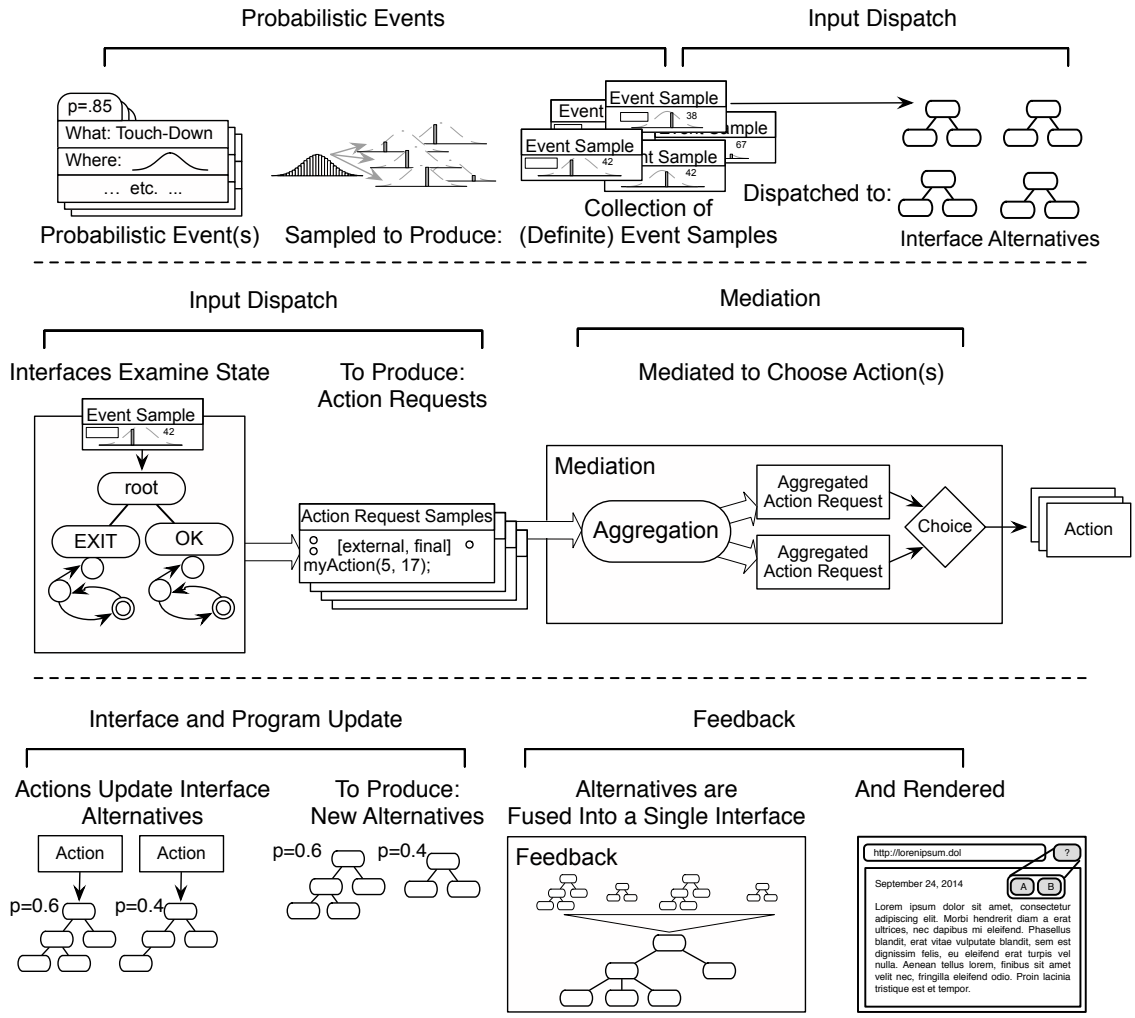


Figure 3.3 Overview of probabilistic input architecture.

3.2.1 Probabilistic Events

To model probabilistic input events, which consist of input alternatives and their likelihoods at the input level, probabilistic input event properties need to be expanded from a single fact to estimates representing a range of possibilities. The JULIA toolkit takes as in-

put probabilistic input events containing event alternatives, labeled with likelihood. The JULIA toolkits then generates a set of event samples, which are obtained by randomly sampling the input probabilistic events. The details of this sampling process are described in Chapter 4. In aggregate, this set of event samples represents the distribution of possible inputs to the system.

3.2.2 Tracking Interface State Distribution *via* Samples

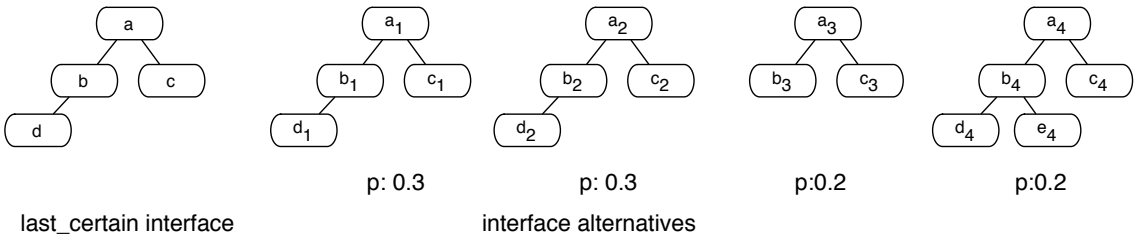


Figure 3.4: The JULIA toolkit tracks a *last_certain* interface (the last certain interface state) along with a collection of possible interfaces and their likelihoods. These are called weighted interface samples.

The JULIA toolkit uses a collection of weighted interface samples to approximate the distribution over possible interface states, along with a *last_certain* interface which represents the last certain state of the interface (Figure 3.4). This approach is inspired by earlier work such as (Schwarz 2010) and (Hudson 1991). The decision to track alternatives at the interface level is key, and contributes to the simplicity of this architecture compared to other approaches such as tracking probabilities explicitly or using samples to approximate the uncertain state of individual interactors (rather than using samples to approximate the state of the entire interface).

Our approach begs the question: why not just analytically track the probabilities of all possible interfaces, rather than using samples? Analytically tracking the probabilities of all possible interface alternatives has the advantage of guaranteeing completeness: no alternatives are left behind. However, as mentioned in section 2.4, this greatly increases the complexity at the toolkit level, and requires the developer to specify conditional dependencies for all other interactor states. Because of the complexity and dynamic nature of modern interfaces, this problem is virtually intractable; in most interfaces there are just too many conditional dependencies between interactors to track. The approach our toolkit takes is to approximate a distribution over interfaces by using a collection of interface samples, and watching how they change as probabilistic input are dispatched. While it may not fully consider unlikely interface alternatives, it has the advantage of making the toolkit tractable and easy for developers use.

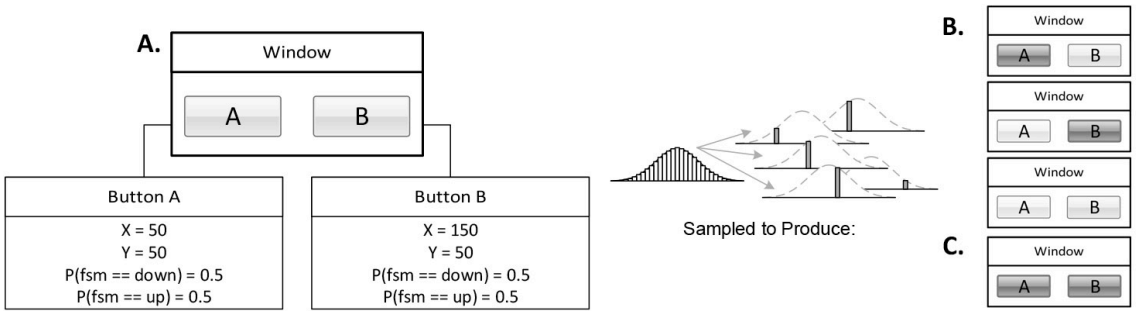


Figure 3.5: Example of sampling over interactor PMF to produce snapshots. A: A window contains two buttons with the given PMF over variables; only one button may be in the down state at a time. B: The system repeatedly samples the PMF over A to generate a set of samples using a naïve algorithm. C: One of these samples is impossible because both A and B are in the down state.

Another question one may ask is why not just use samples to track the probability distribution over the states of individual interactors (Figure 3.5)? In other words, why sample at the interface level? In fact, this is the approach taken in my previous work (Schwarz *et al*. 2011). This however leads to incorrect distribution over actual interfaces, because without knowledge of conditional dependencies is it impossible to know how the state of one interactor affects another. For example, Figure 3.5 shows how tracking only interactor state leads to an impossible scenario of two buttons being pressed simultaneously (assuming only one button can actually be pressed). Additionally, with this model it is very difficult for interactors to take into account the state or values of other interactors in the interface when updating their state, because again the actual distribution over interface alternatives is not known. Sampling at the interface level solves both of these problems: only actual possible interface elements are tracked, and each interactor can trivially examine the state of other interactors within each interface sample.

One drawback of our approach is that the number of interface samples to track increases exponentially as more input events are dispatched. Assuming 1 interface sample and x input event samples at time t, the number of interface samples at time $t + 1$ is proportional to x. This in turn means that the number of interface samples at time $t + n$ is proportional to x^n.

To make this problem tractable, reducing the number of interface alternatives at every step is imperative. Our demonstration applications show that this reduction does not adversely affect the user experience, as in practice most interfaces do not have a large number of alternate states. While we have not spent any effort optimizing our algorithms, our approach is highly parallelizable, as all operations on interface alternatives are independent of one another. Interface and action request reduction can also be optimized significantly. Even without any optimizations, however, our approach is already usable in moderately complex scenarios, as demonstrated *via* the applications in section 10.3.

3.2.3 **Input Dispatch, Mediation, and State Update**

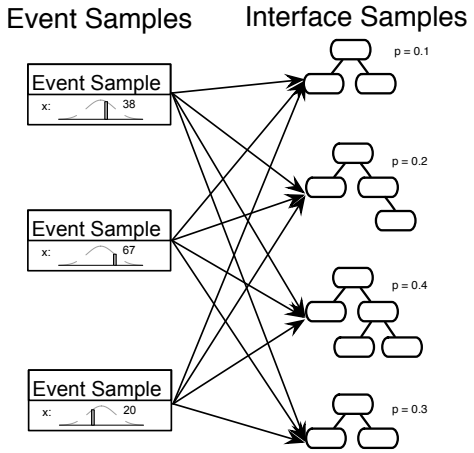


Figure 3.6: During input dispatch, every possible event sample gets dispatched to every possible interface sample.

Conventional input dispatch consists of finding a list of interactors that a particular input event should be dispatched to, and then delivering the input event to interactors in order, until the input event is marked 'handled' by one of the interactors. The user interface toolkit sends the input event to each interactor *via* a callback mechanism. The interactor updates program state in this callback, and is responsible for notifying the user interface toolkit about whether the event should be consumed or not, usually by modifying a property of the input event (for example, by setting the 'handled' property to true). Once an input event is consumed, the user interface toolkit stops sending the input event to other interactors in the list. Each interactor implements a standard interface that encapsulates this process, facilitating the selection of which interactor should receive an event and then the actual delivery of that event.

The JULIA toolkit implements a similar process, with a few key differences. First, rather than a single event, the JULIA toolkit dispatches a list of event samples to every alternative interface it is tracking (initially there is only one). Every event sample gets dispatched to every interface sample (see Figure 3.6). Second, rather than updating program and interface state in response to events, each interactor makes zero or more action requests, each of which represents a possible program state update, and has a likelihood. Because not all interactors handle events, dispatch of a single event sample to a single interface alternative may result in a sequence of actions, appropriately referred to as an *action request sequence*. The JULIA toolkit then decides which action request sequences to accept, reject, or defer (*mediation*) based on the types of action request sequences and their likelihoods. Then JULIA updates the distribution of interface alternatives according to the mediation results, and resamples the set of interface alternatives to ensure that the

number of interface alternatives (*state update and resampling*) to manage remains within some limit. Likelihoods of input and interfaces alternatives are tracked throughout this process.

3.2.4 Describing Interactors using Probabilistic State Machines

When an interactor receives an event, it is responsible for responding to that event. For example, when a button receives a mouse down event, the button must record this information, change how it looks on screen to reflect this change in its internal state, and perhaps execute other application code. More complex interactors such as draggable icons and drawing canvases may have much more information that needs to be tracked.

Interactors in the JULIA toolkit also need to somehow track what state they are in, however this is a much more challenging task, as now an interactor's state, and properties associated with it, is uncertain. For example, if a button received a touch event where the location of the touch indicated that only half of possible locations were in the region of the button, the likelihood that the button was in the down state would be 0.5. Therefore, developers writing code to control buttons must somehow track this likelihood value in the logic of their code. While this may be manageable for an interactor as simple as a button, this task gets much more complex as the interactors gain complexity.

The JULIA toolkit abstracts the task of tracking distribution of interactor state away from the developer by tracking all possible interfaces. This generalization implicitly tracks the distribution over interactor states, appropriately incorporating dependencies between interactors (*e.g.* two buttons cannot be pressed at once).

To further simplify interactor development, the JULIA toolkit provides a mechanism for describing interactor behavior using probabilistic state machines adapted for user interfaces. These state machines allow for multiple transitions out of a single state for some given input, as well as probabilistic transitions: transitions that can be taken with some probability. Given a state machine description, the JULIA toolkit is able to appropriately track the distribution over interactor state without requiring additional effort from the developer. This greatly simplifies the task of tracking state for the developer, and also enables access to powerful features such as leveraging Bayesian inference to adapt interface response to users ('learn as you go'), and incorporating behavior sequences to disambiguate intent (see the Smart Window Resizing example in section 10.1.3).

3.2.5 **Feedback**

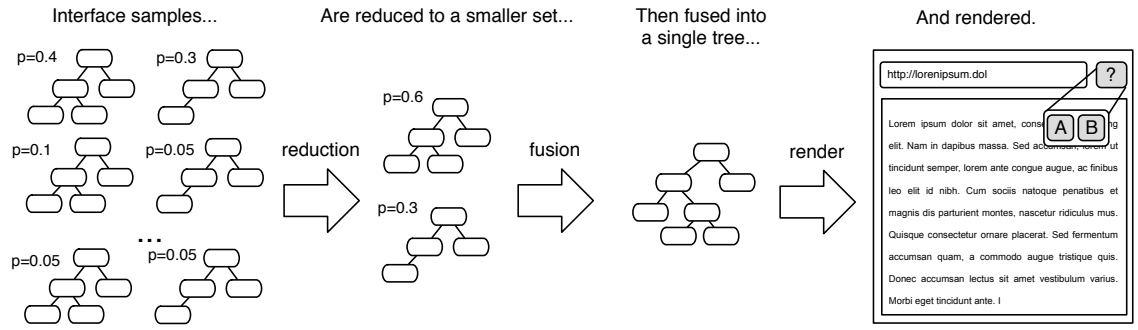


Figure 3.7 Overview of feedback.

The final component of the JULIA toolkit is a feedback system which takes as input the result of state update and resampling and generates meaningful feedback. This is done by analyzing the differences between interface alternatives and generating a new, fused interface which potentially combines multiple alternate representations in ways that promotes understanding of interface state and allows the user to disambiguate their intent. Examples of feedback mechanisms are generating n-best lists displaying alternate interpretations, overlaying alternatives using opacity, and animating between alternate values. The feedback system is designed to be highly flexible, allowing for developers to experiment with different modes of feedback and easily add their own feedback methods.

In addition to communicating system state, the feedback system allows users to disambiguate potentially ambiguous input interpretations. For example, items in an n-best list may be selected to disambiguate input. This can be used in many situations. One example described later on (section 10.3.2) is snapping: When drawing lines in a diagramming application, users can select which control points to snap lines to. This is implemented under the covers by tracking multiple alternate interfaces, each with a different snap target. An n-best list presents the list of alternate interpretations, and users can select the correct interpretation.

In summary, the task of tracking multiple alternative inputs, interface states, actions, and rendering feedback is a task which would be quite difficult to implement by hand. The JULIA toolkit breaks this down into a much more manageable task by using a Monte Carlo approach, tracking the likelihood of possible event samples, interface states, and actions. The JULIA toolkit then uses this information to determine user intent and provide meaningful feedback to users, allowing users to disambiguate when necessary. The next six chapters provide a detailed description of each component of the probabilistic input architecture.

4 PROBABILISTIC EVENTS

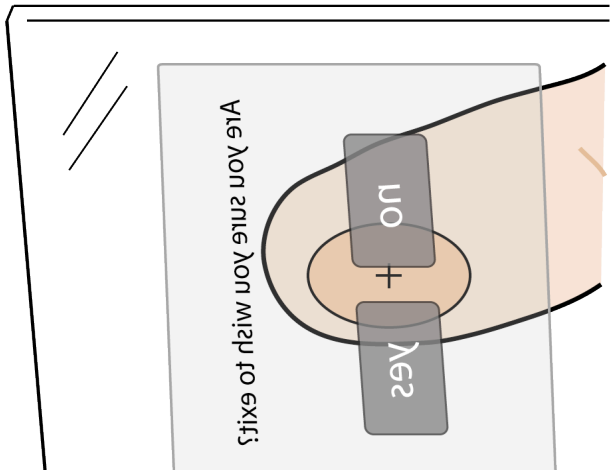


Figure 4.1: Scenario for an ambiguous button press. A user has pressed directly in between the ‘yes’ and ‘no’ buttons in a dialog. In such scenario, most conventional input systems will either execute the ‘yes’ action, execute the ‘no’ action, or do nothing, depending on where the centroid of the finger falls.

In the most modern user interface toolkits, all input is modeled as a discrete sequence of *input events*, each of which records information about some significant occurrence that may affect an interactive program. For example, mouse events are generated every time a user presses down, releases, or moves a mouse, and key events are generated every time keyboard keys are pressed. While the exact form of input events used in systems varies somewhat, most can be characterized as recording the following five categories of information:

1. The **type** of input event that occurred. A record of what occurred, such as “a keyboard key was pressed down”. The type of input determines the structure of the remaining information in the event record.
2. A detailed **value** describing what happened, such as “Key cap #12” for a key press event.
3. **When** the input happened. A timestamp.
4. **Where** the input happened. Most typically the (x, y) position of the primary pointing device.
5. Important **context** associated with the input. A record of other values that might modify the meaning of the input. A conventional example is the state of the modifier keyboard keys (ctrl, alt, etc.).

Most user interface toolkits have no systematic method for recording information about uncertain properties of input events, or possible alternatives. For example, in Figure 4.1 the location of touch input event is ambiguous, and could be anywhere inside (or even slightly outside of) the user's touch area. In a modern user interface toolkit, such as UIKit (part of the Cocoa Touch frameworks for iOS), or Windows Presentation Foundation (the user interface toolkit for Windows-based applications), the user's action will produce a *press* event whose location is a single point, usually computed by taking the center of the touch area. However, this representation fails to take into account the fact that the finger touches an area, not a single point, and also that often a user's intended touch location does not actually match the touch centroid (Henze *et al*. 2011; Weir 2012).

To accurately model the uncertainty behind these new input types, we introduce the notion of a probabilistic input event. A probabilistic input event is a PMF over possible input events. For example, the probabilistic event representing the touch event in Figure 4.1 could be a PMF over a set of touch events whose coordinates can be anywhere near the touch area. This may be represented as a PMF over X and Y. Additionally, the entire input event may have some likelihood. Conventional inputs can also be represented as a probabilistic input event which has only one alternative in its PMF and is marked with probability 1.0.

4.1 Representation via Sampling

```
class ProbabilisticEvent {
 List<ProbabilisticEvent> getSamples(int maxNumberOfSamples);
 float getLikelihood();
}
```

Figure 4.2 Interface for probabilistic input events. A probabilistic event implements a getSamples method which returns a list of probabilistic events representing possible inputs. Additionally, each probabilistic event has a likelihood. The likelihood of a probabilistic event is independent of the likelihood of its samples.

Rather than analytically dealing with the PMF over probabilistic event properties, the JULIA toolkit instead turns this input event into a collection of event samples. Each event sample represents a single possible configuration of the input event. To facilitate this, all probabilistic input events must implement a simple interface, described in Figure 4.2, which generates a random sampling of events. To reduce the number of samples needed to represent a distribution, event samples have a likelihood representing the frequency of that sample in the distribution. As a result, event samples themselves are probabilistic events that have deterministic properties, and a likelihood.

4.2 Example

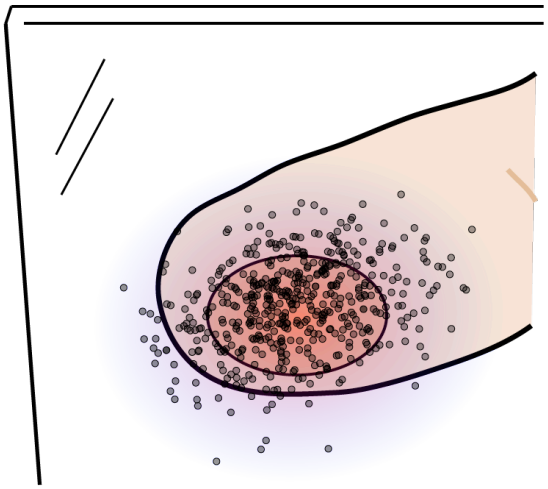


Figure 4.3 Example set of 100 individual event samples that may be sampled from a distribution of touch event location. Each dot represents a sample. Each sample is weighted according to its likelihood of being the actual touch location.

As an example, consider a voice recognition event `voiceEvent` which has two alternatives: “cat”, p(“cat”) = 0.5 and “sat”, p(“sat”) = 0.5. If we call `voiceEvent(100)`, we should expect to receive a collection of at most 100 event samples, each sample representing either “cat” or “sat”. The sum of the likelihoods of “cat” events should roughly equal the sum of likelihoods of “sat” events, however since this is random sampling, these will not necessarily be equal, and subsequent calls to `getSamples()` will yield slightly different results. This distribution could be represented using just two samples (with equal likelihoods), or with 100 samples. The JULIA toolkit does not make strict requirements as to the number of samples generated as long as they are below the maximum specified and they represent a random sampling of the underlying event distribution.

Note that if this voice recognition event had some other uncertain property, for example a skewed distribution amongst alternatives for the speaker (“Mary”, “Jane”, “Bob”), then the distribution would be appropriately distributed between six values, where the normalized sum of sample weights for a *(speaker, result)* pair would be close to *p(result) * p(speaker)* (assuming independence). Normalization here refers to the sum of weighted samples that contain the given *(speaker, result)* pair divided by the sum of all weighted samples. If the input event were not a voice recognition event, but rather a touch event, the set of possible alternatives would be much larger, since the location of a touch has many more possible values. Figure 4.3 illustrates an example of the event samples that might be generated from the distribution of the touch event in Figure 4.1. Just as in the voice detection example, each of the event samples from this touch distribution may

have a likelihood, samples closer to the center would have higher likelihood than those on the edges.

One drawback of the sampling approach is that large numbers of samples are often required to represent distributions where the state space is large: when input events have many variables, or variables that can have many values. In practice, many of these event samples result in the same actions (*e.g.* pressing a button), and so the interface state space gets quickly reduced. Additionally, each dispatch sequence for an *(event sample, interface sample)* pair is independent of others, allowing for optimization *via* parallelization. Our examples demonstrate that we can achieve suitable performance (with little to no optimization) using fewer than 100 samples per event. See Chapter 10 for a detailed overview of examples.

4.3 Summary

In summary, this thesis introduces the notion of a probabilistic input event that represents a likelihood distribution over possible input events *via* weighted samples. While conventional toolkits often throw away valuable information about possible alternate interpretations in order to deal with input dispatch structure, this new representation allows for all information about uncertainty (sensor uncertainty, recognition uncertainty) to be retained into the input dispatch process. This information can be used later on to disambiguate certain edge cases such as when a user presses in between two buttons. Deterministic inputs can still be represented using this notation, demonstrating that this probabilistic representation is a more general representation of input events.

In the next chapter I will describe how the JULIA toolkit dispatches probabilistic input event samples to interface alternatives, aggregating possible interface update operations (and their likelihoods) in the process.

