

Monte Carlo Methods for Managing Interactive State, Action and Feedback Under Uncertainty

ABSTRACT

Current input handling systems provide effective techniques for modeling, tracking, interpreting, and acting on user input. However, new interaction technologies violate the standard assumption that input is *certain*. Touch-based interaction, speech recognition, pen-based systems, free space gestures, and sensors for context often produce uncertain estimates of user inputs. Although current systems tend to throw away uncertainty early on, there is evidence that information available only in the user interface and application can help to ensure that uncertainty is resolved appropriately for the end user. This paper presents a set of techniques for tracking the state of interactive objects in the presence of uncertain inputs.

These techniques use Monte Carlo sampling to maintain a probabilistically accurate description of the user interface that can be used to make informed choices about actions. Samples are used to model the distribution of possible inputs, of possible interactor states that result from inputs, and of possible actions (callbacks and feedback) that result from interactor states. Each sample represents a single, completely certain description of the user interface. Our framework retains all the advantages of maintaining information about uncertainty as long as possible while minimizing the need for the developer to work in probabilistic terms. Because each sample is certain, the developer can specify most of the behavior of interactors in a familiar, non-probabilistic fashion. We present a working implementation of our framework and illustrate the power of these techniques within a paint program that includes three different kinds of uncertain input.

ACM Classification: H.5.2 [Information interfaces and presentation]: User Interfaces - Graphical user interfaces.

General terms: Human Factors

Keywords: Probabilistic Modeling, Uncertain Input, Dialog Specification, Finite State Machines.

INTRODUCTION

Advances in input methods such as touch, pen gestures, in-air interaction and speech recognition have begun to turn the promise of natural user interfaces into reality. Unfortunately, these input modalities are inherently

uncertain. Most interface toolkits assume that input occurred *exactly as reported*. As a result, any uncertainty in input must be resolved quickly and often simplistically. For example, traditional input systems often treat touch input as a single point (the centroid of the touch region) instead of a touch area. Because the center of the touch area may not be over the user's intended target, small differences in position can lead to big errors and a stunted user experience.

Handling uncertain input is a multifaceted problem that includes mechanisms for modeling uncertain input, deciding which interactive element(s) an input gets dispatched to, interpreting input with respect to interactive state, and mediating between alternative interpretations to decide which actions to execute and/or feedback to show. Past work has focused on mechanisms for modeling and dispatching uncertain input [18] and resolving uncertainty [11]. In this paper, we focus on how to interpret uncertain input with respect to interactor state and on giving appropriate feedback to accurately reflect this uncertainty.

Because user input takes place over the course of a sequence of events, interactors must be able maintain internal state across the delivery of events. State machines are a convenient mechanism for modeling state that many interactors use implicitly, if not explicitly.

In the work presented here, we show how to automate the process of tracking uncertain state for each interactor, while requiring developers to only provide a deterministic state machine description. A developer also specifies when an interactor should provide feedback or invoke callbacks, but must encapsulate these actions in *action requests* rather than executing them directly. These small changes allow the framework to automatically track the probabilities of multiple alternative interactive states as inputs arrive, and correctly update state as decisions are made about alternative actions, something no prior work supports.

To concretely illustrate our contribution, consider the following scenario: a user executes a circle gesture, but starts the gesture over a moveable icon (Figure 1). When an icon is dragged, it typically enters the **moving** state after an initial press-move event sequence, and then updates its position (which can be modeled as a *state variable*) after each move event. In the application shown in Figure 1, the specification of the moveable icon is a three state machine that operates like the icon described above. However, the

Submitted to
UIST 2011

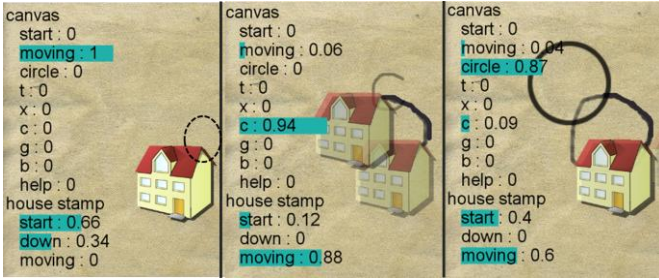


Figure 1: Screenshots from application built using our system. **(left)** User starts a circle gesture over a moveable icon (dotted circle indicates touch area). The system believes the user is dragging the house. **(middle)** User continues the circle gesture. The system indicates a possible drag and a possible C gesture as equally likely. **(right)** User completes circle gesture. The system recognizes the circle as the most likely interpretation. Note: The state probability information at the left is for illustration only and not normally part of the interface.

framework tracks this state without making any irreversible changes because it is *also* tracking the possibility that the user is executing a gesture that would be delivered to the canvas behind the house. The framework accurately tracks the likelihood that the icon is in the **moving** state, and the likelihood that user is making a circle. It updates these probabilities over time as the most likely hypothesis changes. For example, as the motion begins to resemble a circle more and more, the likelihood that the house is in the **moving** state should go down, and the feedback the user sees should reflect this.