5 INPUT DISPATCH

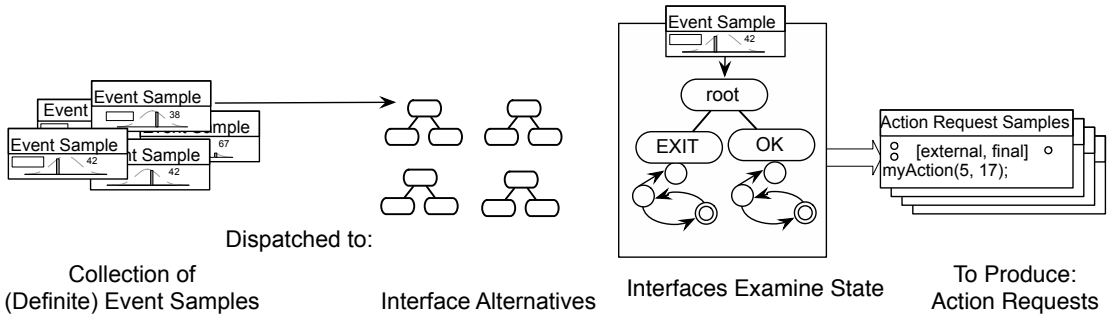


Figure 5.1 Overview of input dispatch process

In the previous chapter I described how input events are modeled as probabilistic events, and how the toolkit turns these events with uncertain properties into a collection of event samples representing the underlying distribution of the input events. The next step in the JULIA toolkit pipeline is to dispatch this list of event samples to the list of alternate interfaces, and get a list of possible update operations that may be executed as a result. Each possible update operation is called an *action request*.

5.1 Action Requests

Action requests represent computation to update the interface and possible program state. In the JULIA toolkit, action requests contain the event sample that generated the request, a function containing the code to execute the update, the interface alternative that generated the action request, and the specific interactor that generated the request.

Some action requests update the state of just the user interface, however other action requests may update program state outside of the interface. For example, an *OnClick* handler for a button may terminate the program or save a file. Because the JULIA toolkit only tracks distributions over interface (and not program) state, executing actions that modify program state must also result in a deterministic interface state: the interface state accompanying such an action must be certain and all other interface alternatives must be removed. Such action requests are called *final action requests* and are marked appropriately. Action requests that only update the user interface state are called *feedback requests.* It is currently the responsibility of interactor developers to determine whether an action request is final or not, however our probabilistic state machine abstraction makes this task very simple. In the future, static or dynamic program analysis

may also be used to determine whether an action request updates external program state, simplifying the task even further for the developer.

5.2 Action Request Sequences

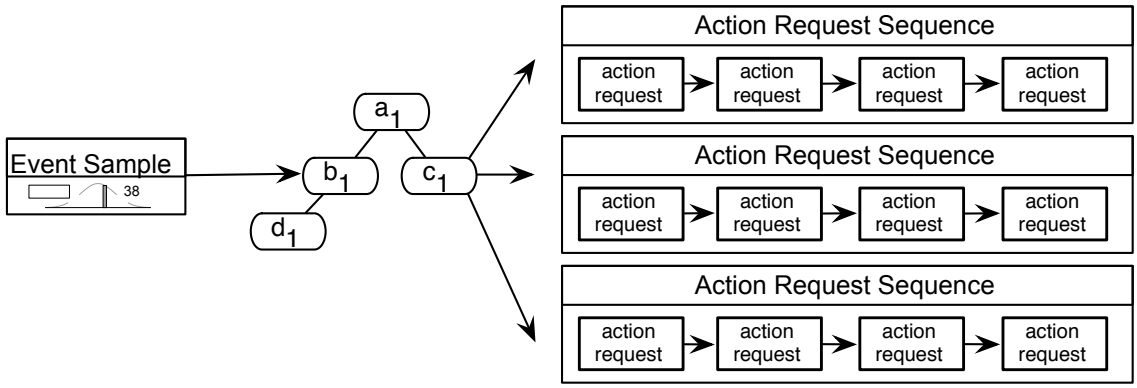


Figure 5.2. A single event dispatched to a single interface may generate several action request sequences.

During event dispatch, several different state operations may be executed as the event is sent from one interactor to another. For example, a mouse event may cause a cursor to update its position and a button to change its background color. Additionally, a single interactor may wish to execute two forms of action, for example change its background color and execute an OnClick handler. This sequence of actions is referred to as an *action request sequence*.

During dispatch, a single interactor may return a list of alternate possible action request sequences. Each action request sequence represents an alternate input interpretation, and will yield an alternate (appropriately weighted) interface alternative. This is especially useful for touch interactions, where the user's intended gesture is only obvious after some time.

For example, consider a touch-sensitive tablet with a scrollable text widget. When a user initially touches down, it is unclear whether he wishes to scroll or select text. Most systems compensate for this by delaying action until a finger has travelled sufficiently far in the horizontal or vertical direction to disambiguate intent. Unfortunately, this workaround is not always effective, and introduces a delay in interface response. Instead, it would be nice if an interface could make an immediate guess as to the user's intent, keep both alternatives around, and disambiguate later when more information was available. We implemented such a scrollable text view as a demonstration, it is discussed in detail in section 10.1.5. This demonstration is implemented as a text view inside a scrollable container (Figure 5.3). When the scrollable container receives an input, it first generates an

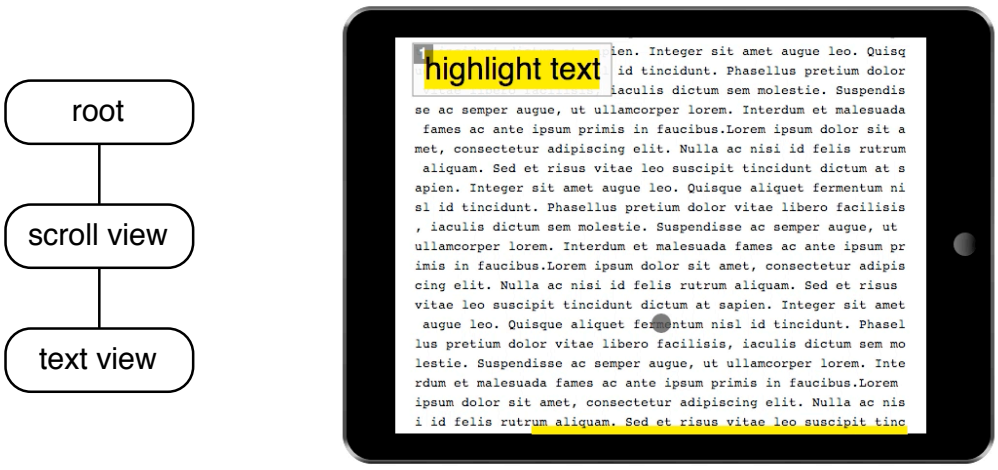


Figure 5.3 Left: Interface hierarchy for scrollable text view demo. Right: Users can disambiguate intent by selecting an alternative.

action request corresponding to a scroll, then sends the event to its children. The children return a list of action request sequences. The weights of these action request sequences are then adjusted so that their sum is equal to the likelihood of the scroll request. If the user's motion indicates a scroll is more or less likely, the sequence weights are adjusted appropriately. The sequences from the children, along with the scroll event, are then returned to the parent. In this example the text view has returned an action request sequence with a single action to highlight text. The scroll view therefore returns two request sequences: one corresponding to highlighting text and the other to scrolling (both sequences actually have just one action request in them). The JULIA toolkit then tracks both operations and their likelihoods. Likelihoods are adjusted by the scrollable container and text view, respectively. Tracking both alteranatives simultaneously allows JULIA to immediately show feedback about the most likely alternative, and switch alternatives if necessary. The user may also explicitly disambiguate by selecting an appropriate alternative (Figure 5.3). By tracking both operations (and their likelihoods), JULIA can both provide feedback about the most likely interaction, and quickly switch to the alternate interpretation if necessary.

5.3 Dispatching Events

As with conventional input systems, JULIA dispatches each input event sample to the interface tree of every interface alternative. The first interactor to receive an event is a container—an interactor that contains other interactors. When it receives an input event, a container sends the event to all children. If a child returns a single action request se-

quence, this sequence is concatenated onto the current action request sequence being tracked.

When a child interactor returns several different action request sequences *c1*, *c2…,cn* (representing *n* different interpretations), the current action request sequence must now branch into *n* new action request sequences (Figure 5.4 step 4). Each of these branches now continues along with the dispatch process independently of other branches, accumulating action requests accordingly. Figure 5.4 illustrates how action requests are accumulated for a single (interface alternative, event sample) pair. The JULIA toolkit provides several extensible container implementations, which we describe below.

The most basic container dispatches input to all children in depth-first order (after first checking for in focus elements). Many of the examples described in this thesis use this container.

Another container type is a scrollable container, containing a single immediate child whose contents may be scrolled. This is the container used in the example in Figure 5.3. When responding to touch input, this scroll container returns action request sequences generated by child elements, as well as a possible sequence where the scroll container itself handles the event (scrolling the contents). The likelihood of a container scrolling its contents relative to not scrolling and instead allowing children to handle events is based off of a simple heuristic (amount of vertical motion), but more sophisticated algorithms are possible.

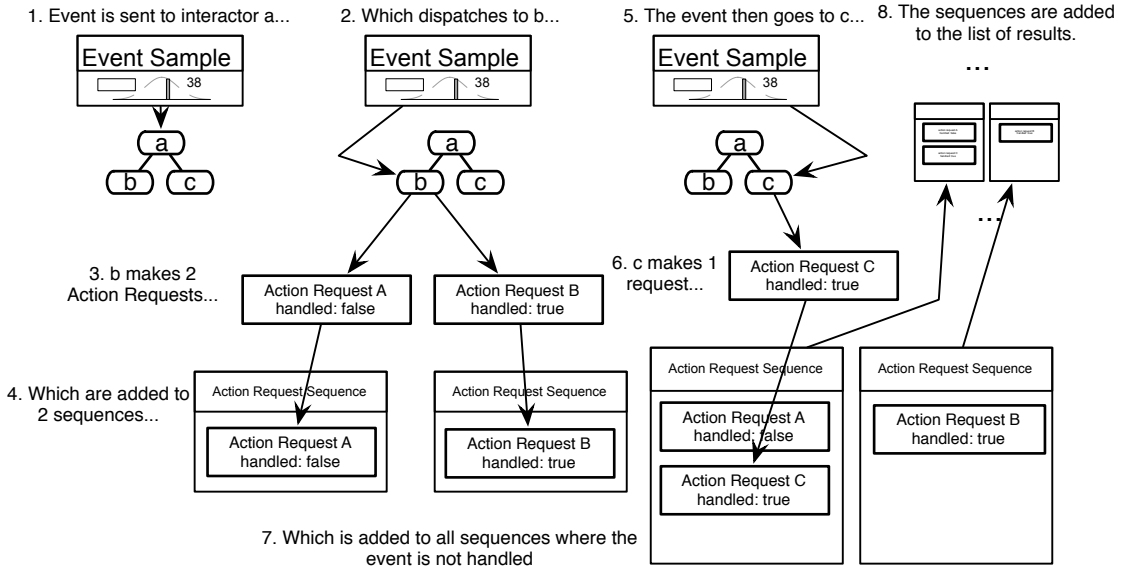


Figure 5.4: Diagram illustrating the sequence of steps used to dispatch events and accumulate action requests.

A final container type implemented in the JULIA toolkit is a container that dispatches an event to its immediate children with identical probability. In other words, it dispatches an event as if all children were at the same 'level'. Every immediate child is given a chance to handle the event, rather than prioritizing the first child. This is especially useful when building a canvas where multiple possible interpretations are possible, *e.g.* a user may be gesturing or drawing a line. This container allows each interactor (the gesture recognizer and line) to be considered independently of one another.

The three container types described allow for a large range of interactive capabilities, but of course many other containers implementing different dispatch policies are also possible, and may be implemented by extending one of the three containers provided.

5.4 Computing Likelihood of Action Requests

After dispatch finishes, each interface alternative yields zero or more action request sequences, each of which represents an alternate possible new state. The likelihood of each action request sequence is multiplied by the likelihood of the event sample and interface sample that it originated from:

*L(ActionRequestSeq') = L(ActionRequestSeq) * L(EventSample) * L(InterfaceAlternative)*

Where *L(x)* represents the likelihood of X, and *ActionRequestSeq*' represents the new action request sequence.

This in turn yields a likelihood distribution of possible state update operations, which is then sent to a mediation process (described in Chapter 7) to determine which state update operations to execute, reject, or defer.

5.5 Example

For example, consider the interface presented in Figure 5.5. This interface has two buttons (A, B), and a set of diagnostic text fields in the corner communicating the number of touches, touch locations, touch state. For the sake of example, assume each is a separate interactor. Figure 5.5A shows the interactor tree. Consider when one event sample overlapping button A gets dispatched. The event is first dispatched to the diagnostics container, which in turn dispatches to each diagnostic. Each diagnostic returns a feedback request, but does not handle the event. The last diagnostic element, however, does return two alternatives, one with the update text "foo" and the other with the update text "bar" the other representing that the user intends to gesture. The diagnostics container view then appropriately zips up these requests (Figure 5.5C) and returns the list of requests to the root container. The root container then sends the down event to button A, which re-

turns a feedback request and DOES handle the event, in both conditions. Button B does not get to see the event. The resulting request sequences contain feedback requests to update the diagnostic fields and change the state of the button.

One final difference to note in event handling is that dispatch of every input event generates some state update operation for an interface alternative. This means that even if no interactors in an interface respond to an event, an update operation representing a self transition (with no variable update) is generated for the given alternative, event pair. This is necessary to avoid killing of interface alternatives that do not respond to a particular event.

5.6 Summary

The dispatch process in the JULIA toolkit has several components which enable the probabilistic input architecture to track interface distribution. First, it tracks a distribution of possible interface update operations while still ensuring that each individual update operation (or sequence of updates) is entirely deterministic. Second, instead of immediately executing update operations, it aggregates all operations and their likelihoods. Finally, all update operations (action request sequences) are independent of one another, ensuring that no alternate interface state affects the state of another interface alternative. In combination, these components allow for proper tracking of possible interface update operations.

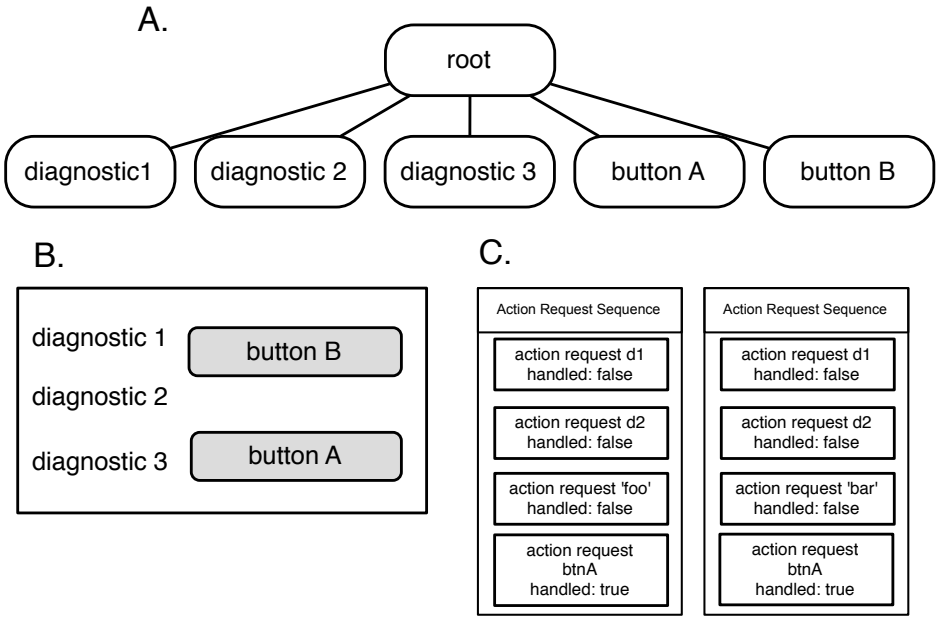


Figure 5.5 Example of dispatch process. A: interactor hierarchy of interface presented in B. C: resulting Action Requests from dispatching an event sample over button B.

Left out of this description was regarding how update operations are actually generated. In other words, how do interactors determine what operations they wish to execute? In the next section I will describe the probabilistic state machine abstraction used in JULIA to simplify interactor development.

6 DESCRIBING INTERACTORS USING PROBABILISTIC STATE MACHINES

When an interactor receives an input event, it is responsible for replying with an action request containing state update operations that it may want to execute. The JULIA toolkit provides a convenient mechanism for tracking state and sending update requests—probabilistic state machines.

6.1 Background

Deterministic finite state machines are a convenient abstraction for responding to input events and tracking interactive state in conventional interfaces. The specific notion and notation of deterministic automata was first introduced by McCulloch and Pitts in 1943 (McCulloch & Pitts 1943) and applied to interaction by Newman (Newman 1968), along with Wasserman (Wasserman 1985). In many cases, the deterministic automata cannot perform sufficiently complex computation to describe the behavior of an interactor. Therefore, machine description languages similar to Augmented Transition Networks (ATNs) (Woods 1970) are used. Though Augmented Transition Networks were introduced to parse natural language, two features make them convenient for describing interfaces. First, transitions in ATNs are conditioned not only on input state, but also on the output of other finite automata (the transitions are recursive). Second, ATNs may read to and write from a register. These two properties, in addition to making the language ATNs describe Turing complete, making them very convenient for specifying interactor behavior.

Many formal languages for describing interactor behavior have spun off from the basic ATN. For example, in (Abowd & Dix 1994), Abowd distinguishes between *input events* which signify an event that should be responded to (*e.g.* a mouse click) from *status* which communicate the change of a variable (*e.g.* mouse position). Abowd then presents a language for specifying the when to change variable constraints in response to input events. Jacob *et al.*(Jacob *et al*. 1999) present a similar architecture where constraints are updated in response to changes in a state machine. In (Abowd 1992), Abowd provides a full overview of formal languages for describing interactor and interface behavior.

While many of the proposed models simplify the description of interactors (and in some cases interfaces) in many scenarios, few have been broadly adopted in commercial user interface toolkits. While the true cause may never be known, one can speculate that mo-

mentum plays a large role. Therefore, the description of probabilistic state machines used in the JULIA toolkit was designed to be as similar to ATNs as possible.

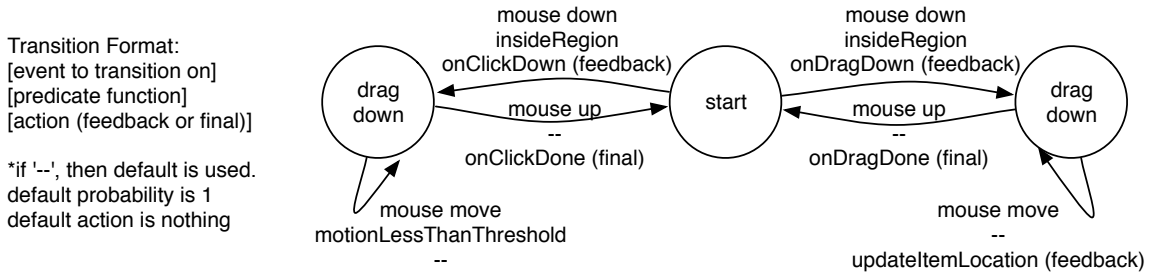
6.2 State Machine Description

The state machines used in the JULIA toolkit are a probabilistic version of ATNs. As with ATNs, transitions are conditioned not on just on input properties but also on the result of functions called predicate functions. Additionally, transition functions may access interactor or interface properties. The JULIA toolkit also introduced a few probabilistic augmentations. First, multiple transitions may exist for identical inputs. When multiple transitions are specified for the same input, multiple action requests (representing alternate representations) are generated for each transition. Second, transition predicates return a score indicating the likelihood that this transition should be taken given the input and current interface state. Third, transitions contain update functions which indicate how program or interface state would be updated in response to the transition. These update functions are then turned into action requests. In the JULIA toolkit, developers specify the state machine for an interactor as a JSON object. Figure 6.1 provides an example. As will be seen shortly, state machine specification is simple and allows for the construction of complex features.

One of the differences between a traditional ATN and the probabilistic machine is that, like a nondeterministic state machine, a single input may lead to multiple resulting transitions. For example, for the state machine in Figure 6.1, several mouse down transitions emanate from the start state. When a mouse down event occurs over an interactor that is in the start state, two transitions are valid: one where the interactor goes to the click_down state, one where the interactor goes to the drag_down state. In this case, the state machine generates two action requests representing two alternate interpretations. In this case the likelihood for both alternatives will be the same, as `is_in_region` will return the same likelihood (recall that predicate functions return the likelihood of taking a transition). However, if the predicate function were different, the likelihood of alternatives may change.

A second difference is the presence of predicate functions. A transition predicate is a condition that must be met for a state machine to take a transition. It is represented as a JavaScript function which returns a number between 0 and 1, representing the likelihood that a transition should be taken. For example, in the state machine in Figure XYZ, the `is_in_region` predicate returns 1 if the event is within the interactor region, and 0 otherwise. The predicate function may inspect properties of the interactor or interface alternative in which the request belongs. In summary, predicate functions are a convenient way to determine whether to take transitions. When combined with nondeterminism and likelihood weights, they can become very powerful, enabling features

such as adapting interactor behavior based on past user actions as well as appropriately responding to recognition likelihoods.




```
{
  start: [
   {
     to_state: "drag_down",
     input_type: "mousedown",
     predicate_function: is_in_region,
     feedback_function: on_drag_down_feedback
   },
   {
     to_state: "click_down",
     input_type: "mousedown",
     predicate_function: is_in_region,
     feedback_function: on_click_down_feedback
   }
  ],
  drag_down: [
   {
     to_state: "drag_down",
     input_type: "mousemove",
     predicate_function: return_true,
     feedback_function: update_item_location
   },
   {
     to_state: "start",
     input_type: "mouseup",
     predicate_function: return_true,
     action_function: on_drag_done
   }
  ],
  click_down: [
   {
     to_state: "start",
     input_type: "mouseup",
     predicate_function: check_motion_less_than_threshold,
     feedback_function: empty
   }
  ]
}
```

Figure 6.1 Sample description for an interactor that is both draggable and clickable. Top: graphical depiction. Bottom: description using JSON format.

A third difference between ATNs and probabilistic state machines is the specification of feedback and final action functions. These feedback and final action functions later get encapsulated into feedback and final action requests. As a reminder, action requests represent computation to update the interface and possible program state. Some action requests update the state of just the user interface, however other action requests may update program state outside of the interface. Action requests that only update interactor state are called *feedback requests*, requests that also update program state are called *final requests*. A transition may contain one feedback function and/or one final function. A transition may also contain neither an action nor feedback function. In this case, only the current state machine state of the interactor is updated when the transition is taken.

6.2.1 A More Complex Probabilistic State Machine Description

Figure 6.2 demonstrates the state machine for a line brush: a brush that can draw either horizontal, vertical, or unconstrained lines. The likelihood of horizontal and vertical lines changes based on the amount of vertical/horizontal motion when a user is dragging. This is the state machine for the line brush in the graphical object editor application presented in section 10.3.2.

This example has two interesting features. First, the start state has several transitions on the mouse down event, each of which represents a different interpretation (line, horizontal line, vertical line). Second, the likelihood of the horizontal and vertical states change based how much horizontal/vertical motion the user is exhibiting. These likelihoods are specified by the functions probabilityIsHorizontal and probabilityIsVertical. For example, if a user is moving their mouse vertically, the likelihood of the horizontal state decreases. As with the scroll view example described in previous sections, this likelihood is implement-

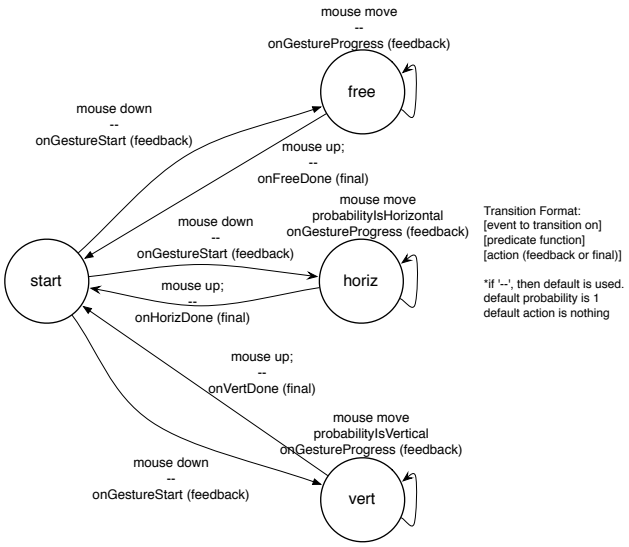


Figure 6.2: Example of state machine description for a line brush. Feedback and final actions are not labeled for sake of clarity. The ‘free’ state represents an unconstratined line.

ed using a simple heuristic (amount of vertical/horizontal motion), however more complex models trained on user behavior (e.g. SVMs, regression) are also possible.

Now that I've provided a description of how state machine is specified, I will explain what goes on under the covers to actually implement such a state machine using this example.

6.3 State Machine Operation

When an interactor described by a probabilistic state machine receives an input event sample, it examines all outgoing transitions from its current state that match the input event. For each transition that should be taken, the probabilistic state machine generates an action request for every feedback and final action defined. Action requests are weighted according to the likelihood value returned by the predicate function specified. If a predicate function returns 0, no action request is added. As a reminder, these action requests are added to action request sequences, whose likelihoods are later multiplied by event sample and interface sample likelihood, as described in Section 5.4. When an action request is added to an action request sequence, the likelihood of the overall sequence is multiplied by the likelihood of the action request being added.

The generated action requests represent a future computation to update the state of the interface (this includes updating the interactor state as well). If accepted, the action request first updates the current state of the interactor, then executes the update functions provided in the feedback or final actions. If no function is specified, only the state of the interactor is updated.

6.3.1 An Example

In the example FSM for the line brush described earlier, consider what happens after a single mouse down event is sent. We are using a mouse event with a single sample here to focus on the alternatives generated by the nondeterministic state machine, and to demonstrate the probabilistic toolkit is valuable even for deterministic inputs such as the mouse. The line brush is initially in the start state.

All transitions going from the start state are examined, and all 3 transitions (free, horiz, vert) have their predicate function return a probability value larger than 0. For each of these transitions, a different action request is generated, with the probability specified by the predicate function. Assume that the transitions to horiz, vert, and free all return 1. In response, the state machine creates three action requests, each with a weight of 1 (remember, the likelihoods of the event sample and interface sample will be multiplied in after the requests have been returned to the dispatcher). Note that the sum of likelihoods is greater than one; these likelihoods will be normalized later. These three requests are all returned by the interactor to its parent. The requests go up the interface tree until

being sent to the dispatcher, which multiplies in event sample and interface sample likelihoods. Because all three requests are feedback requests, all requests are accepted. There are now three alternate interfaces, where the line brush is in the free, horiz, and vert state. All interfaces have the same likelihood. Subsequent mouse move events will serve to adjust the probabilities: vertical motions will cause horiz to be less likely, and vice versa. The likelihood of the 'free' state will not change much.

6.4 Incorporating User Behavior Models *via* Probabilistic Transitions

As an example of the power of probabilistic transitions, I will now describe how the mechanism of probabilistic transitions can be used to easily incorporate a predictive model of user behavior into probabilistic user interfaces. User behavior patterns are a gold mine of information that can be used to not only predict what a user is likely to do, but also perform educated guesses about user intention when their input is ambiguous. For example, if a user presses between the "File" and "Edit" buttons in the middle of working on a document, and the system knows that the user very frequently presses the Edit button in this scenario, the system should be able to guess that a user wishes to press edit, and select this option. Such predictive models are used in systems such as speech recognition and touch typing, however they have yet to move to interfaces more generally because there is no good mechanism for integrating such models. The JULIA toolkit provides a mechanism for integrating behavioral models to determine intended actions, this is implemented using the probabilistic transition mechanism.

The *user behavior model* has two methods of note: `recordTransition(interactor_id, transition_id)` to record that a particular transition for some interactor has been taken, and `likelihoodForTransition(interactor_id, transition_id)` that returns the likelihood that the user's next action is to take the given transition for a given interactor.

Behavior models can vary widely in complexity. The simplest (and default) model returns the same value regardless of input. A slightly more interesting model is to record the last transition executed and increase the likelihood of repeating the same transition. For example, if a user executes a vertical swipe gesture, this model would guess his next gesture is more likely to again be a vertical swiping gesture as opposed to any other action. An even more interesting model would be to use a markov chain based on recorded behavior. Note that here I used the word "action" instead of transition. Really what this behavior model is doing is recording the likelihood of user actions (execute gesture A, press button B). Since some actions do not always correspond to a single transition in a state machine (perhaps several are required), a little extra logic is needed to specify when multiple transitions are equivalent to the same action.

To incorporate behavioral models into the JULIA toolkit, all that is required is to augment the probabilistic state machine mechanism so that it multiplies the likelihood returned from the predicate function for a transition with the likelihood of taking that transition from the behavior model. As a proof of concept, the JULIA toolkit uses the two simple models presented above (uniform distribution and increasing the likelihood of the most recent transition) to provide a very basic form of incorporating user behavior models. Several demos presented later use these behavioral models to adjust the likelihood of actions (see Section 10.1.9).

Note that the models presented use a simplifying assumption: they assume that the likelihood of a user taking a transition from the model is independent of the input event. Of course, more complex models may be integrated. The beauty of the approach used here is that the models may be arbitrarily complex. Developers may integrate arbitrarily complex behavioral models without needing to change any interface code—all that is needed is to change the behavior model. This example highlights the deep power of probabilistic transitions. It allows us to incorporate behavioral models at the user interface toolkit level, something that has never been done before.

6.5 Summary

Probabilistic state machines are a simple way of describing powerful interactor behavior that has yet to be seen in existing interfaces in such a general form. As was demonstrated in Sections 6.2.1 and 6.4, this abstraction enables behaviors such as tracking alternate input interpretation and incorporating Bayesian inference *via* user behavior models. The simple examples of the clickable/draggable buttons and line brush were just the tip of the iceberg: such probabilistic state machines can be used to implement gesture recognizers, complex inference, and more. The examples in sections 10.2 and 10.3 illustrate further capabilities that probabilistic state machines enable.

The next chapter describes how the action request sequences returned by the interactors and dispatch process are examined to determine which actions to execute (if any), and what happens when input is ambiguous. This process is called mediation.

7 MEDIATION

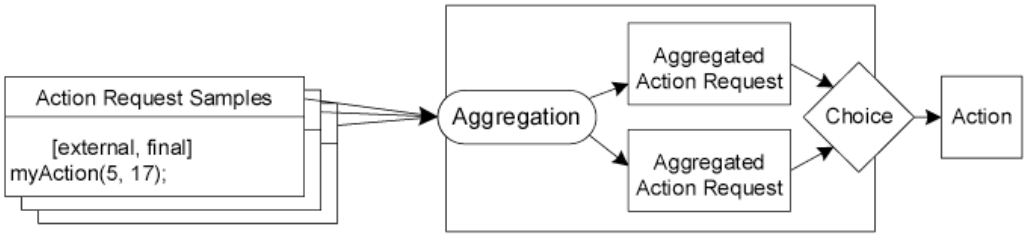


Figure 7.1: Overview of mediation phase to decide which actions should be executed, rejected, or deferred for later.

In Chapter 5 I showed how the JULIA toolkit dispatches a collection of input event samples to a collection of interface alternatives and obtains a list of actions. The next step is for the JULIA toolkit is to decide which actions, if any, to take. In a conventional toolkit, this step does not even exist because at most one action will be selected and it is simply executed. However, here we dispatch events to many potential interfaces, amassing a collection of possible actions and their likelihoods. The reason for this is because we do not know the exact value for the input nor the exact state that the user intending for the interface to be in before the input arrived. Therefore, we are unsure of the exact action. When all interactors have had their say, the JULIA toolkit needs to somehow figure out what should happen, and update program state accordingly.

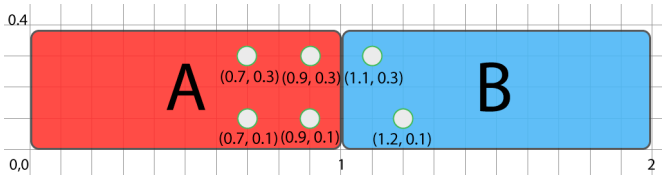
Updating program and interface state will be covered in the next chapter, this chapter focuses only on mediation: the process of deciding which operations to execute. Mediation is performed by a Mediator object which takes in a list of possible action request sequences (output from the dispatch phase) and outputs a list of mediation results: a collection of new action request sequences with a decision to either accept, reject, or defer each action. This component is entirely modular, meaning different mediation algorithms can be used. The mediation algorithms and strategies build off of the original concepts of mediation provided by Mankoff in (Mankoff, 2001). There are two steps to the mediation process: aggregation of actions, and action resolution.

7.1 Action Aggregation

In many cases, the dispatch process will result in multiple action request sequences that represent similar or identical update operations. For example, when 100 event samples (located at slightly different locations) are all sent to a button, the button may return 100

action requests that change the button's state to down and update the button's appearance to reflect that it is depressed. In some cases, the update operation corresponding to the transition may update interface state according to properties in the event sample (for example, moving an item according to the location of the event sample), but in this example the update operation only served to change the button's background color. Therefore, all action request sequences would, after update, result in the same interface in the end. These action requests represent requests that can be combined into a single request whose likelihood is the sum of all 100 original requests.

To identify action requests that can be aggregated, the mediator examines the update code in the action request to determine whether two action requests would, after update, result in sufficiently similar resulting interfaces. For action requests stemming from probabilistic state machines, two action requests are not sufficiently similar if they do not result in the same state machine state. The mediator also analyzes an action request's update code to determine whether properties of the input event are used. Action requests whose code is identical, that do not use event properties, and whose code does not result in different interactor states, may be aggregated. If two action requests have identical code but use an input event's properties, then two action requests may be aggregated only if their event samples are equal. Action requests are aggregated by adding likelihoods (we can add samples because event samples are assumed to be independent of one another), all other properties remain the same. Moreover, the input event attached to each action request does not change when two requests are aggregated—the input event from the first action request is used. Right now input events for similar requests are either not used or identical, so this naïve removal of input events when aggre-



Callback	Target	Likelihood
OnPress	A	0.17
OnPress	A	0.17
OnPress	A	0.17
OnPress	A	0.17
OnPress	B	0.17
OnPress	B	0.17

Callback	Target	Likelihood
OnPress	A	0.67
OnPress	B	0.33

Figure 7.2 Example Action Request aggregation from example in Figure 1.1. For the sake of simplicity, we assuming that both A and B are in the start state with likelihood 1. Top: six touch down events are sent to interactors. Middle: action samples are generated as a result of state machine transitions. Each sample includes information about relevant variables (locations of event), callback, and target. Bottom: Action Requests are aggregated according to target and callback.

gating action requests is sufficient. More sophisticated input event aggregation methods are possible, and are left for future work. Fortunately, action aggregation is implemented *via* a pluggable abstraction that can be replaced with other, more nuanced aggregation methods. For example, an aggregation method could aggregate all action requests that yield the same end state and update the event being operated on to be the mean value of all sample probabilistic events in candidate action request sequences. This would have the effect of generating a 'mean action request' across a list of action request sequences.

Figure 7.2 illustrates the action sample aggregation process for a simple button press. In this example, assume that both buttons are in the start state with likelihood 1, and thus that each interactor has only one sample representing its state distribution. If each interactor had more samples, then the six alternate events could have caused more action requests to spawn (since every event gets sent to every state sample), and then the example would not fit comfortably on the page. Additionally, assume that only six touch down event samples are generated from a touch. Four of the touch down events overlap Button A, which would cause four separate transitions from the start to down state in a button (a separate transition for each of the four event samples that overlap Button A). Each transition spawns an action request to handle the OnPress update function. We will further assume that the OnPress action does not actually use any properties of the input event. Similarly, two of the touch down events overlap Button B, causing two transitions. Once again, each of these two transitions spawns an action request. A total of six action requests get sent to the mediation subsystem. For the sake of simplicity, assume that all event and state samples have equal likelihood. Since all requests originate from the same transition, two resulting requests are aggregated: one for Button A and one for Button B, with A being the more likely request.

7.2 Action Resolution

Since these aggregated actions cannot all be executed, the mediation system must now decide to accept zero or more aggregated actions. All remaining actions are either rejected, or, if the system cannot yet make a final decision, deferred. As we will discuss in the next section, when an action is deferred, the mediator typically shows feedback to seek more information from the user. As with action aggregation, mediation algorithms are pluggable. Developers can use one of a library of provided mediators, or make their own. In the next paragraph I will describe the default mediation heuristic used in my examples.

The default mediation algorithm implemented in the current version of the JULIA toolkit runs as follows: First, the algorithm rejects requests below some developer-specified minimum likelihood. Next, the algorithm determines the likelihood of the most probable final action. It then considers all actions requests with likelihood within a developer specified delta of that maximum. I will refer to this region of within a small delta of the maximum as the "equivalence region". If there is one final action within the equivalence region, that

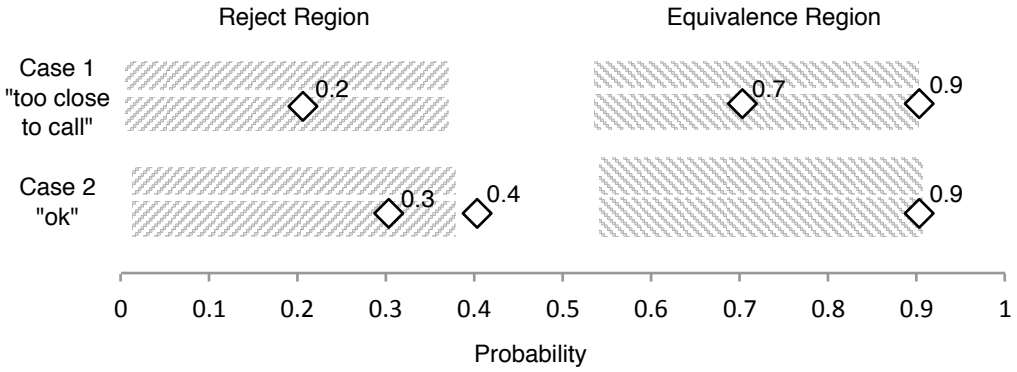


Figure 7.3: Given a set of Action Requests, the mediator sorts Action Requests (diamonds) according to likelihood, then checks to see if any requests are within some small amount (in this case, 0.4) from the maximum value ("equivalence region"). If an Action Request is within the equivalence region, the set is too close to call, and the mediator does not execute action ("deferral"). All requests in this example are final Action Requests.

action is accepted and all other actions are rejected (Figure 7.3). If there are multiple final actions within the equivalence region, the interaction is considered "too close to call". In this case, the mediator rejects all actions below the equivalence region, defers the final actions within the equivalence region, and accepts all feedback actions. If no final actions are accepted, and no final actions are deferred, then all feedback actions are always accepted.

7.3 Deferral

When actions are non final (feedback actions), the resulting interfaces from all update operations are probabilistically tracked until a user either disambiguates explicitly or implicitly through their behavior. However, final actions update program state, and thus need to be resolved. When multiple final actions are too close to call, the mediator defers this action. Deferral is handled in a separate logic path from the regular input loop, as illustrated in Figure 7.4.

The goal of deferral is to give the user an opportunity to disambiguate their intent before a decision is made. For example, if two final action requests are both very likely, it would be best if a user could provide input about their intent before the system decides for them. To accomplish this, the system provides a disambiguation interface rendering each of the deferred alternatives. This disambiguation interface is rendered using the same feedback system as will be discussed in Chapter 9, Figure 7.5 shows an example. Once again, the JULIA toolkit provides an extensible and customizable collection of mediation methods. The default mediation method is to display an "N-best list" choice dialog to the user, a common approach as described in (Mankoff 2001). To communicate the results of

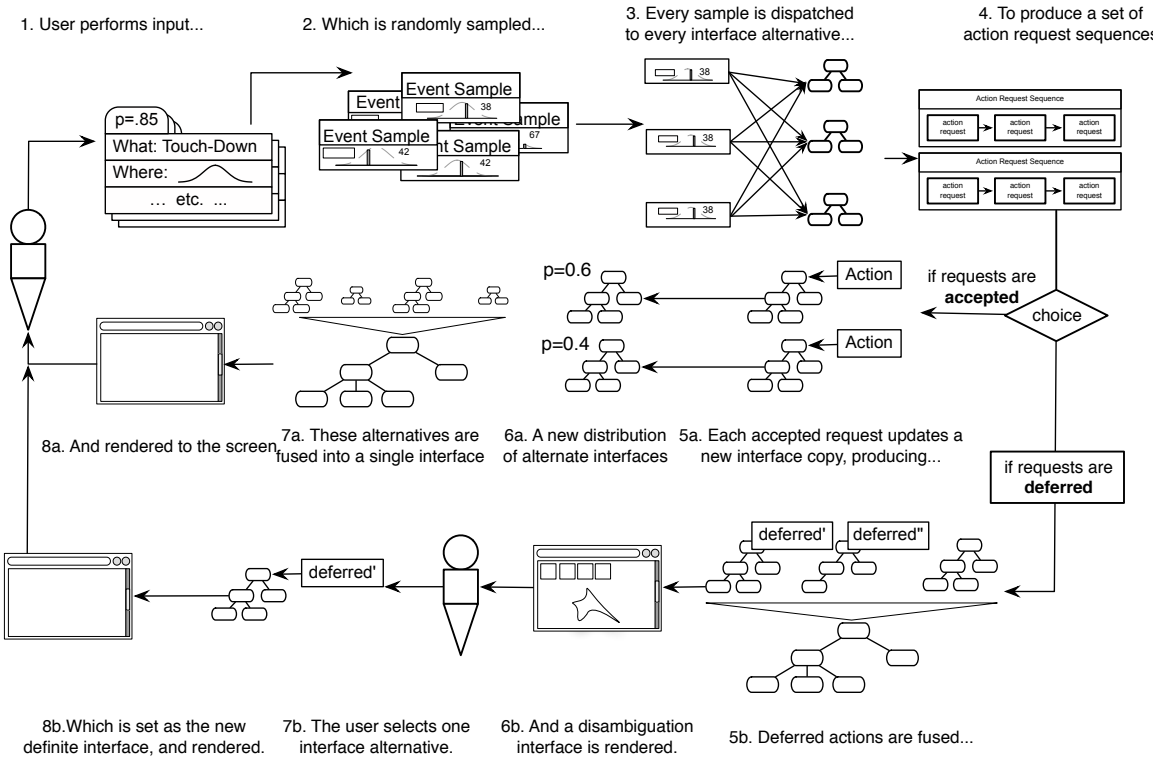


Figure 7.4 Diagram describing input-feedback-action loop, along with separate deferral loop (steps 5b -8b).

final actions, final action requests may optionally include a special draw function, which is called after the alternative is rendered to give a user further information about this action alternative. For example, if a final action may close an application, the draw method for the final action may render the text “close” onto the resulting interface alternative. This does not update the interface in any way (it is a temporary change).

Once the disambiguation interface is rendered, users must choose an alternative. Continuing to interact with the main interface implies selecting the most likely alternative, as this is displayed as the main interface on the screen. The details of ‘continuing to interact with the main interface’ is up to the developer of mediation mechanism to decide, we define this as a click on the main interface. Once the user chooses an alternative, the update code associated with this action request gets executed, the selected interface gets set as the new certain interface, and the regular dispatch process resumes.

The approach currently implemented has a few drawbacks, which we discuss here. Improvements to this approach are possible and we leave this for future work. The primary drawback of this approach is that mediation is forced immediately whenever any ambiguity arises. In other words, this does not allow mediation to incorporate the possibility of alternative interpretations that don’t arise until after mediation is immediately invoked. For example, if a final action arises and another action may arise a little bit later, the se-

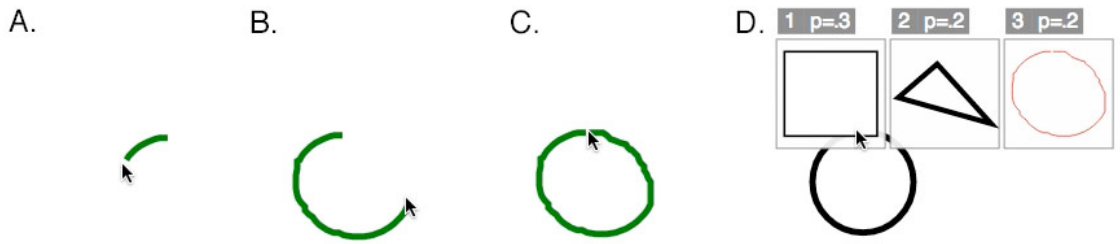


Figure 7.5 Example of deferral. User executes a circle gesture (A-C), several interpretations are likely. The most likely interface is displayed by default, and possible alternatives are displayed as an n best list (D). The user may then select one of the alternate interpretations, or continue interacting using the most likely interpretation.

cond interaction will never be displayed to the user because the first action ambiguity needs to be immediately resolved. This specific area is an area ripe for future work.

7.4 Summary

The mediation process is responsible for determining which operations to execute. The component is pluggable, allowing for alternative, developer-specified mediation to easily be used without interrupting the rest of the interface. When final actions must be resolved, the mediator either resolves these itself, or asks the user for assistance (if it is too close to call). This process of waiting for as long as possible (via feedback actions) for further input has significant advantages over conventional dispatch, which immediately acts on all input events as they occur. By considering multiple interpretations and allowing decision to be deferred until a later time, the mediator allows for the JULIA toolkit to consider far more information than is available in conventional input toolkits when making a decision about interface action.

8 INTERFACE UPDATE

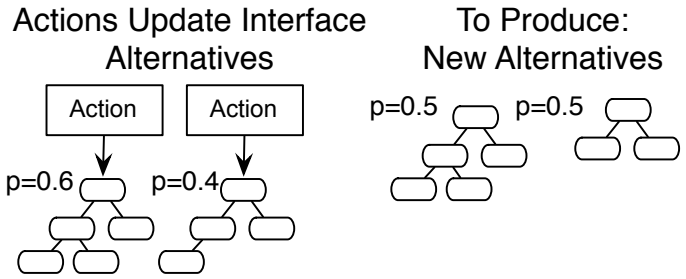


Figure 8.1: Overview of interface update.

After the mediator chooses which operations to execute, the JULIA toolkit needs to update its distribution over interface alternatives to reflect the new update operations. Every accepted action request represents one potential new interface state. The likelihood of an accepted request implies the likelihood of the resulting interface alternative. For example, if two action requests are accepted, this means that two interface states are possible. If only one action request is accepted, only one interface state is possible (this is always the case when a final action is accepted). Cancelled action requests represent an invalid new state, therefore cancelled requests simply do not get executed (and the resulting interface sample does not get put into the new distribution). The interface update process is responsible for updating the distribution over interface states.

The interface update phase can be broken into three parts: cloning interfaces, updating clones, and resampling alternate interfaces. Accepted action requests contain a reference to the alternate interface that generated the request, as well as an action function to update this interface. Interface update is performed by executing the interface update operation on the alternative. Because multiple action requests may refer to the same interface alternative, the first phase of interface update is to clone interface alternatives for every action request.

Note that it is possible for two action requests to come from the same transition from the same interface alternative, where one is a feedback action and one is a final action. To ensure that both are executed on the same interface alternative, one option is to combine requests like this during the action aggregation step (Section 7.1), turning the update function into a function that first executes the feedback update code, then the final update code. This will ensure that cases where action requests from the same transition on the same alternative are updating only one new interface alternative, not two.

8.1 Cloning Interfaces

Cloning interface alternatives is a fairly straightforward process. Every interactor implements a clone method that executes a deep copy of all properties and returns a copy of itself. Interfaces are cloned recursively according to the interactor hierarchy. While this current method is fairly inefficient, the area is ripe for optimization. The sequential nature of the interface alternatives makes the problem suitable for compression using, for example, incremental data structures (Tanin *et al*. 1996). More sophisticated lossless compression methods are also possible (Ziv & Lempel 1977).

One challenge that arises in dealing with interface clones is referring to interface elements. For example, if a developer has a variable pointing to a button in her original interface, this variable will no longer point to the proper button (or rather buttons, since likely there are now several alternate buttons that the variable could point to). If the developer wishes to update properties of other interactors (say change the text of a button) as part of an interface update operation, how does she accomplish this?

One approach is to assign each interactor an ID and refer to the interactor by ID. Every Container View (including the root view) implements a `findViewById()` method which can be used to find and modify views in an update function. Thus, for any update operation that involves other interactors, the root view for that interface alternative first finds the view of interest by ID, then updates its values accordingly.

Another option is to store references to objects themselves directly in the JULIA object that manages alternate interfaces, and modify these references appropriately during the update function call. Other approaches are also possible, including modifying the implementation language or binding the variables to different values according to the interface alternatives being updated. For our example implementation, however, we found that the simple method of tracking interactor IDs worked well.

8.2 Updating Cloned Interfaces

After interfaces are cloned, they must be updated with the appropriate update functions. The process for this is actually fairly simple: every action request is simply executed within the context of the newly cloned interface. Because the update function is executed within the context of an interface copy, no other interface alternatives are modified during this process, only the particular alternative interface corresponding to the new action request is modified. As a result, every alternate interface after the update function is complete represents an alternate representation unaffected by any others. The likelihood of this alternative is simply equal to the likelihood of the corresponding action request (which is in turn computed based on the likelihood of event samples and the result of action request aggregation).

Note that this model of tracking alternate interfaces is very convenient—if at any point the user wishes to select a particular alternate representation as a final representation (such as with final action requests, or through some other mediation process, some of which will be discussed in chapter 9), updating the interface distribution to reflect this is merely a matter of throwing out all other alternative interpretations.

On the other hand, a large number of action requests can lead to too many interface alternatives to manage, causing the number of alternate interfaces to explode exponentially. To solve this problem, the JULIA toolkit reduces the number of interface alternatives, which will be discussed in the next section.