Our solution to this problem is to use Monte Carlo methods [13]. Given a probability distribution (such as the location of a touch event), Monte Carlo methods approximate that distribution as a set of samples, each representing one possible value (such as an x,y location) from the distribution. Operations on each individual sample are deterministic, but in aggregate they can approximate how those operations would affect the entire distribution. In our work, samples are used to approximate three separate probability distributions: possible input events, possible interactor states, and possible actions. Computations (such as updating an interactor’s state) are done on samples. Because of this, they can be deterministic (can ignore the presence of uncertainty). Computations on samples produce new samples (state changes and/or action requests) which represent a new probabilistic distribution. Only where specialized feedback of ambiguity is desired or a decision between alternative actions must be made, does the presence of uncertainty need to be considered.

The state machine for the icon in Figure 1 is shown in Figure 2. Initially, the likelihood that the icon is in the **moving** state is high (meaning that many samples are in the **moving** state), however, as the user continues, the likelihood that the icon is in the **moving** state decreases and the likelihood that the user is making a circle gesture increases to 0.87. Our tool automatically tracks these probabilities for the developer, allowing her to focus on other aspects of interaction.

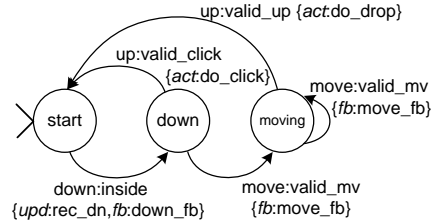


Figure 2: The state machine used by the house stamp. Each transition is annotated with **type:predicate {optional parameters}**. Type is the type of event the transition operates on, the predicate checks for any additional requirements.

Optional parameters include *upd* (an update method for updating state variables), *fb* (a feedback method for displaying feedback to the user), and *act* (an action method).

Past tools for managing uncertainty fall into two categories. One set of tools has sophisticated support for merging input from multiple uncertain modalities. Bourget developed a tool for visually specifying how multimodal input should be interpreted using state machines [2] while xml-based languages have been used to specify the relationship between incoming event triggers and outgoing actions [5]. In each case, the specification focuses on the flow of interaction through an overall dialog with the user, as opposed to the details of how each individual interactor operates (*e.g.*, [4]). By combining modalities, it has been shown in multiple domains that it is possible to gain new evidence for disambiguating the interpretation of uncertain input (*e.g.*, text input [7], speech [15], and multimodal input [17]). This research typically assumes that all of the uncertainty will be resolved before any input reaches a specific interactor (or the application proper). As a result, the uncertainty does not integrate smoothly with our well evolved and familiar GUI interaction techniques.

Another approach (the one taken here) attempts to manage uncertain input within a general model which is based on mainstream approaches for input handling. In prior work we have explored several of the sub problems of modeling and dispatching uncertain events (input modeling, dispatch, state tracking, action, feedback and mediation) [18]. Mankoff *et al.* outline basic aspects of modeling input and provide a mechanism for dispatching and mediating ambiguous input [10, 11]. However, little work has explored managing interactive state. Hudson and Newell address this problem in [8], outlining a theoretical approach for tracking interactive state given uncertain input. However, this approach is not fully integrated with other aspects of input handling and has not been implemented. Moreover, the approach discussed in [8] restricts computational power to strict regular languages and thus may present difficulties in practical use. This paper builds directly on ideas in [8, 10, 11, 18].

In the next section we present an overview of our framework, focusing on the main contributions in this paper: our approach to tracking and managing uncertain state within interactors, without requiring interactor developers to think probabilistically. The framework we’ve designed uses the concepts introduced in [18] as a starting

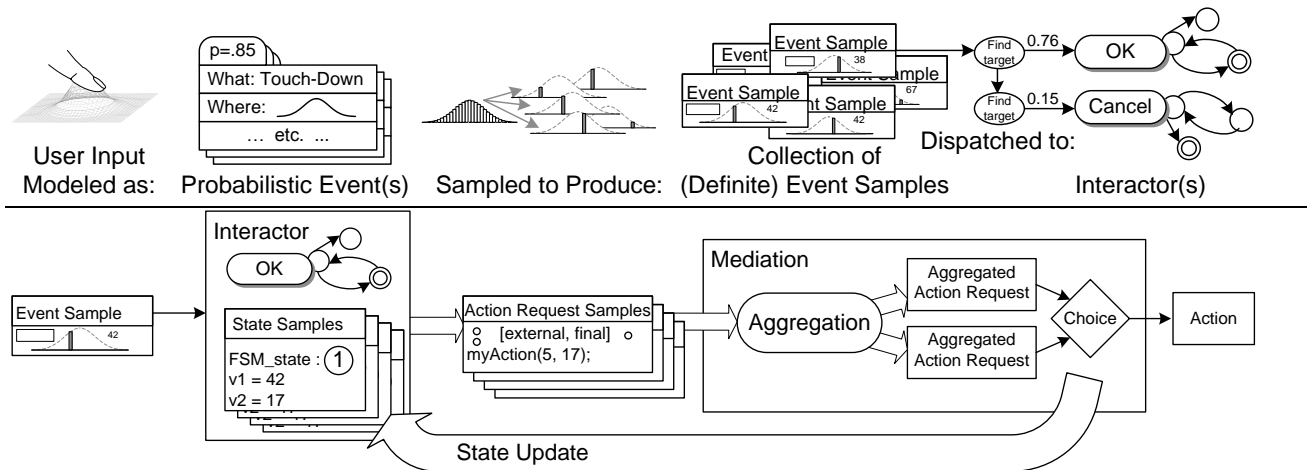


Figure 3: **(top)** A probabilistic event is sampled to produce a collection of (definite) event samples, which are dispatched to interactors such as an “ok” and “cancel” button. **(bottom)** Each interactor may be in more than one state (“state samples”). Event/state sample combinations are evaluated to produce action request samples. The system selects one final action in the end by aggregating similar samples together and selecting among them. This choice is then reflected back to the interactors by updating the state to be logically consistent with the action that is chosen.

point and expands on issues concerning state maintenance. As validation we then briefly describe our implementation and set of sample interactors we built using probabilistic state machines. Our examples illustrate the power of our framework, which brings us one step closer to developing an end-to-end system for handling uncertain input.

FRAMEWORK OVERVIEW

As illustrated in Figure 3 the overall framework we operate in has six components, each of which includes multiple steps. (1) First is *modeling* uncertain input (which produces many event samples). (2) As shown at top of Figure 3, these samples are *dispatched* to interactors. (3) Next, the framework *tracks* the internal state of interactors (Figure 3, bottom left). (4) This produces *action/feedback* requests. (5) *Mediation* aggregates and selects among these requests (bottom right). (6) Finally, a *state update* process ensures that interactor state correctly reflects the fact that certain actions were selected over others.

Modeling

As with conventional input, uncertain input is modeled using events. However, these events may be probabilistic in nature, and may contain probabilistic properties. In the example shown in Figure 1, the user’s input is modeled as two alternative events (a gesture and a touch event). The gesture event is represented as a distribution over all possible gestures (circle, t, x, c, g, b, and help) and the location of the touch event is represented as a distribution over possible x,y positions (in this case derived from the touch centroid using a 2d Gaussian function).

Dispatch

Given a representation of uncertain input, the framework needs to deliver, or *dispatch*, that input to interested interactors. Details about user input such as its type and location are traditionally used during dispatch to make decisions about which interactors should receive each event. Since this information may be uncertain, our dispatch

process must be probabilistic in nature. To reflect uncertainty about which interactor will consume an event sample, the system splits each event sample into several based on the number of interactors that want to consume it. Each sample is weighted based on the probability that it is consumed. In Figure 1, the user’s input is modeled using a number of samples (representing possible locations and gestures). Each of these samples is split in two (one delivered to the canvas and one to the house stamp).

Maintaining State

As event samples are delivered, each interactor needs to maintain its own interactive state. Since the interactor’s state is based on multiple uncertain prior inputs, it is probabilistic. The distribution across possible interactor states is represented as a collection of samples. Each *state sample* includes a current state (such as “start” or “moving”) and values for any variables associated with the state machine. Operation of *probabilistic state machine* based on these samples is a central focus of this paper and is described in detail in later sections.