8.3 Interface Resampling

After interface alternatives have been cloned and updated, it is possible that there will be too many alternatives to reasonably track. For example, a simple touch event with 100 samples could lead to 10,000 or more new interface alternatives. The final step that must be taken is resampling to reduce the total sample count. If left unchecked, over time the number of alternate interface samples will grow exponentially. This increase occurs each time multiple event samples arrive and are combined with interface alternatives. However, state distributions can generally be adequately approximated by a limited number of samples.

Before performing resampling, the JULIA toolkit reduces the number of interfaces by identifying sufficiently similar interfaces. The heuristics to identify identical action requests do not always catch all update operations that yield identical (or sufficiently similar) updates. For example, consider an update operation that sets the x position of an interactor to touchEvent.x modulo 10. Now consider two event samples whose x coordinates are 11 and 21. Both update operations will set the x position of an interactor to 1, however the action requests would be considered different because they used a different event sample as a parameter. As a result, an additional reduction step is needed.

8.3.1 Interface Reduction

Interface reduction takes a list of alternate interfaces and reduces this set to a smaller set of interfaces. In contrast to resampling, the aim of reduction is merely to combine similar or identical interfaces. Interface reduction is designed as a pluggable component, the JULIA toolkit provides a library of several reduction strategies. These reduction components are actually re-used in a later phase of feedback, which I will discuss in section 9.2.

To assist with interface reduction, every interactor implements an `isSimilarTo()` method which is used to compare interfaces. For example, two container interactors are similar if all of their children are identical, and if the two containers have

similar properties. Two interactors governed by probabilistic state machines are equal if their current state machine states are equal, and if all other properties are equal. The default interface reduction implementation uses this `equals()` method to determine if two interface alternatives may be combined.

The default interface reduction algorithm in the JULIA toolkit combines similar alternatives as determined by the `isSimilarTo()` method, however other reduction strategies are also possible. For example, if interface samples differ only by the position of one interactor, this set of interface alternatives may be combined into a single interface with that interactor positioned at the mean of the positions of all samples. This reduction strategy is implemented in the JULIA toolkit and is discussed in section 8.3.1. Once again, because this reduction stage is entirely pluggable, reduction strategies may be interchanged and developed as interface developers see fit.

8.3.2 Resampling Possible Interfaces

To avoid an explosion of possible interfaces, the JULIA toolkit ensures that the number of interface alternatives after each input processing step is no larger than a configurable threshold. I will refer to this threshold as MAX_POSSIBLE_INTERFACES. After reduction, it is likely that the number of possible interfaces is still larger than MAX_POSSIBLE_INTERFACES. Therefore, JULIA needs to resample and renormalize the interface distribution.

The resampling algorithm in the JULIA toolkit is pluggable, and can be replaced with any number of algorithms. The algorithm in the JULIA toolkit is a naïve algorithm to perform resampling: it simply takes the most likely MAX_POSSIBLE_INTERFACES interface samples. This approach works well for the applications of moderate complexity presented in this thesis, however has the disadvantage of removing the most unlikely samples. In the future we hope to provide more sophisticated resampling methods leveraging importance resampling. These methods adjust the probability distribution of the samples being drawn to ensure that likely unlikely (but important) samples are drawn, and then adjusts the weights of these drawn samples according to their likelihood in the original distribution. A common technique used in particle filters is sequential importance resampling, introduced by Gordon (Gordon *et al*. 1993). Because of the modular nature of resampling, algorithms such as sequential importance resampling may easily be integrated without disturbing the toolkit architecture at large. After resampling, the resulting set of state samples is a reduced set representing the new state distribution of the interactor.

8.4 Updating to a Certain State

The point of tracking interface alternatives to better infer a user's actual intent. In other words, the higher level goal of the input system is to correctly determine the actual cer-

tain interface state. When the mediation system has sufficient information to disambiguate input, when input is actually unambiguous, or when a user manually disambiguates input *via* interaction (*e.g.* select an item in an n-best list), the toolkit should update its state distribution to reflect having a single, certain state.

In section 3.2.2 I mentioned that the state of interface alternatives is represented by a list of interface alternatives as well as a *last_certain* interface representing the last certain state the interface was in. The JULIA toolkit has several ways of updating this *last_certain* view. First, when a mediator returns a single action request as being accepted, only one interface update operation is possible (all others were deemed invalid). Thus, the particular update operation is executed and the resulting interface is set as the *last_certain* interface, with probability 1. Second, when a user selects a particular alternative as 'correct', he/she is explicitly telling the input systems what the deterministic state is, and thus the particular alternative that should be set as the *last_certain* interface. Correcting for this is merely a matter of setting the alternative selected by the user as the *last_certain* interface and throwing away all other interpretations.

In contrast to other systems that have complex processes for resolving actions (Mankoff 2001, Schwarz 2010), the JULIA toolkit's mechanism is simple and elegant. Selecting a definite alternative is just a matter of updating the last certain interface.

8.5 Summary

The interface update phase is responsible for updating interface samples and downsampling this set of updated interfaces to appropriately reflect the new distribution of interface state. By cloning interface alternatives and executing update operations in the context of these cloned alternatives, the update phase is able to generate new state distributions without altering the state of other alternate interfaces. Downsampling is a key component that makes the operation of the JULIA toolkit feasible, without this the list of alternatives would grow to be far too large. By maintaining a reasonable number of interface alternatives at each step, the update phase ensures a reasonable approximation of the distribution of interface state.

The final phase in the JULIA toolkit is feedback, which is responsible for providing a meaningful rendering of interface state to the user, and allowing them optionally disambiguate their intent. The feedback phase is covered in the next chapter.

9 PRESENTING FEEDBACK

One of the key components of any user interface is a visual representation of the system state to the user. Users must be in a perpetual input-action-feedback loop in order to effectively interact with a computer. The above chapters have covered how input representation, state update, and action change when input is uncertain. This chapter provides an overview of how feedback changes under uncertainty, and provides an architecture for easily making these changes while requiring minimal developer effort.

The primary difference uncertainty brings is that rather than having a single, certain interface, the system is now dealing with a probability distribution over possible interfaces. The feedback system presented in this chapter provides the following facilities to deal with this new challenge:

- **Reduction** – reducing a large set of alternatives to a smaller set of 'salient' alternatives.

- **Identifying Differences Between Interface Alternatives** – an algorithm for identifying differences between interface alternatives.

- **Visualizing Uncertainty and Interacting with Feedback-** an API for building feedback objects that combine interface alternatives (and their likelihoods) into a single interface that communicates system state and allows users to specify which alternative is the 'correct' alternative.

- **Adjusting feedback based on context** – an API allowing developers to dynamically determine what feedback to show.

This architecture allows developers to easily experiment with different feedback techniques by allowing developers to easily switch between feedback methods as well as develop new feedback techniques. Just as with subjunctive interfaces (Lunzer & Hornbæk 2008), developers are able to change the method of rendering alternatives, as well as how alternatives are fused. To demonstrate the power and generality of our architecture, this chapter will also cover 12 feedback techniques already implemented using this system.

9.1 Overview

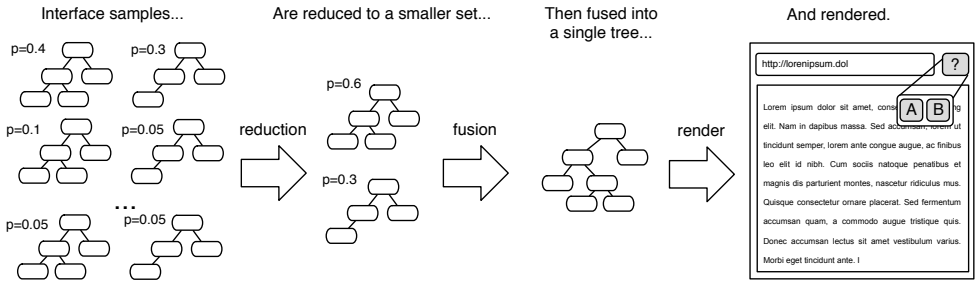


Figure 9.1: Overview of feedback process.

In a conventional interface, an interactor is made up of a set of properties; every time an interface property changes the interface system renders the new interface onto the screen. In the probabilistic input architecture, we now have not a single interface, but a distribution (PMF) over possible interfaces. The goals of the feedback system are threefold. First, the feedback system should support rendering and combination of interface alternatives in a manner that is understandable to users. Second, this rendering should be efficient. Finally, the feedback system should be modular and extensible: developers should be able to pick between a number of different feedback strategies as well as develop their own.

To support this, I designed a modular feedback system whose components may be easily switched out and modified, along with a library of feedback techniques to allow developers to generate understandable feedback of uncertain interfaces. The feedback system has four stages (Figure 9.1). First, the system obtains a collection of interface samples from the result of interface update as discussed in Chapter 8. Next, the system may reduce a potentially large number of alternate interfaces into a representative subset. This subset is then fused into a single combined interface using one of many possible feedback algorithms. Finally, the fused interface is rendered to the screen.

9.2 Reducing the Number of Alternatives to Show

In many cases, the number of interface alternatives exceeds what could reasonably be communicated to the user, even when considering only the most likely interfaces. Therefore, our input system must reduce the number of interface alternatives in an efficient manner, while still representing the initial distribution as closely as possible. Several possibilities present themselves.

One method is simply to pick the N most likely interfaces and show then using one of the feedback techniques described below. While simple, this has the disadvantage of not rep-

resenting a large number of interfaces. For example, if there were 100 alternative interfaces, all of which had a particular interactor in a different location, showing only the top 3 versions would fail to communicate the possibilities. An alternate approach is to reduce the number of interfaces using one of the interface reduction techniques provided in the interface reduction step of interface update (Section 8.3.1).

For example, one such interface reduction algorithm that would be particularly useful is to aggregate all possible values from interface alternatives into a single interface. This reduction algorithm turns each property of an interactor into a PMF over all values the property takes on across all interface alternatives. In other words, now each property (for example, the x coordinate of a location) is not a single value, but a list of values. Interfaces that have new interactors in them would have these interactors added in the final interface.

This reduced interface (or interfaces) can then be fed into different feedback algorithms. As will be seen shortly, these algorithms can then take this list of properties and render the mean, standard deviation, or other aggregated properties as they see fit. One final note about this particular aggregation approach is that instead of combining all values, interfaces may first be clustered and then combined (although we leave the implementation of this in a library for future work).

One major advantage of this reduction step is that it is often doing the same operation as interface reduction in Section 8.3.2. This means that all reduction algorithms implemented for feedback may be used in interface update, and vice versa. Developers may write an aggregation algorithm once and reap the benefits in multiple parts of the toolkit.

9.3 Identifying Differences Between Interface Alternatives

When giving feedback about interface alternatives to the user, it is important to identify which interface components are different from a 'ground truth' (in this case, the last certain state). As mentioned in sections 3.2.2 and 8.4, the JULIA toolkit tracks a *last_certain* interface representing the last certain interface state. Rather than comparing all alternatives to each other, JULIA compares alternative interfaces to this last interface state to determine which components have changed.

Several methods for determining differences are possible. First, all interactors already implement an `isSimilarTo()` method to identify similar interfaces during action aggregation (section 8.3), which can be used to similarly identify specific differences in interfaces. Interactors are marked as 'dirty' when they differ from the *last_certain* interface. Any interactor is dirty if it does not exist in the *last_certain* interface, or if its state has

changed between interfaces. Similarly, containers with different numbers of children (either added or removed) or children of different orderings are marked dirty. These dirty bits can also be set during state update, since state update by definition modifies properties. Rather than comparing all interfaces to the *last_certain* interface, our implementation modifies the dirty bit during state and container update, assuming that interactors can not be changed at other times.

Once differences have been identified, different feedback objects can examine these differences to provide more meaningful feedback to the user, for example highlighting specific objects that have changed.

9.4 The Feedback Object: Specifying What Type of Feedback to Show

In conventional interfaces, interactor developers need to implement their own feedback rendering. Implementing this in the context of probabilistic interfaces leads to extra work, inconsistent visual styles, and errors. One of the aims of the work presented here is to abstract communication of uncertain system state away from the developer in order to facilitate a consistent and reusable way to communicate ambiguity. Importantly, it should be very easy for developers to specify what form of feedback to show, and for developers to write their own forms of feedback either by extending existing feedback algorithms or writing their own.

Rendering feedback in the JULIA toolkit is handled through a feedback object which is responsible for taking interface alternatives and generating a resulting fused interface which is then rendered to the screen. This involves performing reduction, identifying differences, and finally fusing this information into a single interface which will be displayed on the screen. Note that some of these steps, *e.g.* interface reduction and identifying differences are themselves implemented as interchangeable objects, allowing for maximum code reuse. Also note that a feedback object only needs to implement a draw method that takes a list of alternatives, meaning feedback can be arbitrarily simple or complex. For example, later on we will introduce meta-feedback objects, which are feedback objects that contain other feedback objects, and choose which feedback to display based on contextual information.

Developers specify which feedback to show by specifying which feedback object to use in their interface. For example, one form of feedback is to simply render the most likely interface to the screen by calling `draw()` on the root of the most likely interface. This is the default feedback method used in our input system and is what conventional interfaces do.

Figure 9.2 Example of Octopocus-inspired feedback technique of overlaying multiple alternate interpretations using opacity (left), blur (middle) and saturation (right) to reflect probability. The dot is the location of the touch position. The user is drawing an arrow pointing right.

Feedback objects implement a very simple interface: They have a single draw() method which takes a canvas to draw on along with a list of interface alternatives (the result of interface updates). This simple interface allows for a surprisingly complex set of opportunities, which we describe below. Developers may implement their own feedback methods, use one of the many feedback methods provided in the JULIA toolkit, or extend an existing library feedback method.

9.5 Visualizing Uncertainty

As seen in step 3 of Figure 9.1, the fusion step takes a list of interactor trees (each tree represents an interface alternative) and fuses these trees. The simplest option, what in practice conventional interfaces do, is to render the most likely interface. In fact, this is a good option in many cases. However, when several interfaces are highly likely, or when the user indicates indecision (*e.g. via* a dwell), it is worthwhile to communicate alternate possible interface states by visualizing the uncertainty to the user. Below we describe several feedback methods we have implemented to demonstrate the generality of our approach.

9.5.1 Overlay Multiple Versions

Inspired by the feedback techniques presented in Octopocus (Bau & Mackay, 2008), we observed that it may occasionaly be beneficial to overlay interface alternatives on the interface, passing alternatives through various filters. here, all alternatives of an interactor are grouped into a single container, which can render these alterantives as it sees fit. The overlay container can adjust the size or appearance of child interactors (alternate versions of a single interactor) based on probability. For example, it may use alpha compositing, adjust contrast or blur based on likelihood values. Figure 9.2 illustrates our explorations of different forms of feedback for the Octopocus interaction

Figure 9.3 Example of animation feedback. Text fields fade through alternate values.

technique. For background, the Octopocus interaction technique aims to help users learn gestures by overlaying all possible gesture completions on the screen, so that users may better understand how to move their finger to complete a certain action. The feedback visualization techniques explored for communicating probability included adjusting sharpness, opacity, scale, contrast, and rendering separate visualizations (*e.g.* progress bars and text). Another example of overlay in existing work is to draw a 'dashed' or 'impermanent' version of a possible alternative, as shown in Igarashi's interactive Beautification (Igarashi & Matsuoka 1997). To support this, interactors expose a '`drawAmbiguous()`' method which can render an interactor using dashed lines as with Igarashi's work, or in any other form desired.

9.5.2 Animate Between Alternatives

Animation can be used to switch between input alternatives or convey number of alternative interfaces. For example, individual interface elements may jitter or pulse with a frequency according to the number of alternative representations (Figure 9.3).

More practically, individual interface elements may fade between possible values. To the best of our knowledge this form of feedback has not yet been explored in the literature, and provides a new area of feedback to be evaluated and explored by research practitioners.

The advantage of our toolkit is that it allows for these feedback methods to be easily implemented and explored, without requiring any effort on behalf of the interface developer. Developers of feedback may work entirely independently of the interface implementer.

9.5.3 N-Best Lists

The goal of the N-Best list is to quickly show a small number (typically less than 5) of alternatives near the area of interaction. This family of feedback methods is similar to the work presented in Side Views and more generally the work on Subjunctive Interfaces cov-

ered in Section 2.2.3. The most likely interface is rendered in the main interaction area, and interface alternatives are rendered near the point of interaction, or at some other developer-specified location (Figure 9.4). These alternatives can be rendered in several different ways, which we describe below.

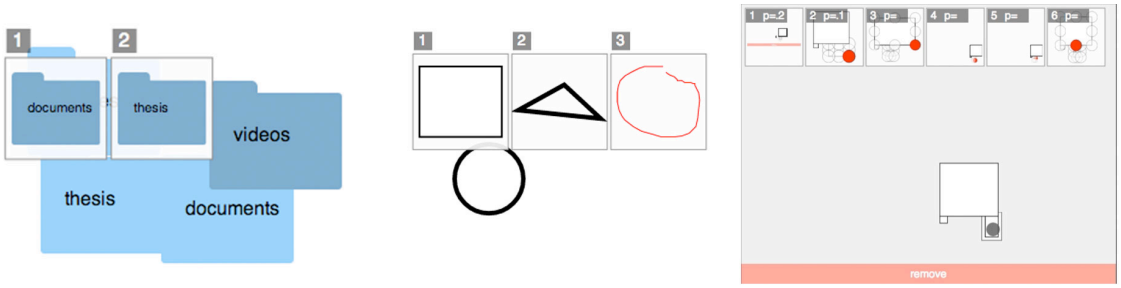


Figure 9.4 Examples of N-Best lists. Left: when it is unclear which folder a user intends to drag, several alternatives pop up. Center: when several gesture recognition results are possible, show alternatives. Right: It is unclear whether a user wishes to drag the rectangle or resize, show alternative options.

First, scaled versions of the entire interface may be rendered (Figure 9.4, right). While simple to implement, this has the disadvantage of being difficult to interpret. One alternative is to render only the interface components that have changed (Figure 9.4, Left, Center). This is done by comparing each interface alternative to the most likely view, marking which interactors are different (have different properties or do not exist in the original), and rendering these to the screen. However, rendering only the changed views removes the interface context and can also be difficult to interpret.

A third approach is to render a scaled version of every interface alternative and highlight the changed regions (Figure 9.5, left). Developers can dynamically switch between any of these feedback methods merely by changing the type of feedback. Many other approaches are possible, for example summarizing changes as text, or adding animation. The existing N-Best Lists can easily be extended to add these changes with minimal developer effort.

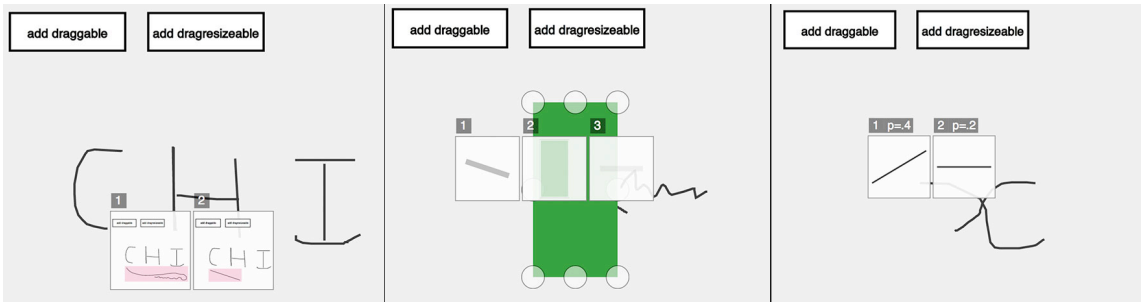


Figure 9.5 Examples of alternate N Best list renderings. Left: Highlight changed regions, Center: Communicate probability using opacity. Right: Communicate probability using text.

Alternatives are sorted by probability and displayed left to right (though of course other display strategies are also possible), and various filters or effects may be applied to alternatives to communicate the likelihood of every alternative, for example opacity, size, and contrast. We also implemented a simple thermometer visualization comparing likelihood (a completely full thermometer indicated probability 1).

9.5.4 Rendering Aggregate Properties

As mentioned in Section 9.2, the results of different interface reduction algorithms may be fed into different types of feedback to create interesting forms of aggregate feedback. For example, we can take the output of the property aggregator discussed in Section 9.2 to render the mean, mode, median, and standard deviation of different interface properties. Figure 9.5 shows an example of rendering the mean and standard deviation of a cursor's position, with a simple overlay of all possible values on the left.

There is a wide body of work on visualization of uncertain information which this aggregate feedback mechanism can implement. Some examples are explored in (MacEachren 1992) and (Pang *et al*. 1997). The beauty of the approach in this toolkit is that it now enables the rendering and exploration of these visualization techniques in the domain of user interfaces, something that has not yet been demonstrated before.

9.6 Interacting With Feedback

In many cases, visualization of uncertainty naturally leads to a desire to resolve this uncertainty. The concept of mediation is covered in Section 6, however because feedback and mediation are so closely linked, a treatment of mediation in this section is warranted here.

As an example, we have implemented several methods for selecting interface alternatives during and after interaction. While most of our examples are implemented using some

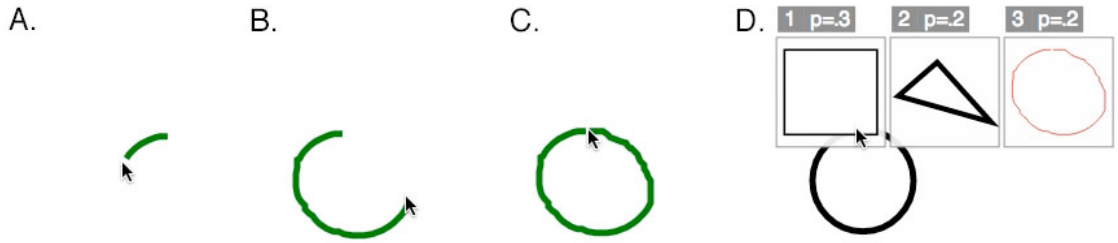


Figure 9.6 Example of interactive feedback. After a user executs a gesture, several alternatives are possible. If she selects the rectangle, this alternative will be selected. If she ignores the n-best list, the circle will remain on the screen.

form of N-Best list, alternative selection may also be possible using different feedback methods.

As discussed in Section 8.4, the implementation for selecting alternatives is very simple: when a user selects an alternative, this alternative is marked as 'certain' and all other alternatives are removed. For example, in Figure 9.6, after a user executes a gesture, several interface alternatives are presented, representing alternate interpretations of her gesture. She selects one of them and this is set as the certain interface.

Additionally, in Figure 9.7 the user wishes to drag a box, but the system interprets her actions as resizing the box. She wishes to change to an alternate interpretation. This is accomplished by dragging his finger/mouse through the proper alternative, as with crossing interfaces. Afterward, the correct alternative is selected. Note that a limitation of this interaction is that the square in the selected alternative may become dragged into a position that is unintended. A second limitation is that an n-best list item may obscure the location where a user wishes to drag the item, causing an n-best list item to be unintentionally selected.

A similar effect can be achieved by pressing a key mid-interaction, or by using voice input. Alternatives disappear after a timeout or after a user completes a gesture. Other error correction strategies such as those reviewed by Bourguet (Bourguet 2006) are possible and easily added in our architecture.

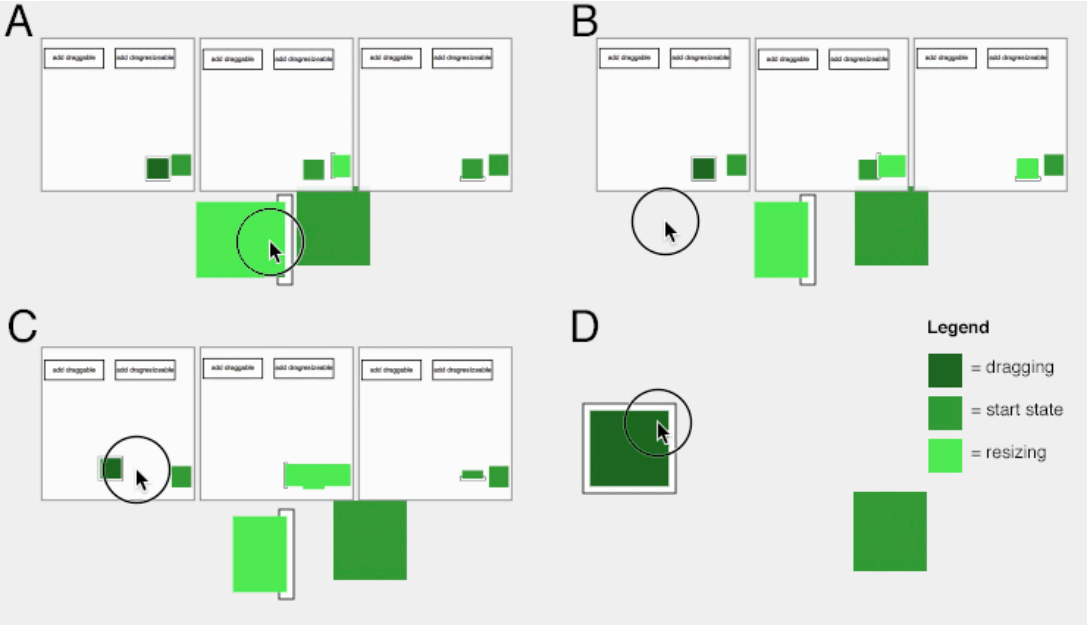


Figure 9.7 Example of interactive feedback. A user wishes to drag the left box, however it is being resized instead (A). *While keeping the mouse down*, she moves over to the leftmost interface alternative, which represents a drag (A dotted line, B, C). Once she crosses over the desired alternative, it is selected.