Action Request

Each state machine transition may have an associated action. When an input event sample causes a transition with an associated action, an *action request sample* is generated to request that some action be taken. Action request samples are conceptually related to command objects [14] and contain information about the action, such as a method to carry it out as well as its likelihood. For example, when the user lifts his finger, the canvas interactor will transition to a state indicating a completed circle gesture. This in turn causes an action request sample to be generated that will create a new stamp if invoked. Because interactors are uncertain about their actual state, many possible action samples may be generated for each uncertain input event that is dispatched. This collection of action samples represents a probability distribution across possible actions the system might take.

Actions may have permanent consequences and/or effect future interactors, or may only produce feedback and not affect future interactions. We call these two forms of actions *final* and *feedback actions* respectively.

Mediation and Action

Mediation is a decision process that determines which action request samples (if any) to invoke (accept) and which to reject. In addition to a probability estimate for each action sample, the system maintains a notion of which requests are compatible with each other, and which are mutually exclusive, to aid in mediation. Mediation processes are considered in detail in [11] and we borrow those approaches as a starting point for the work here.

Our default mediator permits all feedback requests. For example, in Figure 1, the user’s stroke is shown, as is a circle and a semi-transparent indication of the house’s potential new location. When the user lifts her finger, multiple final action samples are created. The mediator selects the most probable action sample, which creates a new stamp.

State Update

When a final action request is accepted, this implies that the system has decided to act based on one interpretation of the inputs. Other interpretations may not be compatible with this action. The process of *state update* removes any state samples which are incompatible with an accepted action, leaving each interactor’s state machine in a less ambiguous state. For example, when the circle gesture is completed and acted upon, all state samples representing other possible gestures will be removed.

To summarize, our framework takes a probabilistic event, generates *event samples*, and dispatches each sample to various interactors which might receive the input. These interactors track their state using *state samples* and generate potential *action samples*. The mediation system decides which (if any) *action samples* to execute, and interactive state is updated accordingly. Collectively, these samples represent a distribution across input events, interactive state, and action. However, individually these samples can be treated by developers as deterministic since they each encapsulate a set of specific values.

MONTE CARLO METHODS FOR MANAGING STATE AND ACTION UNDER UNCERTAINTY

In this section we consider details for each component of the framework described above. We show how our framework manages sampled input, state, and actions to accurately track uncertainty throughout the input handling process.

Modeling Events

As with conventional input, probabilistic input is modeled using event records. Roughly following the approach in [8], a distribution across alternative events is indicated by assigning a weight to each event alternative, reflecting the probability that this alternative is the true input. Extending [8] (and following [18]), however, we also allow properties

of each event alternative to be uncertain. These properties are modeled using a probability mass function (PMF).

By default, separate probabilistic events are assumed not to be compatible, meaning they are part of a distribution of possible alternative interpretations of the user’s input. For example a “move” and a “gesture” resulting from the same underlying touch would be incompatible with each other (only one interpretation is correct). However, certain events represent parallel but independent (compatible) actions by the user. An example is simultaneous touches by two different fingers in a multi-touch system.

Dispatching Probabilistic Input

Figure 3 illustrates our system’s sample-based dispatch process. Instead of dispatching an uncertain event directly as in [18] (and thus forcing interactor developers to directly handle uncertain events), our dispatch method samples the probabilistic event. Each *event sample* is weighted according to the estimated probability that this sample represents a correct interpretation of the user’s actual intent. The dispatch mechanism then dispatches each (definite) event sample to each appropriate interactor in an interface. The events are dispatched in a dispatch order defined by the normal interactor structure (*e.g.*, by performing hit tests, using the current input focus, “bubbling” events, etc.).

To reflect the uncertainty about which interactor should consume each event sample, we break each event sample into multiple samples representing a distribution across the interactors they might be delivered to as follows: when each event sample is dispatched to an interactor, *i*, it is split into two new samples. One split sample represents the possibility that the original sample should be dispatched to *i* (weighted with probability p_i). The other split sample represents the possibility that the original sample should not be dispatched to *i* (weighted with probability $(1 - p_i)$). The value of p_i , represents the probability that the event sample is consumed by the interactor *i*. This *consumption probability* is the sum of the probabilities of all action requests made by *i*. The second split sample is dispatched to the next interactor in the dispatch order and split again using the same algorithm. This continues until all interactors have seen the sample, or the sample probability reaches zero.

State Machine Description

State machines are a convenient mechanism for modeling interactive state. Prior tools have introduced augmentations to make state machines more practical and easy to use (*e.g.*, [9, 16, 19]). In our system, interactor developers specify interactor behavior with a simple XML description. This description encodes an augmented form of a familiar *deterministic finite state machine*. We have added a small set of state variables and corresponding transition predicates. These variables can be used along with event properties in the transition predicates to help decide whether a transition should be taken. For instance, in a multi touch interface, the current pointer ID can be modeled as a state variable. A transition predicate can then ensure

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<dasm>
  <states>
    <state name="start">
      resetFunction="resetVariables" >
      <transition to="down">
        eventType="down"
        predicateFunction="insideRegion"
        updateFunction="recordDown"
        feedbackFunction = "doDownFeedback" />
    </state>
    <state name="down">
      <transition to="start">
        eventType = "up"
        predicateFunction="isClick"
        actionFunction="doClicked" />
      <transition to="moving">
        eventType="move"
        predicateFunction="validMove"
        feedbackFunction="doFeedbackMove" />
    </state>
    <state name="moving">
      <transition to="moving">
        eventType="move"
        predicateFunction="validMove"
        feedbackFunction="doFeedbackMove" />
      <transition to="start">
        eventType="up"
        predicateFunction="validUp"
        actionFunction="doDropped" />
    </state>
  </states>
  <startstate name="start" />
  <variables type="BlockStateInformation" />
</dasm>

```

Figure 4: XML description of stamp state machine.

that an interaction starts and ends with the same pointer ID by comparing the current event sample's touch ID to the touch ID stored in the pointer ID state variable at the start of the interaction.

By definition, every state machine has a start state. Our framework extends this notion by allowing multiple states to be designated as *stable* states (always including the start state). Stable states may represent the start of a new interaction. For example, a conventional check box might have two stable states, one for its checked state and one for its unchecked state. This allows our framework to roll the state machine back to the last sensible (stable) state when a possible interaction is rejected. For each stable state, the developer can optionally provide code for initializing or resetting the variables of the machine to correspond to that state.

Figure 2 illustrates a sample state machine in a graphical notation. As is customary, each node represents a state, while directed edges between nodes represent transitions. Edges (transitions) are annotated with the notation **type:predicate {optional parameters}** to describe the type of input to transition on, a predicate function which must be satisfied to take the transition, an optional update function which updates state variables when a transition is taken, and optional feedback or actions functions to execute when taking the transition.

Feedback and action functions are encapsulated as feedback and final action request samples to be executed if/when the

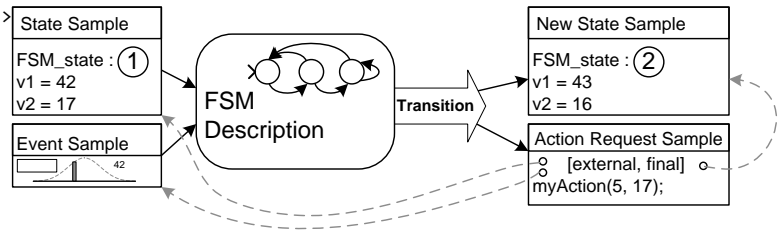


Figure 5: Our framework takes a state machine description, a state sample, and an event sample, and executes the appropriate transition of the state machine. The result is a new state sample and (in some cases) a final and/or feedback action request sample. All of this is handled in a deterministic fashion: It is the presence of multiple state/event samples that allows us to track state probabilistically.

requests get accepted. An action request sample encapsulates a method to carry out the action, along with a likelihood score, links to the event and state samples that led to that action being created. This gives the action's method access to state variables and event properties that may be needed to carry out the action.

Neither predicate nor state update functions may have side effects in the application. The update function may modify state variables only. Figure 4 illustrates the corresponding XML description developers would need to write to provide full details for the machine shown in Figure 2. Note that this state machine specification is deterministic: It could be used unchanged in a standard non-probabilistic input handling framework.

State Machine Operation

When an interactor receives an input event during the dispatch process, it must correctly update its internal state and produce actions based on this input. In a deterministic setting, as each event arrived, an interactor's state would be updated to a new (deterministic) state based on the type of the input event. In the case of our augmented state machine, the predicate function would also need to return true for a transition to be taken. Additionally, all of the methods associated with the selected transition (feedback, state variable update, and action) would be executed.