9.7 Feedback About Future Events

In addition to providing feedback on the current state of an interface (and its alternatives), it is possible to emulate potential future events and render the possible future interfaces resulting from them. For example, input prediction models might predict future pointer locations or future events given past behavior. These input event predictions (and their likelihoods) can be simulated through our system as probabilistic events, and the resulting alternatives presented as feedback to the user and make them available to be selected as shown in the previous section. This could both provide prompting indicating to novices what follow-on possibilities there are, and provide an automated and generalized mechanism for the system to produce shortcut interactions for expert users without having to explicitly build such shortcuts into every part of the interface.

Figure 9.8 shows an example of future feedback discussed in the 'predictive menus' demonstration from Section 10.1.6. This demonstration depicts a menu that seems to predict which items a user is likely to select based on their current selection and their past behavior. Note that the menu interactor is actually nothing special and does not have any prediction logic. The event prediction is handled by a separate component, while feedback is handled by the JULIA toolkit.

Here feedback about possible menu selections can be used to speed up menu selection and also educate users about common menu options. The results of prediction can be communicated using any of the techniques above. In the development of this demo, I was able to experiment with all 12 different feedback techniques presented, Figure 9.8 shows a few of the most promising options.

Surprisingly, I found the opacity overlay (Figure 9.8 bottom left) to work well in educating novices about commonly used menu items, something unexpected that came from easily being able to select different feedback techniques.

There are several ways that future event prediction can be implemented, here we cover two alternate methods. First, after each mouse or key event is dispatched, a special 'event predictor' event source may separately dispatch a new 'prediction' event representing alternate future follow-on inputs. As a result, all future interface states caused by prediction are fully reversible, meaning the resulting alternate interfaces always get generated and rendered. A distribution over possible interfaces is generated from this predicted input event, and results are rendered to the screen. This is the method used to implement prediction in predictive menus. Note that this method does *not* require the development of special interactors.

A second approach is to generate event prediction in response to transitions on interactor states. For example, when the menu interactor makes a particular transition, it may generate a new 'prediction' event that needs to be dispatched to the interface. This is a simi-

lar approach to that used in generation of draggable events discussed in Section 10.3.2. Again, all predictions generate feedback actions, which in turn generates a list of alternate interfaces that are rendered to the screen using the same feedback mechanisms used for current ambiguous alternatives.

Presenting feedback about possible future events is a strong and unexpected capability that comes as a result of the architecture being general and extensible. This approach will become especially useful as event prediction becomes more popular. Our feedback approach makes it easy to communicate possible future events (and their likelihoods) without requiring any additional work.

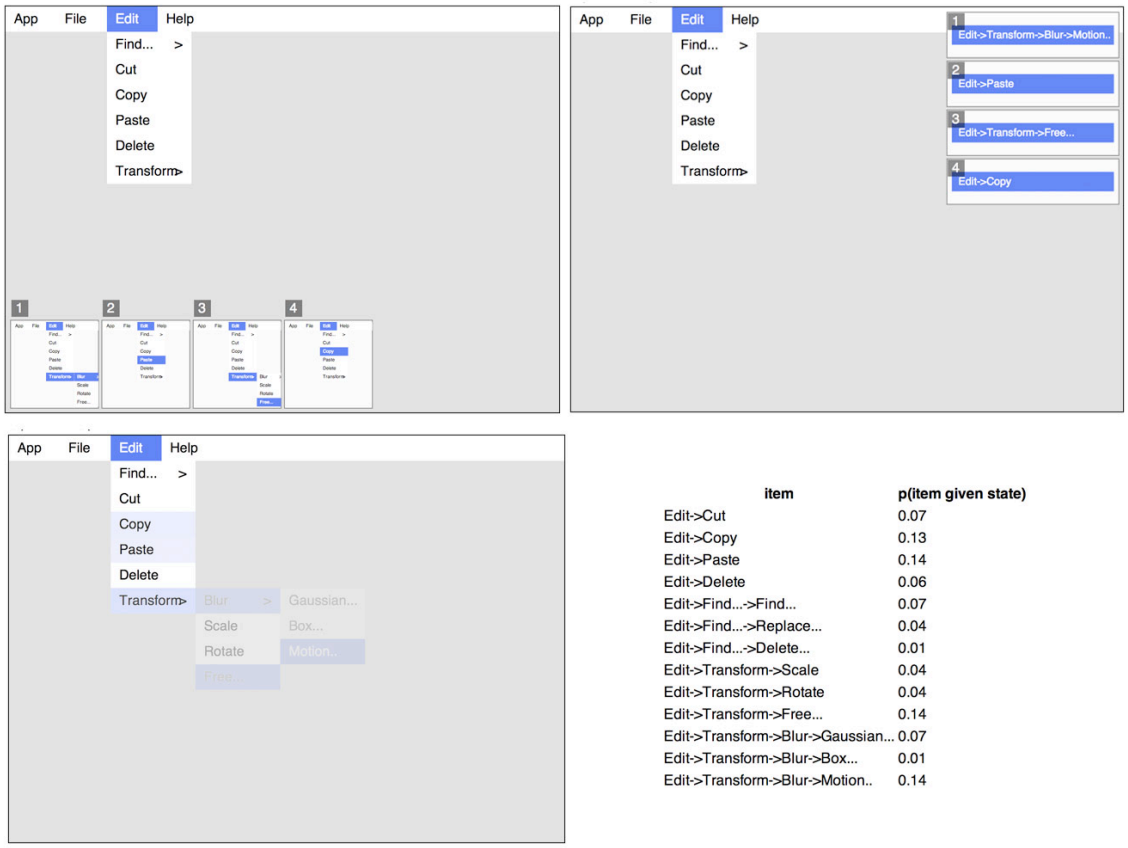


Figure 9.8. Several examples of feedback about future events. The user has moved his mouse over the Edit menu, the likelihood of each subsequent menu item selection is presented in the bottom right and is based on past behavior. Top Left: An n best list of the 4 most likely alternatives is displayed. Top Right: A more compressed n best list is displayed, this time showing a custom rendering of menu commands. Bottom Left: Every interface alternative is overlaid, with opacity reflecting likelihood.

9.8 Meta-Feedback: Adjusting Feedback Based on Context

In many cases developers may wish to alter the feedback method used based on the likelihoods of alternatives, input types used, or even (within an interface) the interactors that have ambiguity.

To support this and allow for maximal flexibility, we introduce the notion of meta-feedback objects: feedback objects which decide what form of feedback to display given system state. Meta-Feedback objects are just feedback objects that contain other feedback objects. Meta-Feedback objects analyze the inputs or alternatives and select which feedback object to use for rendering based on context. Because meta-feedback objects implement the feedback interface, meta-feedback can be combined in any order, allowing for arbitrarily complex feedback.

Below we give examples of several meta-feedback objects implemented in our system as examples.

9.8.1 Inspecting Interface Likelihoods

When several alternatives are likely, developers may want to render another form of feedback, for example an N-Best list (see section 7.5.1). The `DeltaLikelihood` feedback object allows developers to specify a threshold and feedback types indicating what feedback is shown at which threshold levels. For example, if feedback should be shown when there is more than one interface within probability 0.5 of the most likely interface, the developer would specify a delta of 0.5, and two feedback types `MostLikelyFeedback` and `OpacityFeedback`. When the next most likely interface is within delta > 0.5, the most likely interface is shown, otherwise all interfaces within 0.5 of the most likely interface are overlaid using opacity.

9.8.2 User Hesitation

Another scenario when developers may want to show feedback is when users indicate indecision, for example when a cursor is dwelling on the screen. The `Dwell Feedback` object shows the most likely interface until a user dwells on the screen, at which point it displays a new feedback type specified by the developer. This feedback can be combined with the `DeltaLikelihood` object to only show feedback when the user has dwelled and interface alternatives are within some threshold of one another.

9.8.3 Input Type

The `InputTypeMetaFeedback` object allows developers to adjust the feedback rendered based on the type of input event. For example, one could render the mean interface when users are using mouse, and change the feedback method to show only the most likely interface when the keyboard is used. Developers may adjust this using a special feedback type, `InputTypeMetaFeedback`, specifying a mapping from input type to feedback.

9.8.4 Number of alternatives

Certain types of feedback, for example N-Best lists, may not be appropriate when the number of alternatives is large. Therefore, it may be desirable to adjust the type of feedback based on the number of interface alternatives present. Developers may use the `NAlternativesMetaFeedback` feedback type to adjust the feedback based on the number of alternatives.

9.8.5 Which interactors are ambiguous

Not all types of feedback are appropriate for all interactors. For example, a feedback to render the mean interface is appropriate for a cursor but not a text field. Developers may specify a mapping between interactor id (or interactor type) to a feedback object *via* an `InteractorMetaFeedback` object, adjusting feedback accordingly. This feedback type first splits an interface according to the filters specified for a feedback object. It then renders feedback appropriately according to each filter type, using the feedback object specified for that filter type. The resulting renderings are then layered on top of one another appropriately.

9.9 Summary

In this chapter, I have proposed building a rendering system that renders the uncertain state of an interface while allowing developers to stick to a conventional and familiar method for rendering interfaces. Five general forms of feedback were presented, each of which can be easily configured in several ways leading to dozens of different feedback techniques that may be implemented.

10 VALIDATION BY DEMONSTRATION

The aim of the JULIA toolkit is to provide a robust mechanism for handling uncertainty across input sensing, interpretation, and application action in a general, re-usable fashion. To demonstrate the viability of the JULIA toolkit, I explore a total of 17 case studies, in the form of interaction techniques and applications. The 17 case studies discussed in this chapter illustrate that the probabilistic input architecture handles many different kinds of ambiguity, enables rendering of a wide variety of feedback, is easy to work with, can be used to implement traditional deterministic user interfaces, and can be implemented on a wide variety of platforms. Additionally, the demonstrations illustrate that even without any optimizations the architecture performs well enough to be usable in fairly complex interfaces.

This chapter is organized as follows: first, I present nine new interaction techniques developed using the probabilistic input architecture that demonstrate not only that this new architecture provides a foundation for a multiplicity of new interaction techniques, but also that the architecture can handle a wide range of ambiguity types. Next, I present seven feedback techniques that demonstrate the versatility of the probabilistic input architecture. Third, I present three more complex applications to demonstrate that the architecture scales well to full interfaces, and is backwards compatible. Finally, I describe the platforms that successive versions of the toolkit were implemented on, demonstrating that the architecture works well in a desktop, mobile and web environment. Table 8.1 gives an overview of the work presented in this chapter. The demonstrations discussed in these chapters were built using different iterations of the JULIA toolkit with somewhat differing architectures, ambiguity representations and state update methods (see Section 10.4). While in actuality they were built using different iterations, it will become evident that each of these demonstrations could have been built using just the final version of the toolkit. The breadth and code simplicity of these demonstration applications show that this toolkit is not only powerful, but is also simple to work with and can be made to perform reasonably well within the near future.

10.1 Interaction Techniques

In this section I will present nine new interaction techniques developed using the probabilistic input architecture. These techniques demonstrate not only that this new architecture provides a foundation for a range of new interaction techniques, but also that the architecture can handle a wide range of ambiguity types.

		Sensor Ambiguity	Interpretation Ambiguity	Action Ambiguity	Provides Feed-back	Input Modality
Interaction Technique	Two button dialog	✓			✓	Touch
	Interacting With Multiple Sliders		✓			Touch
	Smart Window Resizing		✓			Touch
	Predictive Menus			✓	✓	Mouse
	Bayesian Draw		✓	✓	✓	Touch
	N Best List for Ges-tures		✓		✓	Touch
	Ninja Cursors		✓	✓	✓	Mouse
	Smarter Text Entry	✓	✓			Keyboard/ Speech
Feedback Technique	Octopocus				✓	Touch
	Side Views				✓	Any
	Hover Widgets	✓	✓	✓	✓	Touch
	Future Feedback	✓	✓	✓	✓	Any
	Animation Between Alternatives	✓	✓	✓	✓	Any
	Mean Feedback	✓			✓	Any
	Highlight Only Changed Regions	✓	✓	✓	✓	Any
Application	GesturePaint	✓	✓	✓	✓	Touch
	Interactive Beautifi-cation	✓	✓	✓	✓	Touch

Table 10.1: Overview of completed validation presented in this chapter. Work is organized by type (Interaction Technique vs. Application), and then criteria are presented.

10.1.1 **Probabilistic Buttons**

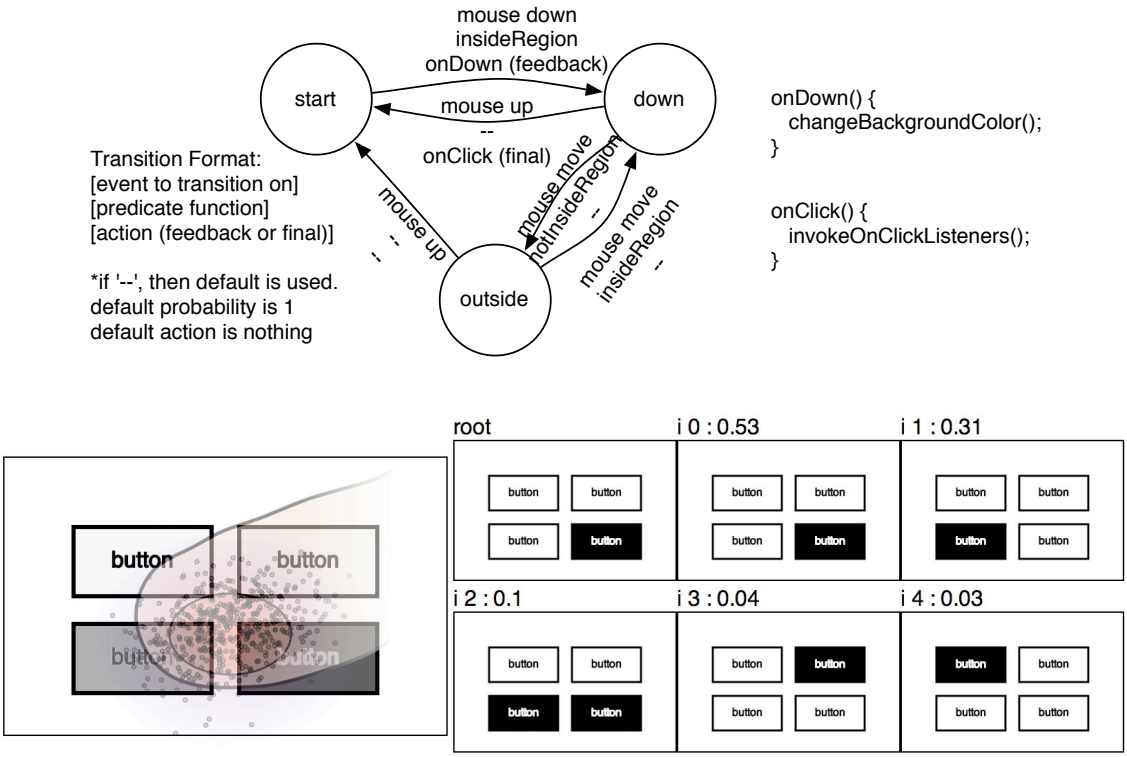


Figure 10.1. Top: state machine for a button. Feedback and final actions are present on the start → down transition and down → up transition. Bottom: Example of interface distribution (bottom right) and feedback (bottom left) from a touch event.

The example of the two-button dialogue runs throughout this dissertation, and this example demonstrates handling uncertainty at the sensor level. Specifically, the touch sensor in this example introduces sensor imprecision, as it is unclear which specific location the user intends to target when he or she is touching the screen. By representing the touch location as a distribution of possible touches, then aggregating all possible update operations, the JULIA toolkit is able to integrate the likelihood of possible button press states over a set of possible locations and determine the button that the user most likely hit. If two buttons are equally likely, then the toolkit does not execute action and waits for further user disambiguation. Note that it is also possible, through the use of probabilistic transitions, to incorporate a prior belief about the likelihood of a button being pressed. The two button dialog demonstrates how a simple button can be made to respond intelligently to user behavior.

As Figure 10.1 illustrates, the state machine logic for these buttons is very similar to that of a standard button. The probabilistic toolkit does all of the heavy lifting regarding tracking of alternatives and likelihoods, so the developer does not need to worry about it. As can be seen from the simple state machine description, this example demonstrates how the logic for one of the most basic interactors remains simple under the JULIA toolkit because the system takes care of tracking likelihoods and alternatives.

The demo illustrated in Figure 10.1 was written in JavaScript for the third iteration of the toolkit (see section 10.4). In this demo, 20 event samples are generated for each touch event, and at most 10 alternative interfaces are maintained. The button demonstration, including the specification of the button interactor, totaled to about 150 lines of uncompressed JavaScript code.

10.1.2 Ambiguous Sliders

Sliders (Figure 10.2) demonstrate handling of uncertainty at the input interpretation and sensor level. At the input interpretation level, one form of ambiguity is target ambiguity: it is sometimes unclear which particular interactor a user intends to interact with. In this demonstration, users can actuate sliders at a distance, meaning they do not need to be directly over the sliders to manipulate them. To determine the likelihood of actuation, each slider computes the distance between it and the current move event, as well as the direction of motion of the finger. As shown in Figure 10.2, all viable target sliders show feedback. This example demonstrates how the JULIA toolkit sends inputs to multiple interactors, computes action likelihoods, and delays action when input is uncertain, allowing instead for the interactors to provide application feedback.

This demo was written for desktop, for the first iteration of the toolkit. This particular iteration did not use event samples (see Schwarz 2010), however I estimate that about 20 event samples per touch event, down sampling to 10 alternate interfaces at each step would do a more than adequate job of supporting this interaction technique. The demonstration code totaled about 200 lines of C# code.

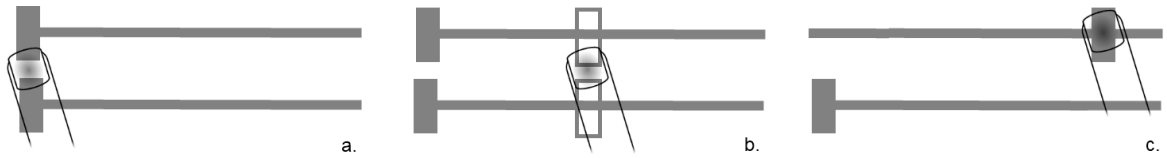


Figure 10.2: Sliders provide feedback when a user moves in between two sliders.

10.1.3 Smart Window Resizing

Smart window resizing demonstrates how my toolkit handles target ambiguity. In this example (Figure 10.3), when a user presses down, it is unclear whether she intends to resize a window by moving horizontally, or move the icon beneath the window. Subsequent motions serve to disambiguate this: horizontal motion means a window resize, while more vertical motion indicates movement of the icon.

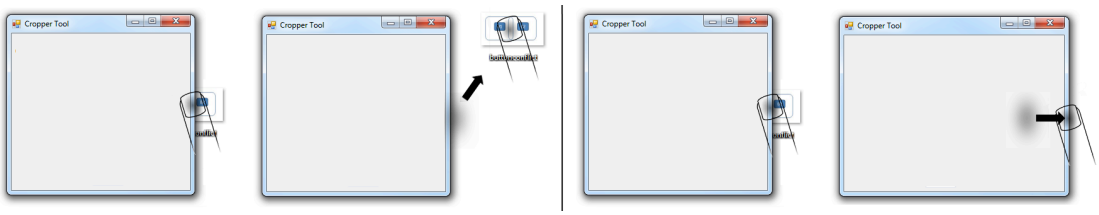
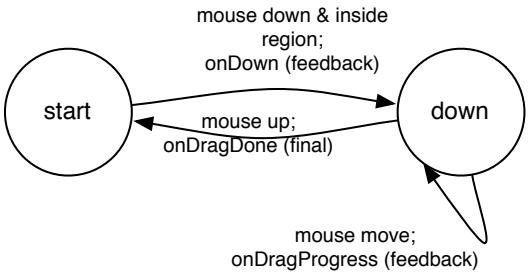


Figure 10.3: Illustration of implicit disambiguation for touch input. **Left:** User presses down and moves diagonally, moving the icon. **Right:** User presses down and moves horizontally, resizing the window instead. When the user presses down, both interactors are equally likely to respond. The user's motion later disambiguates their intention and interactors in our framework respond appropriately.

Figure 10.4 shows the state machine for both the resizable window (left), and the icon (right). When a user initially touches down (Figure 10.3, left), her finger overlaps both the icon and the resizable window, making both the icon and window possible interpretations. As the user moves her finger vertically, the transition weight on the resize transition in Figure 10.4 decreases, and the interpretation that the window is being resized becomes less likely. Appropriate feedback is drawn to the screen. When the user releases her finger, the interpretation for icon movement is sufficiently more likely than a window resize, that this final action gets accepted and a new deterministic interface is set. This demo is implemented in JavaScript, with 20 samples per input event and at most 10 interface samples at the end of each dispatch loop.

Icon Probabilistic State Machine

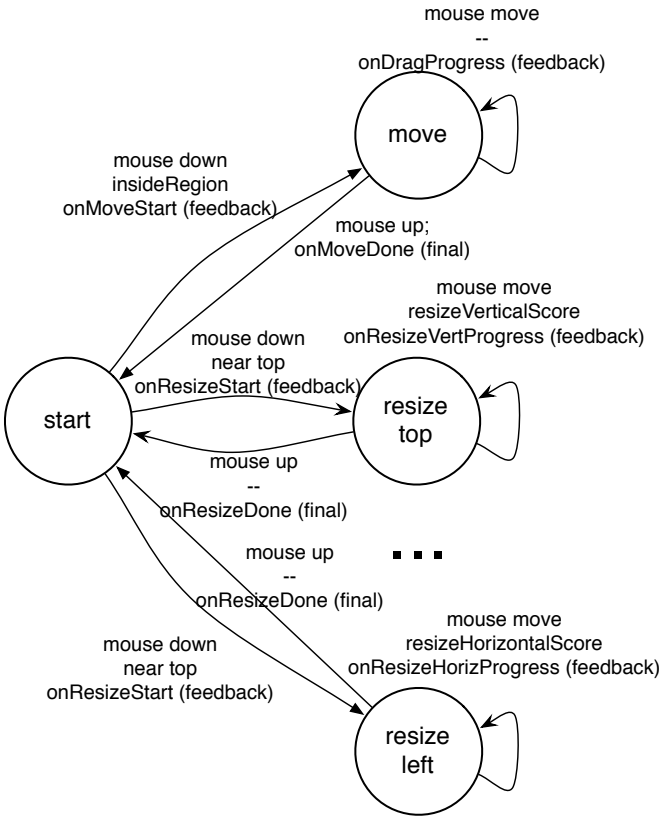



```
onDown() {
  downMousePos = getMousePos();
  downIconPos = getIconPos();
  sendDragBeganEvent();
}

onDragProgress() {
  positionChange = getMousePos -
                        downMousePos;
  iconPos = downIconPos + positionChange;
  sendDragProgressEvent();
}


onDragDone() {
  // send a new event in
  sendDragCompletedEvent();
}
```

Resizable Window Probabilistic State Machine



Predicate Functions:

```
insideRegion() {
  if(mouseInsideBoundingBox()) {
    return 1;
  }
  return 0;
}
resizeVerticalScore() {
  angle = absoluteValue(getAngleOfDrag());
  // remapValue(min_in, max_in, min_out, max_ou
  return remapValue(45, 90, 0, 1, angle);
}

resizeHorizontalScore() {
  angle = absoluteValue(getAngleOfDrag());
  // remapValue(min_in, max_in, min_out, max_ou
  return remapValue(90, 45, 0, 1, angle);
}
```

Action Functions

```
onMoveStart() {
  downMousePos = getMousePos();
  downWindowPos = getWindowPos();
}
onResizeStart() {
  downMousePos = getMousePos();
  downWindowSize = getWindowSize();
}

onResizeStart() {
  downMousePos = getMousePos();
  downWindowSize = getWindowSize();
}

onResizeHorizProgress() {
  positionChange = getMousePos -
                        downMousePos;
  windowSize = downWindowSize + positionCha
}

onDragProgress() {
  positionChange = getMousePos -
                        downMousePos;
  windowPos = downWindowPos + positionChan
}
```

Figure 10.4 Probabilistic state machine description for resizeable window and icon interactors

10.1.4 Text Entry in a Form

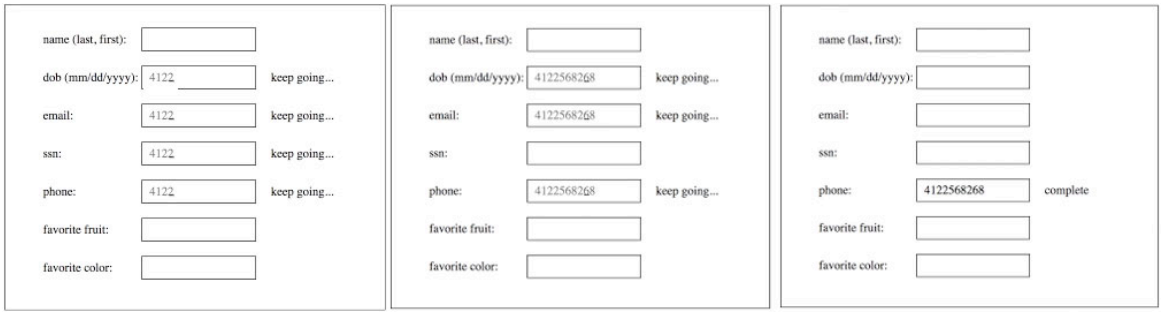


Figure 10.5 Behavior of probabilistic text entry. A user is typing his phone number. Initially (left), email, ssn and phone are possible inputs. Once the user has typed more than 9 characters (middle) ssn is no longer an option. Finally the user clicks the 'phone' field, and this is selected as ground truth (right).