In the case of probabilistic input, interactor state is a probability distribution over multiple possible states rather than a single current state. Our framework uses a collection of weighted *state samples* to track this state distribution. Each *state sample* has a current machine state and a collection of values for the state variables. Together, these characterize a (deterministic) sample "state" of our state machine. When a (deterministic) event sample arrives, the state sample contains sufficient information to transition to a new state. As illustrated in Figure 5, this process produces a new state sample containing the new machine state and a new set of state variable values (as produced by the developer specified *updateFunction* for the transition). If the developer has specified a *feedbackFunction* or an *actionFunction* for the transition, one or more *action samples* are produced. Each action sample is assigned a

probability (weight) defined as $p_{es} * p_{ss}$, where p_{es} is the probability associated with the event sample (a combination of occurrence and dispatch likelihood) and p_{ss} is the probability that the state machine is actually in the state indicated by the state sample.

Each *event sample* is combined with every *state sample* in a given interactor to determine its probabilistic transitions. The set of new *state samples* produced after all event samples are processed represents the new distribution of interactor state. Action samples are accumulated across all transitions resulting from samples of a given probabilistic event. The set of action samples produced by each interactor represents the distribution across potential actions that interactor could invoke (as implied by the transition from the old state distribution to the new state distribution).

Mediation – Choosing Between Alternate Actions

Once dispatch of all the samples generated from a set of compatible probabilistic events is complete, we will have produced a set of one or more competing action samples. The *mediation* process decides which action samples to accept and invoke.

As mentioned in the overview, there are two types of action samples: *feedback actions* and *final actions*. Feedback actions are assumed to be transient and reversible. They provide information to the user, but do not change the future behavior of the system. An example is showing visually how a gesture is being interpreted. Final actions are assumed to have irreversible consequences which may change external objects in the application or the way the system acts in the future. An example is adding or removing an application component as a result of a completed gesture.

Since feedback actions are reversible and do not change the future behavior of the system, they are by default assumed to be compatible with each other (meaning that more than one of them can safely be invoked together). This has the benefit of indicating to the end user that ambiguity is present, as in Figure 1 where a circle and a faded second house are both visible. In the case of the faded house, the feedback action has translated its likelihood (weight) into an alpha value for the feedback image being drawn.

Final actions on the other hand would normally be incompatible with other actions. However, they may remain compatible if they represent interpretations of the user input and resulting state which do not conflict with each other. An example of this is input coming from two different users, two different devices, or even just two different fingers on a multitouch device that are acting independently. In that case the actions do not conflict and may be executed together. To handle this more general case, action requests each have an *isCompatible()* method which by default returns true only if the action requests come from different interactors and the probabilistic events which cause the action are compatible (as indicated by their *isCompatible()* methods). Alternately, the developer may

override this method to handle other more complex situations.

Before mediation can begin, *action aggregation* must take place. The event sampling, dispatch, and state tracking processes will often produce a number of action samples that invoke the same feedback or final action method. These samples, however, may vary in the sampled values found within the particular state and event samples they depend on. For example, two actions associated with the same gesture may contain different samplings of the user's touch location. However, these action samples might logically be considered equivalent. Our framework will aggregate such samples into one aggregated action request.

Although this can be overridden by the developer, our current implementation of action aggregation provides a default aggregation strategy that combines equivalent actions. Equivalent actions are actions which both encapsulate the same method (meaning they execute the same operation if invoked), and are associated with transitions within the same interactor's state machine. The combined action is assigned a weight that is the sum of all of the individual actions it is based on. More sophisticated custom strategies might, for example, only consider actions to be similar enough if certain state variables are within a limited variation of each other or might even perform a simple clustering of action samples based on state variable values.

The result of action aggregation is a collection of aggregated action request samples (each of which links back to the full set of action samples they aggregate over so that no information is lost). The next step in mediation is to form sets of incompatible action requests. These are requests that cannot logically be executed together because they each represent a decision to commit to different interpretations of input and resulting state that are in conflict with each other.