Smarter text entry illustrates how target ambiguity is resolved during the feedback phase. Although the text a user is typing is certain, *where* this text should go may be ambiguous. Many forms don't respond when a user starts typing text without first selecting a text box, and those that do respond often do so by selecting the topmost text box. Things get even more complicated when a speech interface is in use, especially given that speech is often used because of physical or visual impairments that eliminate the ability to use a pointer.

The JULIA toolkit can address this problem with text fields that change their action likelihood based on what content they expect to receive. If several text boxes can receive input, the JULIA toolkit delays action and provides feedback accordingly.

I implemented two examples – smart text delivery and speech text entry. Both examples deliver text to a form with several different fields, *e.g.* name, phone number, and email. Because it is initially unclear which text field should handle input, the JULIA toolkit ends up tracking *n* alternate interfaces when a voice or key event is dispatched, where *n* is the number of form fields. This creates *n* alternate interfaces, each of which has a different text box in focus. Subsequent key events are dispatched to the text field in focus, likelihoods of alternatives are adjusted based on how well the entered text matches what the field expected. As the input becomes less likely, the transition likelihood decreases (or becomes zero). Once the transition likelihood is zero, no interfaces are generated as a result of input and the alternative is removed.

Figure 10.5 demonstrates the behavior of these new text fields. When a user types text without having first selected a textbox, all text boxes which have not yet been filled in show the typed text in gray. A user can then continue typing or select the correct textbox. The textboxes have a special behavior attached to them: If a text box is selected, the al-

ternative corresponding to this text box is set to the *last_certain*, or correct view. Note that this is all handled by the toolkit, the text box merely sends a request to finalize interface ambiguity. When a user presses enter, all text fields send a final action requests with a corresponding likelihood.

For speech input, recognition results often contain multiple uncertain interpretations, such as “2” and “q”. All uncertain interpretations are sent to all text fields and shows feedback appropriately, as before. The likelihood of a recognition result is propagated to the likelihood of different interface alternatives. When a user says the word ‘next’ or pauses long enough, the text fields send final action requests, and the most likely interface alternative (corresponding to the most likely text field) is selected.

The form demo also incorporates a simple Bayesian behavioral model to further disambiguate results. Since people tend to fill out forms top to bottom, the topmost entry is initially more likely than others. Also, given the index of the last entered element, the next highest element is more likely than other elements.

The probabilistic input form demonstrates three features of the JULIA toolkit. First, the example shows how the likelihood scores of recognition events are directly translated to interface likelihoods rather than immediately being thrown out. This example also demonstrates that the JULIA toolkit works well for new input types such as voice input. Finally, the example demonstrates how voice recognition uncertainty can be combined with a behavioral model to further reduce uncertainty.

For this demonstration, 20 event samples are tracked, and interfaces are down sampled to 10 alternatives at each step. This demo was implemented using the third iteration of the toolkit, and totals about 200 lines of JavaScript code.

10.1.5 Rapid Disambiguation Between Text Selection and Scrolling for Touch Interfaces

Selecting text can be very cumbersome on mobile devices, requiring a press and hold interaction. The reason for this is that when the finger touches down it is unclear whether a user intends to interact with the scrollable content or the scroll view itself. Touch-based operating systems take a seemingly ad-hoc approach of requiring a press and hold to trigger a text selection. Our probabilistic approach easily circumvents this issue by simulating both alternatives and waiting for future finger movements to disambiguate the interaction. Figure 10.6 illustrates this disambiguation. The algorithm uses a simple heuristic: the total amount of vertical or horizontal motion to adjust the likelihood of selecting text vs. scrolling. The most likely interface is presented on screen. When a user pauses, the alternate interpretation is shown on screen. Users can switch to this interpretation simply by tapping on it.

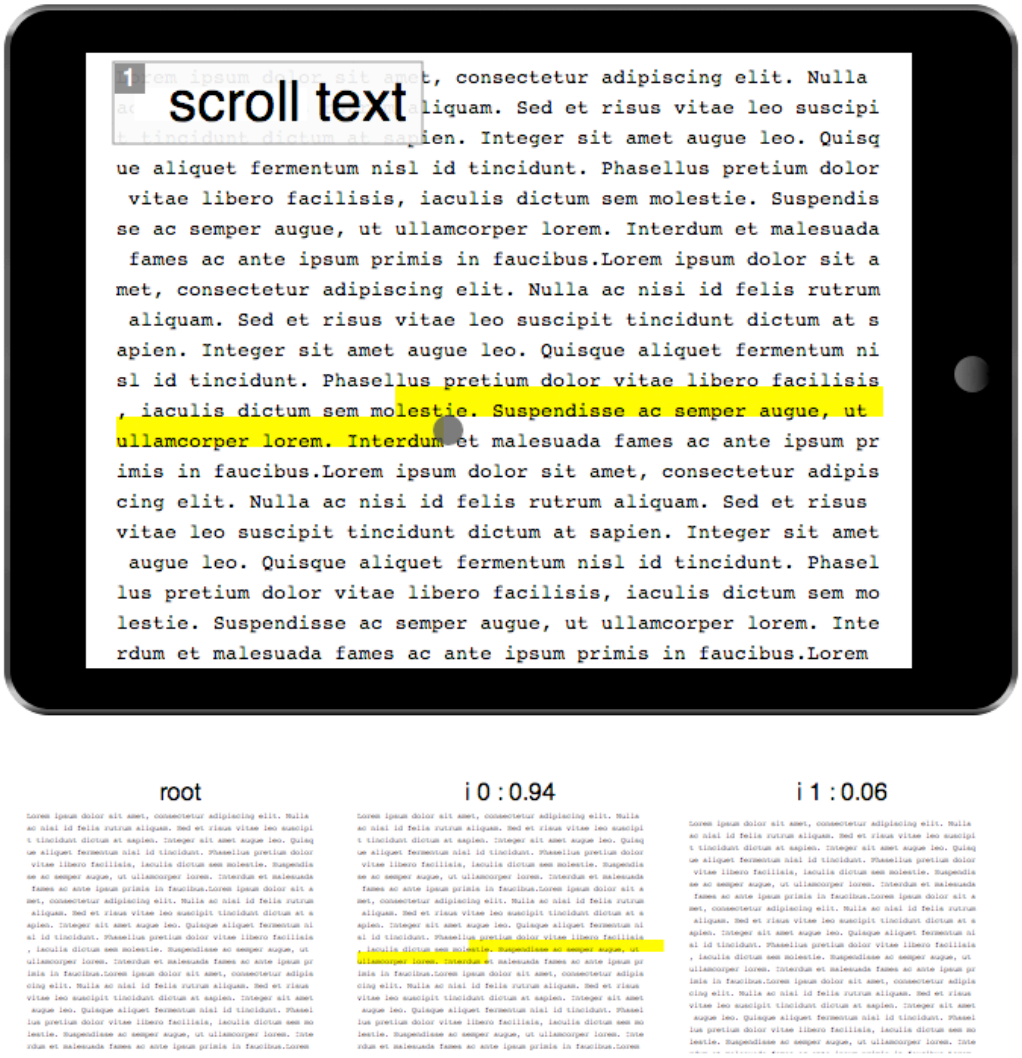


Figure 10.6 Demonstration of disambiguation between text selection and scrolling. Both alternatives are tracked with likelihoods (bottom row, center, bottom row, right). The most recent certain state is also tracked (bottom row, left). In this case the system believes the user is scrolling with likelihood 0.94. A user may switch interpretations by selecting the alternative in the top left corner.

This demo was implemented in JavaScript. 20 event samples are generated for every key event, and interfaces are downsampled to 10 alternatives at each step.

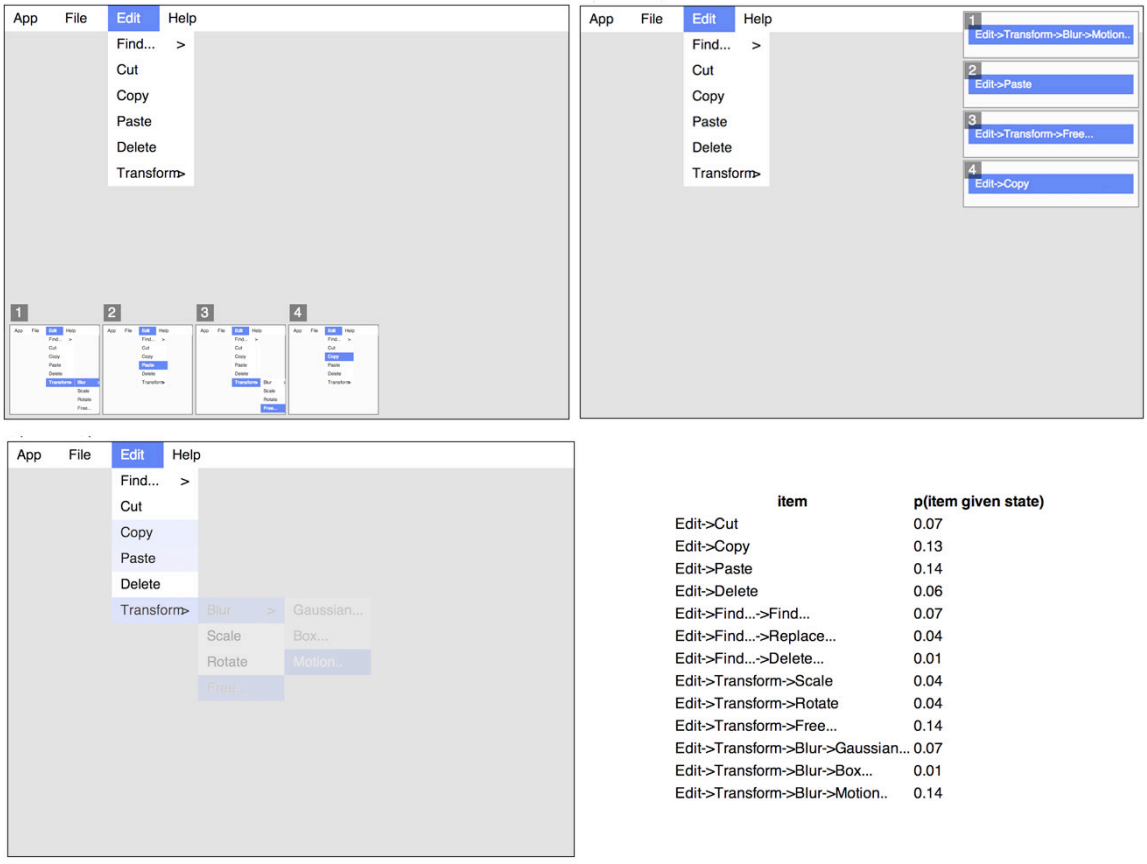
10.1.6 Predictive Menus

Menus in desktop applications can often be complicated and difficult to learn. Furthermore, with the exception of the 'recent items' list many applications support, along with the ill-fated adaptive menu feature provided in Microsoft Office 2003 (Microsoft 2003), menus rarely adapt to our behavior. In fact, it has been shown that changing the menu

items themselves to *e.g.* put more commonly used items up top is actually less efficient (Findlater 2004). Nevertheless, it would be nice if menus could take advantage of usage behavior to predict what users will select and accelerate menu selection. Additionally, for novice users, it would be beneficial to visualize commonly used menu items.

To solve both of these problems, I built a predictive menu system which supports both of these requirements without rearranging or modifying the underlying menu. Figure 10.7 shows an example. Menu items have different selection likelihoods given the currently selected menu item. These likelihoods are built using the Markov assumption, and based on user history. Given the likelihood of menu items, the predictive menu shows one of several developer or user-specified interfaces.

First, the menu can show the most likely interface, corresponding to a conventional menu system. Second, the menu can overlay possible future menu selections given current input (Figure 10.7), adjusting the opacity based on the likelihood of selecting a particular



item	p(item given state)
Edit->Cut	0.07
Edit->Copy	0.13
Edit->Paste	0.14
Edit->Delete	0.06
Edit->Find...->Find...	0.07
Edit->Find...->Replace...	0.04
Edit->Find...->Delete...	0.01
Edit->Transform->Scale	0.04
Edit->Transform->Rotate	0.04
Edit->Transform->Free...	0.14
Edit->Transform->Blur->Gaussian...	0.07
Edit->Transform->Blur->Box...	0.01
Edit->Transform->Blur->Motion..	0.14

Figure 10.7 Illustration of predicted menus. A user has moved his mouse over the Edit menu, the likelihood of each subsequent menu item selection is presented in the bottom right and is based on past behavior. Top Left: An n best list of the 4 most likely alternatives is displayed. Top Right: A more compressed n best list is displayed, this time showing a custom rendering of menu commands. Bottom Left: Every interface alternative is overlaid, with opacity reflecting likelihood.

item. Third, the menu can show an n-best list of possible menu item selections, ordered by likelihood. Thus, when the user selects Edit…, the most likely menu item, Gaussian Blur, is shown as the most likely completion. Users can then either select the completion or press the corresponding key to jump right to that option. Several renderings of n best lists are possible, as shown in Figure 10.7.

Although the capabilities of this demo are fairly complex, the implementation is no more complex than that of a normal hierarchical menu. In fact, more complexity is devoted to the implementation of the hierarchical menu itself than to rendering predictions. Menus respond to mouse events, behaving in the same manner as standard hierarchical menus. Menus also respond to special "select menu item" events, which specify a specific menu item to select. When a menu item is selected, the menu interactor determines a list of predicted future events (and likelihoods) and dispatches a new probabilistic event, which generates a distribution of events samples according to the prediction likelihoods. Each of these probabilistic events is then handled by the menu interactor, and interface alternatives representing possible future selections are generated. These alternatives are then displayed as interactive n-best lists or as overlaid feedback.

This example demonstrates several things. First, it shows how the JULIA toolkit supports prediction of future events. Second, it demonstrates how prediction of future events can accelerate interactions. More importantly, it demonstrates how complex interactions such as predicting future actions and showing completion can be added to interfaces with minimal developer effort. The JULIA toolkit handles all complexity regarding tracking interface alternatives, rendering feedback, and disambiguation.

10.1.7 Ninja Cursors

To further demonstrate how the probabilistic input architecture enables complex interaction techniques can be built using simple logic, we re-implemented a version of Ninja Cursors (Kobayashi 2008). Ninja cursors is an interaction technique that uses multiple cursors to reduce target selection time. Rather than manipulating a single cursor, users manipulate an array of cursors distributed evenly across the screen. The original technique used a clever set of heuristics to ensure that no more than a single cursor was present inside a target at a time. Figure 10.8 demonstrates the behavior of this interaction technique. Multiple evenly spaced cursors are generated as samples from a special Probabilistic Ninja Cursor event. To disambiguate multiple inputs, Kobayashi changed the behavior of cursors to ensure that no more than one cursor was inside a button at a time. Our approach is slightly different: Button press likelihood is proportional to the proximity of a cursor to the center of the button. When button press likelihoods are too close, the mediator defers action, and an n-best list is displayed to the user, allowing him to disambiguate (Figure 10.7).

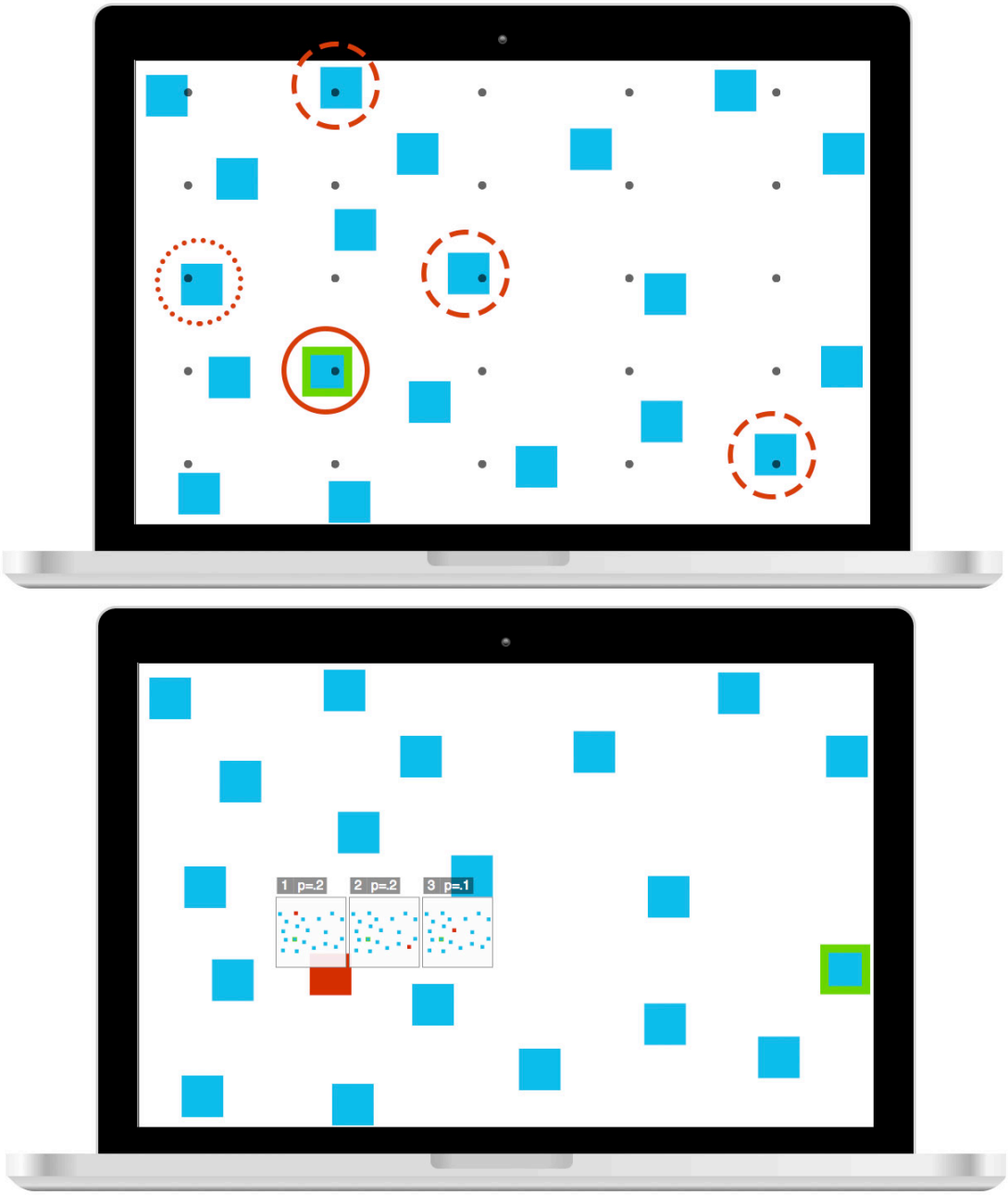


Figure 10.8 Demonstration of Ninja cursors. Rather than a single cursor, the user has a grid of cursors. The user wishes to select the target outlined in green (circle with solid line). Several other cursors overlap this target (dashed and dotted lines). After clicking, the toolkit selects the most likely interface, and shows the three most likely alternatives (dashed lines). The least likely alternative (dotted line) is not shown.

Note that this implementation allows for likelihoods to be adjusted by more than just proximity to button center. For example, eye gaze can be used to further adjust likelihoods. Other context such as previous actions may also be used.

Once again, while the resulting interaction is complex, the implementation of this interaction technique is fairly simple, requiring only a small modification to the default button interactor to add a likelihood to the transition of the mouse click event, and the creation of a new NinjaCursor mouse event. This demonstration is yet another example of how novel and fairly complex interaction techniques can be implemented using very little extra work. Once again, developers do not need to worry about the actual tracking of alternatives, rendering alternatives, or disambiguation. This is all handled by the toolkit.

This demonstration was implemented using the third iteration of the JULIA toolkit. For Ninja cursors, the number of event samples equaled the number of Ninja Cursors desired. In our case, this was 25 evenly spaced cursors. Interface samples were reduced to 10. In total, the demonstration was written in about 350 lines of JavaScript code.

10.1.8 N Best List for Gestures

We also re-implemented the disambiguation method for gestures mentioned in (Mankoff 2001) to further show the generality of our approach. Specifically, we built an n-best list for a gesture recognizer. Gesture recognizers can be notoriously inaccurate, and when these gestures perform actions that are difficult to back out of, this can be very frustrating. To avoid this, we built an n-best list for gesture recognizers which allows users to disambiguate their gesture if several gesture interpretations are likely. Figure 10.9 demonstrates the behavior.

We implemented a simple shape drawing application where users draw shapes *via* gesture (circle, triangle square). When a user draws a gesture, the most likely shape is drawn to the screen. If multiple interpretations are likely, alternate interfaces containing results

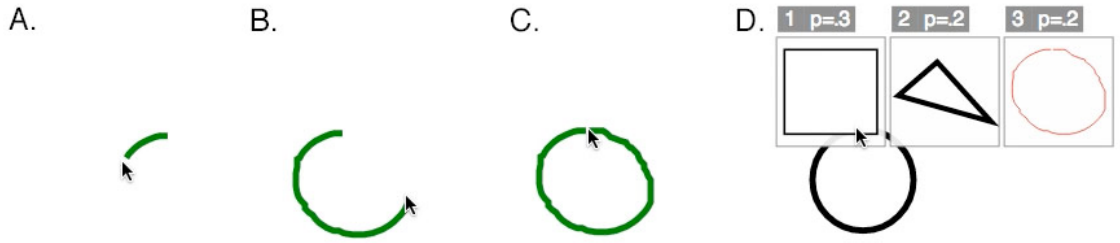


Figure 10.9 Example of N Best list for gestures. User executes a circle gesture (A-C), several interpretations are likely. The most likely interface is displayed by default, and possible alternatives are displayed as an n best list (D). The user may then select one of the alternate interpretations, or continue interacting using the most likely interpretation.

of other gestures (in our application, drawing different shapes) are displayed. The user may then switch to an alternate interpretation, or continue their interaction if the current shape is correct.

A JavaScript implementation of the $1 recognizer is used (Wobbrock 2007). When a gesture is recognized, a final action request is sent. The mediator used here executes the final action in an alternate interface and presents the result. This example was implemented with 20 samples per event and at most 10 interface samples, and demonstrates once again the versatility of the toolkit.

10.1.9 Bayesian Drawing: Leveraging Prior Probabilities to Adjust Future Action Likelihood

As a final demonstration of how prior actions can be used to adjust the likelihood of future actions, I discuss a behavior implemented in the flagship application (graphical object editor) in Section 10.3.2. Specifically, this application guesses the shape a user is drawing based on their motion and past actions

Users draw shapes or lines by dragging out the bounding box of the shape. Horizontal, vertical, and unconstrained lines may be drawn, in addition to rectangles and ellipses. If the user dwells in the middle of an interaction, an n-best list pops up, allowing the user to disambiguate. Likelihood of shapes and lines drawn depends on motion: horizontal motions reduce the likelihood of vertical lines, and vice versa. Additionally, a user's last action adjusts the likelihood for their next actions. If a user drew an ellipse last, an ellipse is more likely.