Since these aggregated actions cannot all be executed, the mediation system must decide to *accept* zero or more aggregated actions in each set of incompatible action requests. All remaining actions in the set are either *rejected*, or if the system cannot yet make a final decision *deferred*. When an action is deferred, the mediator typically seeks more information from the user via a dialog or other display.

Although algorithms for deciding between actions are not the main focus of this work (and of course many options exist [10]), we will briefly describe the default mediation algorithm used in our examples. Each incompatibility group is handled separately as follows: First, the algorithm rejects requests below some developer-specified minimum probability. Next, if there are only feedback requests left, all feedback actions are accepted. Otherwise, the algorithm selects a threshold equal to the most probable final action in the current incompatible group. It then considers all final actions requests with likelihood within a developer-

specified delta of that threshold. If there is one final action within delta of the threshold, that action is accepted and all other actions are rejected. If there are multiple final actions within delta of the threshold, the interaction is considered “too close to call”. In this case, the mediator rejects all actions below the delta, defers the final actions within the delta, and rejects any feedback actions within the delta. For deferral it invokes a developer specified mediation method. The default version of this method displays an “N-best list” choice dialog to the user (a common approach as described in [10]). Alternatively, the developer can specify a different method. Once the user designates a choice, the other possibilities are rejected and the chosen action is accepted.

Carrying Out Actions

The result of mediation is a list of action requests to be accepted and a list to be rejected. All accepted final action requests are invoked (executed as indicated in the action request). At present, feedback actions are treated the same way. As shown in Figure 1, small changes in how an interactor is drawn (transparency, shadow, *etc.*) or simple indicators (such as the current most likely gesture) are easily distinguishable and work well even in a complex application. However, in future work we envision a feedback merger step which might use negotiation techniques related to those developed for fluid documents [3] to combine more sophisticated feedback of ambiguity in flexible ways. This would allow feedback to scale up to even more complex situations.

When an accepted feedback or final action request is invoked, it is passed a copy of the aggregated action request object as a parameter. This aggregated request contains links to all of the original action requests it was aggregated from, which in turn link to the set of event and state samples (including state variables and event properties) causing the transitions which led to those actions. The system provides an extensible library of *value aggregators* such as *average*, *median*, and *most-probable*, which allow the action method to quickly establish definite values for all the values it needs to carry out its function.

State Update and Resampling

Once actions have been executed, we need to update interactive state to reflect the consequences that this implies. Acceptance and execution of a set of feedback actions does not require any state adjustment. However, acceptance and execution of a final action represents a decision to commit to a particular interpretation of input, associated with a particular state machine state. As a result, we need to adjust the state distributions to reflect this decision. In particular, for each interactor executing one or more accepted final actions, the mediator ensures that its state is consistent with those actions. Since each action arises from a transition, the target states of those transitions are consistent with the interpretation of the user’s input that has been selected. All state samples not matching one of the target states of the aggregated accepted actions are deleted.

For interactors which have rejected actions but no accepted actions it is necessary to reset the interactor state to one that would be consistent with rejecting all current interpretations. This is done by *resetting* each state sample in the machine – specifically by restoring it to the last stable state it passed through during its specific execution history (each sample maintains a record of this state). Restoring sets the state sample state number to the stable state and then executes a reset method to establish consistent machine variable values. Restoring the state samples makes the state distribution compatible with where the state distribution would have been if the rejected input sequence had never happened.

The final step that must be taken is *resampling* to reduce the total sample count. If left unchecked, over time the number of state samples will grow exponentially. This increase occurs each time multiple event samples arrive and are combined with each transitioning state sample. However, state distributions can generally be adequately approximated by a limited number of samples. We resample and renormalize the state distribution based on a developer specified limit of total samples.

In our current implementation (on a memory and CPU limited mobile phone) we enforce a fairly small sample budget. This requires down sampling, which we handle using an aggregation process. To return to our sample budget, we find state samples with the same machine state and aggregate the variable values within those samples to produce a single sample that replaces two or more originals. In systems allowing a more generous sample budget an alternative (and preferable) approach would be to employ importance sampling as is commonly used with particle filters [6, 12] (which have the same state expansion issues). Once resampling is complete we renormalize the weights of the states in each interactor’s state machine so that they again sum to 1.0.