The implementation again for this interaction is fairly simple. Lines and shapes are separate interactors, they are placed in a container that dispatches to each child with equal likelihood (this is part of JULIA's standard library of interactors). The line recognizer adjusts its likelihood for horizontal/vertical lines according to line motion. Both the line and shape recognizers have probabilistic transitions to the down state, which examine the last action and adjust likelihood accordingly.

This demonstration was implemented in JavaScript using the third iteration of the toolkit. 20 samples per event were generated, leaving at most 10 alternatives at the end of each dispatch step. This demonstration shows how to incorporate priors into application logic. Once again, the framework does all of the heavy lifting.

10.2 Feedback Techniques

In addition to implementing several new interaction techniques, I have also implemented six feedback techniques: three existing, and three new, to demonstrate the versatility of the feedback system implemented.

10.2.1 Side Views

The N-Best list fusion technique discussed in Section 7.5.2 provides an implementation of a Side Views-like interaction. Side View is a technique described by Terry *et al.*(Terry & Mynatt 2002) for providing persistent, on-demand previews of commands. One improvement my approach provides over side views is the ability to communicate the likelihood of alternatives, as can be seen in Figure 10.10. In particular, the feedback techniques use opacity, scale and blur to communicate the likelihood of alternatives.

10.2.2 Octopocus

To demonstrate the versatility of the Overlay feedback method, we implemented an Octopocus-like feedback technique published by Bau *et al*. A similar interaction technique is also described in Bennet *et al.*'s paper on SimpleFlow (Bennett *et al*. 2011). For background, the Octopocus interaction technique aims to help users learn gestures by overlaying all possible gesture completions on the screen as a user is performing a gesture, so that users may better understand how to move their finger to complete a

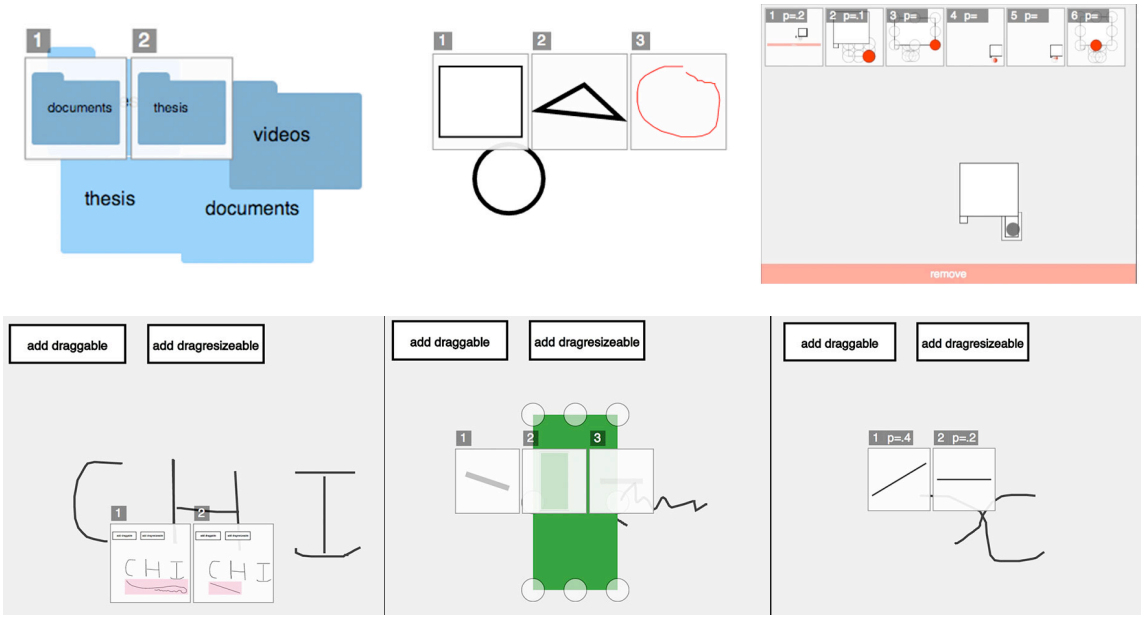


Figure 10.10 Examples of N-Best lists presented throughout this thesis, aggregated for clarity. N-Best lists provide a similar functionality as Side Views.

certain action. The state machine for the Octopocus interactor is presented in Figure 10.11. The likelihood of each gesture is adjusted based on the likelihood of each gesture at each point in time. This likelihood is in turn used to adjust the likelihood of transition probabilities for each of the possible gesture interpretations.

In addition to the reimplementation of the above existing techniques, several novel feedback techniques are presented in Chapter 9, including rendering possible future states, animating between alternatives, and having side views highlight only the changed interface states.

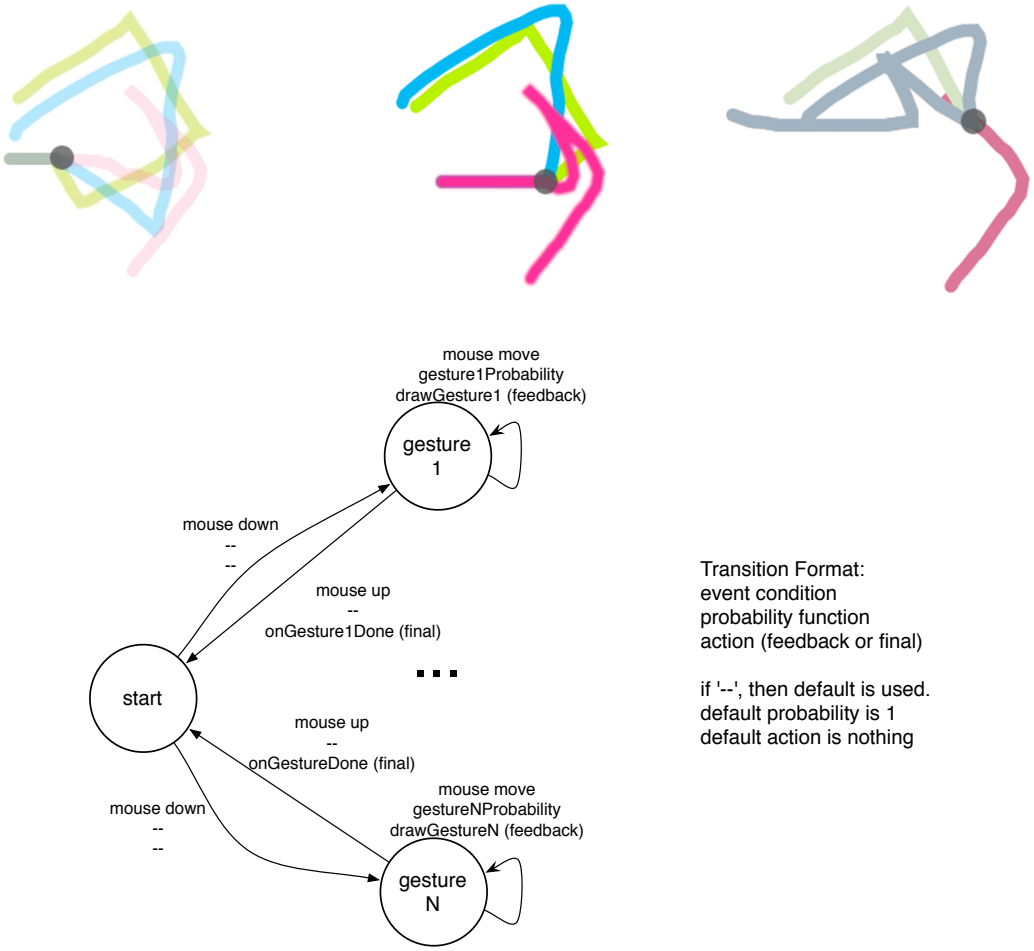


Figure 10.11 Example of Octopocus-inspired interaction technique of overlaying multiple alternate interpretations using opacity (left), blur (middle) and saturation (right) to reflect probability. Bottom: state machine for Octopocus-inspired interaction technique.

10.3 Applications

In addition to the interaction techniques presented above, I implemented three more complex applications using the probabilistic input architecture. These applications demonstrate the scalability of the approach in terms of interface complexity. Not only is the probabilistic architecture able to appropriately track interface alternatives and likelihoods, but the development burden on developers is similar to what would be required for conventional applications.

10.3.1 GesturePaint

GesturePaint was developed for the second iteration of the JULIA toolkit. This painting application demonstrates carrying of uncertainty from the sensor, to input interpretation, and finally action level. Additionally, the application delays action until it is certain of an interpretation. Below I briefly describe several of the interactors used in this application.

All of my example interactors were written in very much the same form as a conventional interface, without having to think or code probabilistically. The programmer doesn't have to be concerned with probabilities because the system does it for them in a nice, transparent way. The GesturePaint application itself is written entirely without regard to uncertain events, and is a relatively simple application consisting of roughly 400 lines of code with the same setup code and logic as any standard paint application.

My paint application allows users to "stamp" (add onto the canvas) images onto their painting. To support this I developed a stamp interactor that can be both moved and pressed. The stamp interactor uses three states to accomplish this. The state machine has two state properties – the drag start position and the touch ID (to support multitouch).

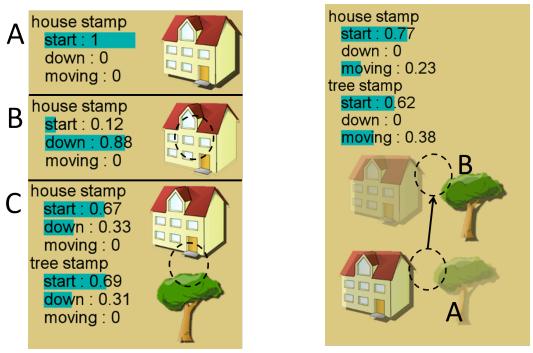


Figure 10.12: Screenshots of feedback provided by stamp interactors. The dotted circles have been added to indicate the position of the user's finger, and the numeric probability displays at the left are for illustration, but would not normally appear in such an interface. Left: Press feedback: A: Stamp not pressed. B: Stamp depressed completely when press is unambiguous. C: Stamp depressed partially when press is ambiguous. Right: Move feedback. A: User pressed in between the house and tree when beginning their drag, overlapping the tree more than the house. B: Both the house and tree are shown, with the tree being less transparent than the house reflecting overlap difference.

Stamps provide feedback to indicate the likelihood that they are being pressed vs. moved. Stamps provide press feedback by manipulating the shadow to make the interactor appear depressed in proportion to the likelihood that they are pressed (Figure 10.12, left). To provide feedback about moving, stamps show a 'ghost' version of the moved stamp. The transparency of the ghost stamp is based on the stamp's move likelihood (Figure 10.12, right). Sometimes multiple stamps might be selected or moved ambiguously (because the initial touch overlapped both interactors). To accommodate this, each stamp uses an alpha value corresponding to its move likelihood, which helps the user to see what the system thinks is happening (Figure 10.12, right).

This has the advantage that the user can back out of an incorrect interpretation before any final actions are invoked (by moving the stamp away from and then back to its original position). Importantly, I do not explicitly support this escape mechanism: it is a natural solution that arises from an understanding that dragging and pressing are both possible.

The specific feedback used by the stamp class is different for different transitions. However, all of them use likelihood as a drawing parameter (for shadow size, transparency, etc.). Outside of this parameter, the stamp has no other code that uses probabilities.

Stamps demonstrate the impact of uncertainty about which exact screen location the user intends to touch, what direction the user is moving, what interactor is being targeted, and so on. In these examples uncertainty arises directly from the properties of individual input events (at the sensor level). Another source of uncertainty is recognized input, as in the case of gesture recognition. Gestures that are prefixes of one another (such as 'c', 'g' and circle gestures) are especially problematic. Many applications are designed to avoid common prefixes or provide sophisticated feedback because of the resulting high degree of ambiguity.

My framework handles this sort of uncertainty without requiring any special effort by the developer. By default, the recognizer is called repeatedly after each new input event (*e.g.*, TOUCH_MOVE) arrives. Each time, it generates a probabilistic gesture event which contains a distribution specifying the probability that each possible gesture is the correct in-

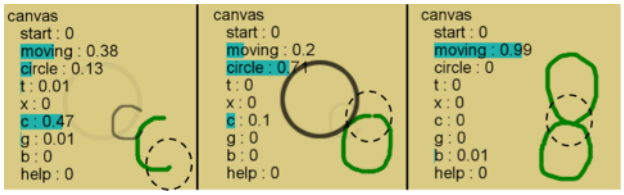


Figure 10.13: Screenshots of the canvas interactor as user draws a figure 8. The "c" and circle gesture share a common prefix. Dotted line indicates finger location. Left: Initially a 'c' gesture is most likely (probability 0.47). Feedback indicates that c is the most likely interpretation, though a circle is possible. Middle: The system becomes confident that the gesture is a circle, reflecting this in feedback (also evident in state distribution). Right: As the user completes the figure 8, the canvas believes the user is painting and removes all gesture feedback.

terpretation. During event sampling, this event is divided into event samples for individual gestures (*e.g.*, a circle gesture, a 'c' gesture, etc.), weighted by likelihood.

I developed a canvas interactor for my paint application that handles both 'moving' (for painting) and gesture events. Figure 10.14 shows the gestures the canvas recognizes. The canvas interactor has 9 states (one for each recognized gesture as well as a start and moving state), and 37 transitions. The canvas provides feedback about the canvas state (paint and interpreted gestures) using transparency. Figure 10.13 illustrates what happens when the user paints an 8 shape on the canvas. Because 8 shares a common prefix with 'c' and 'circle', the top hypothesis shifts from 'c' to 'circle' to painting on the canvas as the user draws. The canvas interactor demonstrates the toolkit's support for recognition ambiguity: this interactor supports recognition of gestures with common prefixes without any additional complex logic to handle ambiguity.

Developers working in a conventional input handling framework could certainly implement this canvas, however they would need to track not only the gesture probabilities, but also would need to include logic to determine when to decide whether the user is painting or gesturing. In my framework, the developer handles all of this simply by including both gestures and raw down/move/up events on transitions in the canvas's state machine. The underlying system handles all logic relating to tracking probabilities and deciding between paint and gesture events.

Each of these interactors is interesting individually, but the interactions become even more interesting when the interactors are combined in an application. In addition to ambiguity about touch location and gesture, for any user action it is always unclear whether a user intends to paint on the canvas, execute a gesture, click on a button, or move a stamp. The success of the paint application hinges on its ability to manage multiple alternative interpretations across multiple interactors for as long as possible (i.e., until the user lifts his/her finger, at which point the system needs to act). The framework I developed ensures that the application gives appropriate feedback about each possible action to the user. When the user lifts her finger, the framework acts appropriately (either resolving input when the resulting action is clear or prompting the user to disambiguate). No changes are required to the interactors described above for this to happen: Thanks to my framework, this complex application simply works.

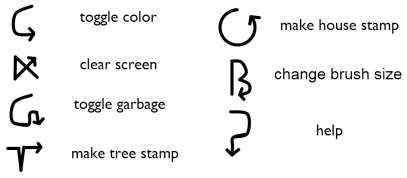


Figure 10.14: Gestures recognized by the Canvas Interactor

The power of my framework is clear when all of these interactors are combined together. While it might be feasible to write these interactors individually using a conventional input framework, writing an application that correctly handles uncertainty across interactors would be extremely difficult, and not reusable. For example, in a hand built solution, it would be difficult to remove one type of interactor (say, a stamp) and replace it with another, whereas my framework handles this *via* cancellation and acceptance of actions. The JULIA toolkit provides a general method for handling uncertain input, tracking interactive state and resolving ambiguous actions.

This implementation was built using the second iteration of the JULIA toolkit, and used about 20 event samples for every touch input event. The application totaled about 150 lines of application code (for interactor setup and event handling, this doesn't include the code to describe interactor behavior).

10.3.2 Graphical Object Editor

The graphical object editor is a drawing tool inspired by Igarashi's work on interactive beautification (Igarashi & Matsuoka 1997) and Zeleznik's Lineogrammer (Zeleznik *et al*. 2008). It also leverages concepts from Lunzer's subjunctive interfaces work (Lunzer & Hornbæk 2008). The Graphical Object Editor was developed for the third iteration of the JULIA toolkit. In addition to the features demonstrated by GesturePaint in section 10.3.1, this application also demonstrates incorporation of prior action into future behavior, proper maintenance of interface alternatives, more complex and nuanced interactions, generations of new events in response to existing events, and rendering of feedback to allow for disambiguation. Below I describe the application behavior and implementation.

The graphical object editor is a diagramming application that allows for drawing and manipulation of shapes and lines. Users may draw rectangles or ellipses using free form gestures (*e.g.* draw a circle to make a circle) or by dragging out a bounding box. They may also draw straight lines: unconstrained, horizontal or vertical. The application tracks a user's previous action and adjusts the likelihood of shapes accordingly such that the most recently drawn shape is most likely (all other things being equal).

Shapes and lines may be dragged and resized. When multiple actions are possible, the toolkit tracks all options and their likelihoods. A line's endpoints snap to control points on shapes. If multiple snap points are possible, all alternative shapes are tracked. Items can be removed by being dragged to the bottom of the screen. When being dragged, appropriate feedback shows up indicating the removed region. Of course, the user may also simply wish to place an item near the bottom of the screen. Once again, both alternatives are tracked. When a user pauses or dwells, an n-best list appears showing the most likely interpretations. Users can then disambiguate their intent by tapping on an alternative.

Many of the controls in this application are actually borrowed from the interaction techniques presented earlier. For example, drawing lines are the same as presented in section 10.1.9. Drawing shapes by freehand is done using the same gesture recognizer in section 10.1.8. Drawing shapes by specifying the bounding box is nearly identical to lines, the only differences being in the feedback rendered, the resulting event handlers, and in the removal of logic that considers horizontal and vertical lines.

The interactors for the ellipse and rectangle shapes are similar to the window interactor in section 10.1.3. Lines have similar logic. In addition to being resized and dragged, lines can also snap to different geometry. To do this, the line queries all targets. For every snappable point, a new action request is generated, updating the line endpoint to the snap position. After a dwell timeout, JULIA renders feedback, which is an n-best list as described in 9.5.3.

One interactor is custom for this application, however: the 'remove' button. The remove button only appears when a draggable item is selected. When a draggable item is dropped over the remove button, that item is removed from the canvas. This is implemented through the generation of new events in response to drag begin and drag end interactions. After drag interaction begins, any draggable object sends a new DragBegan event. This injecting is performed through special call in the JULIA toolkit. However, under the covers the dispatch logic for this call is almost identical to normal dispatch. The only difference is this event is not dispatched to all possible interfaces, just to the specific interface called from. The trash button then handles these drag began and drag end events. Note that this behavior does not happen on all interface alternatives, only the ones where an element is actually being dragged.

Once again, much of the complex behavior in this application: tracking of alternatives, disambiguation, is built into the toolkit. Many interactors here have already been implemented, the power comes from when everything is combined. The success of the graphical object editor hinges on its ability to manage multiple alternative interpretations for as long as possible, and to easily be able to disambiguate between alternatives. The probabilistic architecture ensures that the application gives appropriate feedback about each possible action to the user. When the user lifts her finger, the framework acts appropriately (either resolving input when the resulting action is clear or prompting the user to disambiguate). No changes are required to the interactors described above for this to happen.

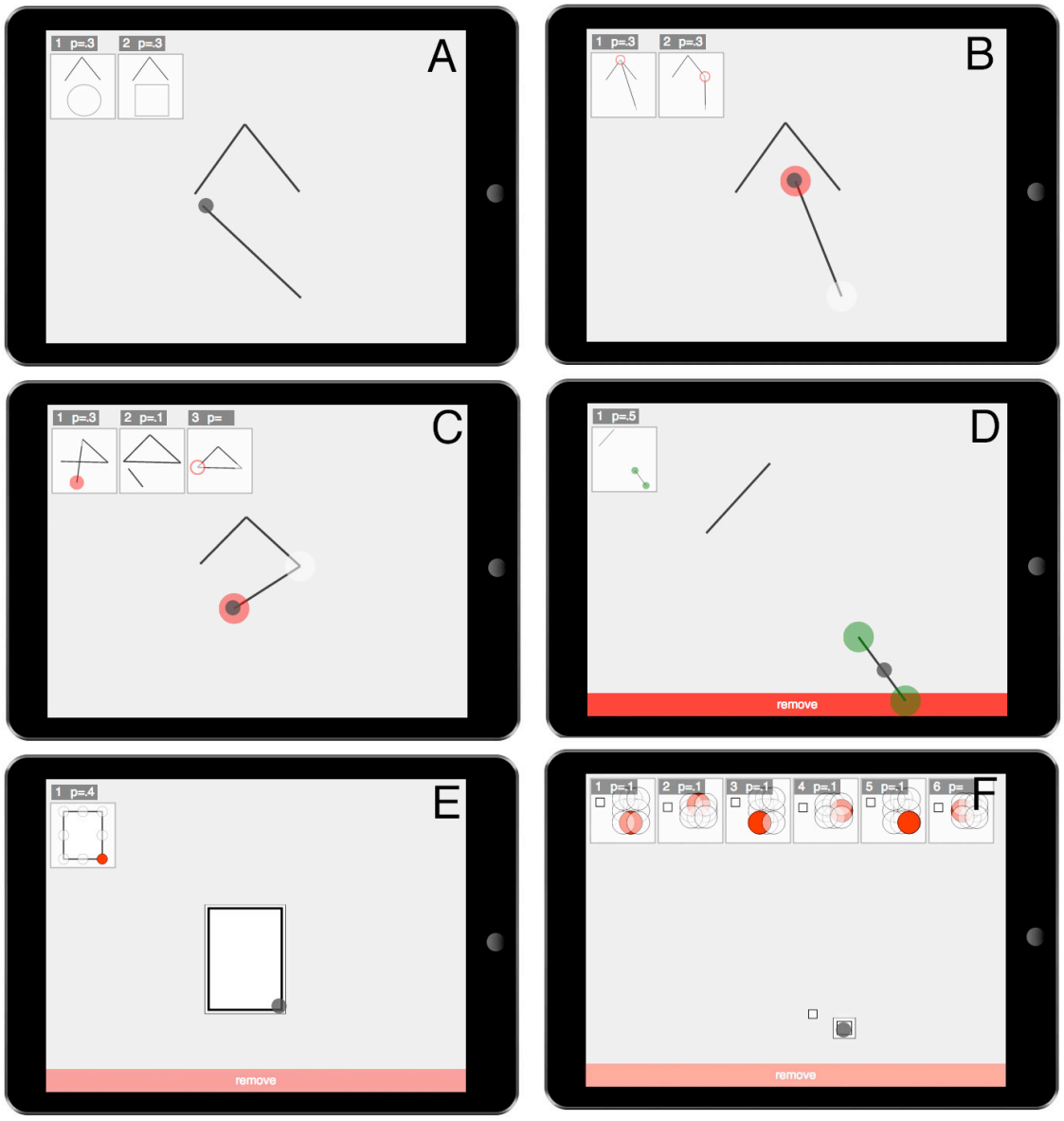


Figure 10.15: Illustration of a few features of the graphical object editor. Gray dot indicates touch location. A: When a user drags on an empty area, he may create either a line, rectangle, or ellipse. The most recent shape is weighted to be more likely. B: Line endpoints may be moved, they snap to different endpoints. Multiple endpoint candidates may be chosen (hollow circles in alternative views), or the user may continue free manipulation (solid circle). C: When the intended target of a drag is ambiguous (first alternative shows an alternate line, second alternative shows creating a new line, third shows a snap), alternatives are shown. D: Lines may be dragged to the bottom to be removed, or (as the alternative shows), a user may opt to just move the line to the bottom of the screen. E: Boxes may be moved or resized, the main interface shows drag feedback, alternative shows resize feedback. F: When a box is very small, all alternatives are shown, allowing the user to disambiguate.

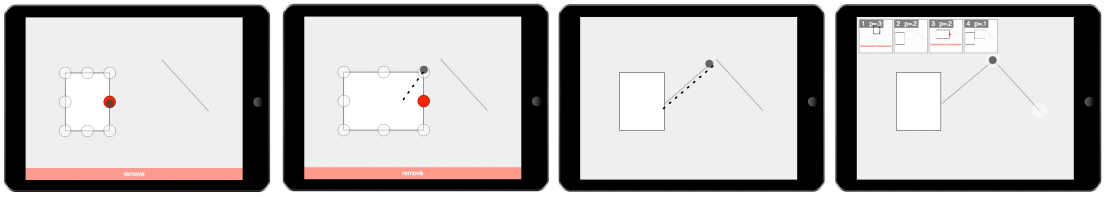


Figure 10.16: Example of user dragging a line from the edge of a box (dotted line shows drag). User begins drag, initially belief is box resize. Eventually, likelihood of box resize diminishes, line draw is next. After a pause, all alternatives are shown.

As an example, consider what happens when a user starts dragging on the edge of a box (Figure 10.16). Several interpretations are possible: dragging the rectangle, resizing the rectangle, drawing a line, drawing a rectangle, drawing an ellipse. All alternatives are considered, the most likely, a drag, is displayed. As the user continues dragging upward, the horizontal resize becomes less likely. The user pauses, and all alternatives are displayed on the screen. As the user continues, the updates dynamically update. Finally, the user selects the alternative for a line, and continues drawing, now with a line visible. When the user releases his finger, an OnLineCompleted handler is called, which adds a new line object to the screen. Note that this is the same interaction as the Window resizing example presented in section 10.1.3.

This type of interaction would be very difficult to implement in a conventional user interface framework. In fact, one would need to build something almost as complex as the probabilistic input system itself to properly handle all cases (such as, for example, the remove button appearing only when draggable items are selected). In contrast, this application was written largely without regard to probability, and its fairly complex operation was described in only a few hundred lines of JavaScript. As with other applications, this application could be run with just 20 samples per event, and up to 10 alternatives tracked at once. This application demonstrates the power of the probabilistic input architecture: it enables development of complex interfaces without requiring complex programming.

10.3.3 Backwards compatibility

The JULIA toolkit can be used to simulate traditional, deterministic input. This is merely a matter of disallowing nondeterministic state machines and only having one event sample. In this situation, only one interface alternatively is maintained and updated. In this representation, specification of the state machine, as well as dispatch, transforms into exactly the same algorithms that are used to dispatch conventional input.

10.4 Platforms

To demonstrate that the probabilistic input architecture is language and platform independent, we implemented different iterations of the architecture in three different platforms: a Windows desktop platform, the Windows Phone platform, and a web platform (Chrome, JavaScript + SVG). Additionally, the architectural concepts used were refined in every interation, so that the final iteration represents over three iterations of successive improvements.

The first iteration of the JULIA toolkit (Schwarz *et al*. 2010) was developed in C# to run on the Windows 7 platform. It was developed using the Windows Presentation Foundation (WPF) toolkit. The toolkit ran on a Dell XT2 touchscreen tablet with an Intel Core 2 Duo 1.2 GHz processors and 3GB of RAM. I chose a laptop mostly for convenience of development and the availability of touch input. Many interactions were touch interactions in a desktop environment, which is often a very challenging interface.

The second iteration of the JULIA toolkit (Schwarz *et al*. 2011) was built for the Windows Phone OS on top of the XNA game framework, which has a primitive input handling system supporting only polling for input but not events. I chose the phone because touch input is a widely used medium that contains a large amount of uncertainty (i.e., the intended location of the touch event is uncertain). The phone also illustrates that my approach works on systems with relatively low amounts of memory and processing power. The development and testing for this version of the tool was done on a phone with 512MB of RAM and a 1GHz processor, which compared to typical modern desktop or laptop machines is quite limited (and even current top of the line phones). I support touch, gesture, and accelerometer-based shake events. The framework has about 2,000 lines of C# code, and the 10 demos I wrote totaled about 1,000 lines of C# code.

The third iteration is implemented in JavaScript and the HTML5 APIs. The toolkit hooks into existing touch event handlers, overriding them with probabilistic event handlers and dispatching input *via* the JULIA toolkit. The JavaScript platform was chosen to illustrate the versatility of the toolkit and make the resulting toolkit and demos easier to share. The implementation supports mouse, keyboard, touch, and voice input. Finally, this implementation demonstrates that the probabilistic architecture can be implemented on a universally used platform: the web browser. The development and testing for the third iteration was done on a computer with 16 GB of RAM and a quad core, 2.3GHz processor. This toolkit supports touch, mouse, keyboard, and voice recognition events (from the Google Chrome Voice API). The toolkit and demos were developed and tested in the Chrome browser. The toolkit (including the library of interactors and feedback objects) totals to about 8,000 lines of uncompressed JavaScript code, and the 14 demos written total about 1,600 lines of JavaScript code. Collectively, these implementations demonstrate that the architecture is platform and language agnostic.

11 CONCLUSION

This dissertation has presented a new user interface architecture that treats user input as an uncertain process, approximates the probability distribution over possible interfaces using Monte Carlo sampling, and provides tools for interface developers to easily build probabilistic user interfaces. Importantly, alternate interfaces (and their likelihoods) are managed by the architecture itself, allowing for interface developers to reap the benefits of probabilistic interfaces without needing to think probabilistically. To recap, the contributions of this thesis are as follows:

- A new architecture for modeling and dispatching uncertain user input, using Monte Carlo sampling to approximate the probability distribution over possible interface states.

- A system for generating fluid interactive feedback which continuously communicates alternative (or future) input interpretations as a user is interacting and allows her to disambiguate intent.

- An API and library which allows user interface developers to easily build probabilistic interfaces. This includes a mechanism for specifying interactor behavior using probabilistic state machines, a rich set of classes for developing interactive feedback, and a built-in mechanism for adjusting likelihood of future actions based on past input.

- A large collection of interaction techniques and applications which demonstrate the versatility and power of the toolkit. These include demonstrations of how to leverage prediction to accelerate interaction, how to build interfaces that allow users to easily switch between multiple input interpretations, and how to perform Bayesian inference to adapt interface behavior.

This architecture provides the foundation for a new era of user interfaces which handle uncertainty gracefully to better infer user intent.

11.1 Limitations

Several limitations of the work are alluded to throughout the thesis, warranting a more careful discussion. These limitations fall into two categories. First, the current toolkit requires significant CPU and memory resources. Second, the usability of many of the techniques, along with the APIs used, has not been tested.

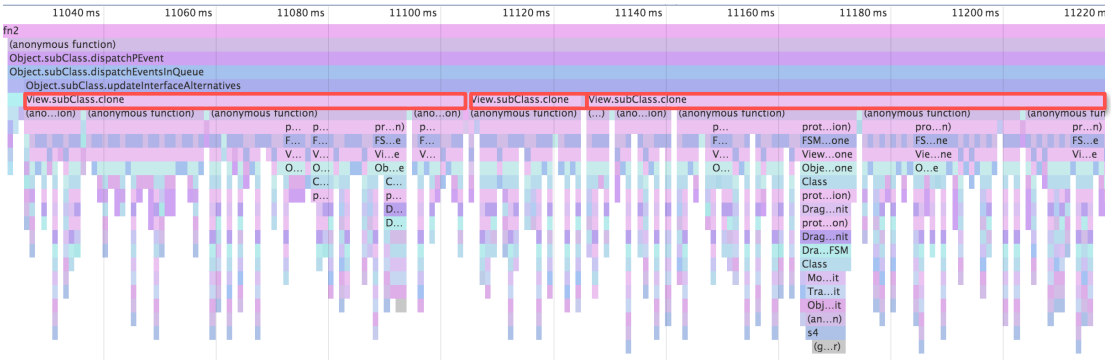


Figure 11.1: CPU profile of dispatch process obtained during dispatch of a single input event. Most time is spent in the View.subClass.clone method, cloning interface alternatives (rectangular boxes outlined in red).

One drawback of our approach is that large numbers of samples are often required to represent distributions where the state space is large: when input events have many variables, or variables that can have many values. In many cases, the state space for user interfaces is not that large, and thus tens of samples are often sufficient (as is demonstrated by our interaction techniques and applications). However, the current approach breaks down for cases with large amounts of ambiguity where hundreds or even thousands of samples are required.

This dissertation work focused on developing early concepts in using Monte Carlo techniques for managing uncertainty, and did not focus heavily on optimizations to allow for simulation of large numbers of samples. However, many optimizations are possible. A CPU profile of toolkit in operation (Figure 11.1) reveals that most time is spent cloning interfaces. Several optimizations are possible here. First, each dispatch sequence for a (event sample, interface sample) pair is independent of others, meaning we can execute update operations in parallel. Additionally, rather than copying all interface properties, we can use incremental data structures (Tanin *et al*. 1996) to store interface alternatives. This will not only reduce the time required to clone alternatives but will also reduce the memory footprint. Many other optimizations are possible and left for future work.

A second limitation of this work is that the interaction techniques and applications presented in this dissertation were not evaluated for usability. While our validation by demonstration is suitable for the contribution of this work, the practicality of the demonstrations presented remains unknown. Many interaction techniques (such as animating between alternate versions of interfaces), are interesting but questionable in terms of usability. Further investigation is needed into the usability of the interaction techniques presented.

Additionally, the developer-facing API (probabilistic state machines) was not rigorously validated. While an informal validation based on number of interaction techniques and code simplicity is presented, a more formal evaluation is needed before the approach can be determined as ready for practical use. These evaluations will likely point out problems that will lead to improvements, which will in turn be their own valuable contributions.

Despite these limitations, the work presented in this thesis is a great leap forward in the domain of architectures for handling uncertain input, and the demonstrations provided show the feasibility of the Monte Carlo approach—a surprising and positive result.

11.2 Future Work

The probabilistic input architecture lays the foundation for an entirely new type of user interface architecture, however this is just the tip of the iceberg. Most of the work needed to bring this architecture into practice lies before us. Several avenues of future work present themselves: improving performance of the toolkit, developing more complex interfaces, and building more sophisticated features.

In addition to the performance improvements mentioned in the previous section, the interfaces in the demos and applications, while complex, still do not approach the complexity of most real applications. A further avenue of future work may be to track multiple alternative program states, to remove the need to decide on all final actions. Finally, the probabilistic input architecture lends itself nicely to input prediction and Bayesian simulation. A more general method that allows for systematic integration and interface improvement would be interesting to explore. In short, there's an enormous body of work to explore, and the future of probabilistic input is a promising field that I hope to see flourish in the near future.

12 REFERENCES

53Designs, 2012. Paper. Available at: http://www.fiftythree.com/paper.

Abowd, G.D., 1992. *Formal aspects of human-computer interaction*. Oxford University.

Abowd, G.D. & Dix, A.J., 1994. Integrating status and event phenomena in formal specifications of interactive systems. *ACM SIGSOFT Software Engineering Notes*, 19(5), pp.44–52.

Arvo, J. & Novins, K., 2000. Fluid Sketches : Continuous Recognition and Morphing of Simple Hand-Drawn Shapes. *UIST '00*, 2, pp.73–80.

Balakrishnan, R., 2004. "Beating" Fitts' law: virtual enhancements for pointing facilitation. *International Journal of Human-Computer Studies*, 61(6), pp.857–874.

Bau, O., 2010. *Interaction streams*. University of Paris.

Bau, O. & Mackay, W.E., 2008. OctoPocus: a dynamic guide for learning gesture-based command sets. In *UIST '08*. New York, NY, USA: ACM, pp. 37–46.

Bellegarda, J. et al., 2012. Method, device, and graphical user interface providing word recommendations for text input.

Bennett, M. et al., 2011. Simpleflow: enhancing gestural interaction with gesture prediction, abbreviation and autocompletion. In *INTERACT'11*. London, UK: Springer-Verlag, pp. 591–608.

Bier, E.A. et al., 1993. Toolglass and magic lenses. In *SIGGRAPH '93*. New York, New York, USA: ACM Press, pp. 73–80.

Bohus, D. & Horvitz, E., 2009. Learning to Predict Engagement with a Spoken Dialog System in Open-World Settings. In *SIGDIAL '09*. New York, NY, USA: ACM Press, pp. 244–252.

Bolt, R.R.A., 1980. "Put-that-there": Voice and gesture at the graphics interface. In *SIGGRAPH '80*. New York, NY, USA: ACM Press, pp. 262–270.

Bourguet, M.-L., 2006. Towards a taxonomy of error-handling strategies in recognition-based multi-modal human–computer interfaces. *Signal Processing*, 86(12), pp.3625–3643.

Bragdon, A. et al., 2009. GestureBar: improving the approachability of gesture-based interfaces. In *CHI '09*. New York, NY, USA: ACM, pp. 2269–2278.

Cohen, P.R. et al., 1997. QuickSet: multimodal interaction for simulation set-up and control. In *ANLC '97*. Morristown, NJ, USA: Association for Computational Linguistics, pp. 20–24.

Eisenstein, J. & Davis, R., 2005. *Gestural Cues for Sentence Segmentation*, Boston, MA.

Eisenstein, J. & Davis, R., 2006. Gesture improves coreference resolution. In *NAACL '06*. Stroudsburg, PA: Association for Computational Linguistics, pp. 37–40.

Eisenstein, J. & Davis, R., 2004. Visual and linguistic information in gesture classification. In *ICMI '04*. New York, New York, USA: ACM Press, pp. 113–120.

Frankish, C., Hull, R. & Morgan, P., 1995. Recognition accuracy and user acceptance of pen interfaces. In *CHI '95*. New York, NY, USA: ACM Press, pp. 503–510.

Freeman, D. et al., 2009. ShadowGuides : Visualizations for In-Situ Learning of Multi-Touch and Whole-Hand Gestures. In *ITS '09*. New York, NY, USA: ACM Press, pp. 165–172.

Goodman, N. et al., 2012. Church: a language for generative models. *Uncertainty and Artificial Intelligence*, pp.220–229.

Google, 2013. Google Glass. Available at: http://www.google.com/glass/start/.

Gordon, N.J.J., Salmond, D.J.J. & Smith, A.F.M., 1993. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEEE proceedings. Part F. Radar and signal processing*, 140(2), pp.107–113.

Grossman, T. et al., 2006. Hover Widgets : Using the Tracking State to Extend the Capabilities of Pen-Operated Devices. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*. New York, NY, USA: ACM Press, pp. 861–870.

Hammersley, J.M. & Handscomb, D.C., 1964. Monte Carlo Methods S. Kotz & N. L. Johnson, eds. *Annals of Statistics*, 44(247), pp.335–341.

Harrison, C. & Hudson, S.E., 2009. Abracadabra: wireless, high-precision, and unpowered finger input for very small mobile devices. In *UIST '09*. New York, New York, USA: ACM Press, pp. 121–124.

Harrison, C., Schwarz, J. & Hudson, S.E., 2011. TapSense: Enhancing Finger Interaction on Touch Surfaces. In *UIST '11*. New York, New York, USA: ACM Press, pp. 627–636.

Henze, N., Rukzio, E. & Boll, S., 2011. 100,000,000 taps: analysis and improvement of touch performance in the large. In *MobileHCI '11*. New York, New York, USA: ACM Press, p. 133.

Hinckley, K. et al., 2010. Pen + Touch = New Tools. In *UIST '10*. New York, New York, USA: ACM Press, pp. 27–36.

Hinckley, K. et al., 2012. Touch and stylus discrimination and rejection for contact sensitive computing devices.

Hofstadter, D.R., 1979. *Godel, Escher, Bach: An Eternal Golden Braid*, New York, NY, USA: Basic Books, Inc.

Holz, C. & Baudisch, P., 2010. The generalized perceived input point model and how to double touch accuracy by extracting fingerprints. In *CHI '10*. New York, NY, USA: ACM Press, pp. 581–590.

Holz, C. & Baudisch, P., 2011. Understanding touch. In *CHI '11*. New York, New York, USA: ACM Press, pp. 2501–2510.

Horvitz, E., 1999. Principles of mixed-initiative user interfaces. In *CHI '99*. New York, New York, USA: ACM Press, pp. 159–166.

Horvitz, E. et al., 1998. The lumière project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann Publishers Inc., pp. 256–265.

Huckauf, A. et al., 2005. What you don't look at is what you get. In *Proceedings of the 2nd symposium on Appied perception in graphics and visualization*. New York, New York, USA: ACM Press, pp. 170–170.

Igarashi, T. et al., 1998. Pegasus. In *CHI '98*. New York, New York, USA: ACM Press, pp. 24–25.

Igarashi, T. & Hughes, J.F., 2001. A suggestive interface for 3D drawing. In *Proceedings of the 14th annual ACM symposium on User interface software and technology - UIST '01*. New York, New York, USA: ACM Press, pp. 173–181.

Igarashi, T. & Matsuoka, S., 1997. Interactive beautification: a technique for rapid geometric design. In *UIST '97*. pp. 105–114.

Istance, H. et al., 2008. Snap clutch, a moded approach to solving the Midas touch problem. In *Proceedings of the 2008 symposium on Eye tracking research applications ETRA 08*. pp. 221–228.

Jacob, R., Deligiannidis, L. & Morrison, S., 1999. A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1), pp.1–46.

Kaiser, E. et al., 2003. Mutual disambiguation of 3D multimodal interaction in augmented and virtual reality. In *Proceedings of the 5th international conference on Multimodal interfaces - ICMI '03*. New York, New York, USA: ACM Press, pp. 12–19.

Kettebekov, S., Yeasin, M. & Sharma, R., 2005. Prosody based audiovisual coanalysis for coverbal gesture recognition. *IEEE Transactions on Multimedia*, 7(2), pp.234–242.

Lalanne, D. et al., 2009. Fusion engines for multimodal input: a survey. In *International Conference on Multimodal Interfaces*. New York, New York, USA: ACM Press, pp. 153–160.

Latoschik, M.E., 2005. A user interface framework for multimodal VR interactions. In *International Conference on Multimodal Interfaces*. New York, New York, USA: ACM Press, pp. 76–83.

Li, Y., 2009. Beyond Pinch and Flick : Enriching Mobile Gesture Interaction. *Computer*, 42(12), pp.87–89.

Liu, J., Wong, C.K. & Hui, K.K., 2003. An adaptive user interface based on personalized learning. *IEEE Intelligent Systems*, 18(2), pp.52–57.

Lü, H., 2013. Gesture Studio : Authoring Multi-Touch Interactions through Demonstration and Declaration. In *CHI '13*. ACM Press, pp. 257–266.

Lunzer, A. & Hornbæk, K., 2008. Subjunctive interfaces. *ACM Transactions on Computer-Human Interaction*, 14(4), pp.1–44.

MacEachren, A.M., 1992. Visualizing uncertain information. *Cartographic Perspective*, 13(13), pp.10–19.

Mankoff, J., 2001. *An Architecture and Interaction Techniques for Handling Ambiguity in Recognition-Based Input*. Georgia Institue of Technology.

Mankoff, J. et al., 2000. OOPS: a toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces. *Computers & Graphics*, 24(6), pp.819–834.

Martin, J.-C., Veldman, R. & Béroule, D., 1998. Developing Multimodal Interfaces: A Theoretical Framework and Guided Propagation Networks. In *Multimodal Human-Computer Communication, Systems, Techniques, and Experiments*. London, UK: Springer-Verlag, pp. 158–187.

McCulloch, W.S. & Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), pp.115–133.

Metropolis, N. & Ulam, S., 1949. The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247), pp.335–341.

Moscovich, T., 2009. Contact area interaction with sliding widgets. In *UIST '09*. New York, New York, USA: ACM Press, p. 13.

Murugappan, S. et al., 2012. Extended Multitouch : Recovering Touch Posture , Handedness , and User Identity using a Depth Camera. In *UIST '12*. New York, NY, USA: ACM Press, pp. 487–496.

Newman, W.M., 1968. A system for interactive graphical programming. In *AFIPS '68*. New York, New York, USA: ACM Press, pp. 47–54.

Nielsen, J., 1993. *Usability Engineering*, Morgan Kaufmann.

Olwal, A. & Feiner, S., 2003. SenseShapes : Using Statistical Geometry for Object Selection in a Multimodal Augmented Reality System. In *ISMAR '03*. Washington, DC, USA: IEEE Computer Society, pp. 300–301.

Oviatt, S., 1996. Multimodal interfaces for dynamic interactive maps. In *CHI '96*. New York, New York, USA: ACM Press, pp. 95–102.

Oviatt, S., 1999a. Mutual disambiguation of recognition errors in a multimodal architecture. In *CHI '99*. New York, New York, USA: ACM Press, pp. 576–583.

Oviatt, S., 1999b. Ten myths of multimodal interaction. *Communications of the ACM*, 42(11), pp.74–81.

Oviatt, S.L. & Cohen, P.R., 1999. Multimodal integration-a statistical view. *IEEE Transactions on Multimedia*, 1(4), pp.334–341.

Pang, A.T., Wittenbrink, C.M. & Lodha, S.K., 1997. Approaches to uncertainty visualization. *The Visual Computer*, 13(8), pp.370–390.

Rabiner, L.R., 1989. A tutorial on hidden Markov models and selected applications in speech recognition A. Waibel & K.-F. Lee, eds. *Proceedings of the IEEE*, 77(2), pp.257–286.

Rogers, S. et al., 2010. FingerCloud: uncertainty and autonomy handover in capacitive sensing. In *Imagine*. Association for Computing Machinery, pp. 577–580.

Rosenberg, I. & Perlin, K., 2009. The UnMousePad: an interpolating multi-touch force-sensing input pad. *ACM Transactions on Graphics*, 28(3), p.65.

Schwarz, J. et al., 2010. A framework for robust and flexible handling of inputs with uncertainty. In *UIST '10*. ACM Press, pp. 47–56.

Schwarz, J., Mankoff, J. & Hudson, S.E., 2011. Monte carlo methods for managing interactive state, action and feedback under uncertainty. In *UIST '11*. pp. 235–244.

Sewell, W. & Komogortsev, O., 2010. Real-time eye gaze tracking with an unmodified commodity webcam employing a neural network. In *CHI EA '10*. New York, New York, USA: ACM Press, pp. 3739–3744.

Stewart, C. & Murray-smith, R., 2011. AnglePose : Robust , Precise Capacitive Touch Tracking via 3D Orientation Estimation. In *CHI '11*. New York, NY, USA: ACM Press, pp. 2575–2584.

Tanin, E., Beigel, R. & Shneiderman, B., 1996. Incremental data structures and algorithms for dynamic query interfaces. *ACM SIGMOD Record*, 25(4), pp.21–24.

Terry, M. et al., 2004. Variation in element and action: supporting simultaneous development of alternative solutions. In *CHI '04*. New York, NY, USA: ACM Press, pp. 711–718.

Terry, M. & Mynatt, E.D., 2002. Side views: persistent, on-demand previews for open-ended tasks. In *UIST '02*. New York, NY, USA: ACM, pp. 71–80.

Vanacken, D. et al., 2008. Ghosts in the interface: Meta-user interface visualizations as guides for multi-touch interaction. In *2008 3rd IEEE International Workshop on Horizontal Interactive Human Computer Systems*. IEEE, pp. 81–84.

Vogel, D. & Baudisch, P., 2007. Shift: a technique for operating pen-based interfaces using touch. In *CHI '07*. New York, NY, USA: ACM Press, pp. 657–666.

Wasserman, A.I., 1985. Extending State Transition Diagrams for the Specification of Human-Computer Interaction. *IEEE Transactions on Software Engineering*, SE-11(8), pp.699–713.

Weir, D. et al., 2012. A user-specific machine learning approach for improving touch accuracy on mobile devices. In *UIST '12*. New York, NY, USA: ACM Press, pp. 465–476.

Weir, D., 2012. Machine learning models for uncertain interaction. In *UIST Adjunct Proceedings '12*. New York, New York, USA: ACM Press, pp. 31–34.

Welch, G. & Bishop, G., 2006. An Introduction to the Kalman Filter. *In Practice*, 7(1), pp.1–16.

Wensveen, S.A.G., Djajadiningrat, J.P. & Overbeeke, C.J., 2004. Interaction Frogger : a Design Framework to couple action and function through feedback and feedforward. In *Proceedings of the 5th conference on Designing interactive systems DIS '04*. New York, NY, USA: ACM Press, pp. 177–184.

Wigdor, D. et al., 2009. Ripples: utilizing per-contact visualizations to improve user interaction with touch displays. In *UIST '09*. ACM, pp. 3–12.

Williamson, J., 2006. *Continuous Uncertain Interaction*. University of Glasgow.

Williamson, J., 2003. Hex: Dynamics and probabilistic text entry. In *Switching and Learning in Feedback Systems*. Berlin, Heidelberg: Springer-Verlag, pp. 333–342.

Williamson, J. & Murray-Smith, R., 2002. *Audio feedback for gesture recognition*, Department of Computing Science, University of Glasgow.

Wilson, A. & Shafer, S., 2003. XWand: UI for intelligent spaces. In *CHI '03*. New York, NY, USA: ACM, pp. 545–552.

Wobbrock, J.O., Wilson, A.D. & Li, Y., 2007. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *UIST '07*. New York, NY, USA: ACM, pp. 159–168.

Yeh, S.-T. & Chen, J.-M., 2011. Method and device for palm rejection.

Zeleznik, R. et al., 2010. Hands-on math. In *UIST '10*. New York, NY, USA: ACM Press, pp. 17–26.

Zeleznik, R.C. et al., 2008. Lineogrammer: creating diagrams by drawing. In *UIST '08*. New York, New York, USA: ACM Press, pp. 161–170.

Ziv, J. & Lempel, A., 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3), pp.337–343.