IMPLEMENTATION

We implemented a proof of concept system that uses the sampling techniques described above to accurately track interactive state. Our system is 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. We support touch, gesture, and accelerometer-based shake events. Our development and testing has been done on a phone with 512MB of RAM and a 1GHz processor, which compared to typical desktop or laptop machines is quite limited. Due to these limitations, our example application implementation employed a very limited sample budget: using at most 50 samples per event and resampling to 10 state samples per interactor. These values can of course be changed by the developer depending on the interface/application. In practice we ran into no problems using these low sample numbers because of the small state machine size that most interactors have. Fortunately, even for unusual cases needing very large state

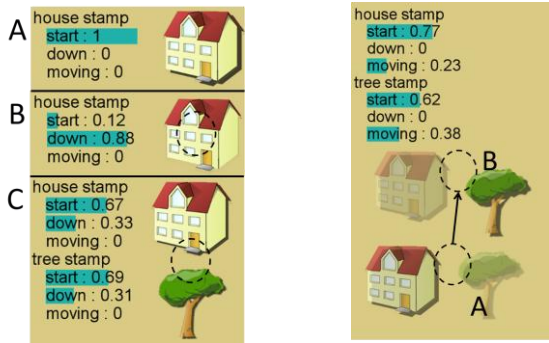


Figure 6: Screenshots of feedback provided by stamp interactors. The dotted circles have been added to indicate the position of the user’s finger. **(left)** Press feedback: A: Stamp unpressed. B: Stamp depressed completely when press is unambiguous. C: Stamp depressed partially when press is ambiguous. **(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.

machines, our approach is highly parallelizable, offering plenty of room for future optimizations.

Once details such as the number of samples are specified, developers rarely have to think probabilistically when they are developing interactors and applications that use these interactors. Aside from feedback methods that make use of likelihood, neither the interactor code nor the application code we developed involves explicit consideration of multiple possible interpretations of the user’s input.

EXAMPLE USES

To test the effectiveness of our approach and demonstrate its applicability to a wide range of problems, we developed six interactors of varying complexity. We then combined them in a gesture-based paint application. Please refer to the accompanying video figure to see the interactors and application in action.

Below we briefly highlight the implementation of several interactors we built to show what developers should expect when writing interactors. Although developers have access to probabilistic data (aggregated action requests and their associated events and state samples), they rarely need to concern themselves with this information. All of our example interactors were written largely without regard to probabilistic events (an exception is feedback designed to indicate the probability of a specific potential interpretation of the user’s input). The paint application itself is written entirely without regard to uncertain events, and is a relatively simple application consisting of the same setup code and logic as any standard paint application.

Stamps: Moveable Buttons

Our paint application allows users to stamp images onto their painting. To support this we developed an interactor which can be both moved and pressed. The stamp interactor uses three states to accomplish this. Figure 2 illustrates the

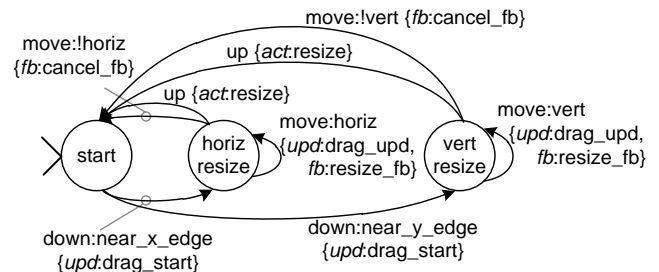


Figure 7: State machine for resizable interactor. state machine description used for stamps and Figure 4 gives the corresponding XML description. The state machine has two state variables – the drag start position and the touch ID (to support multitouch). Stamps provide feedback to indicate the likelihood 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 6, 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 6, 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 6, right).

The specific *feedbackFunction* 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.

Resizable Box: Context-Dependent Interaction

Our system allows developers to create resizable interactors that use direction of motion to differentiate between a resize action and other input such as a gesture or moving another interactor which may be underneath. To facilitate such interactions in previous work [18] required manually tracking the probability in each state. Our framework simplifies this drastically. The developer simply writes two predicates – one to check whether the current event is horizontally related to the original touch down (stored as a state variable), and the other to check whether the current event is vertically related to the original touch down. Once these predicates are associated with the appropriate transitions (from the middle and right state in Figure 7, respectively), the interactor will behave properly. As with stamps, feedback depicts probability using transparency in a ‘ghost’ view of the resize result.

Canvas: Handling Paint and Gesture Simultaneously

Stamps and resizable boxes 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

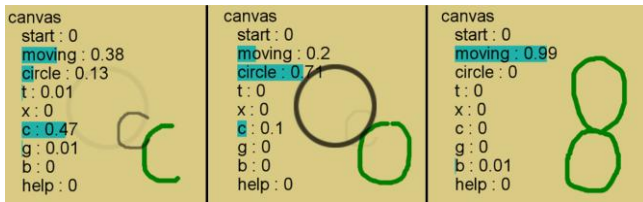


Figure 8: Screenshots of canvas interactor as user draws a figure 8. **(left)** Initially a ‘c’ gesture is most likely (probability 0.47). Feedback indicates that c is the most likely interpretation, though a circle is possible. **(middle)** The system becomes confident that the gesture is a circle, reflecting this in feedback (also evident in state distribution). **(right)** As the user completes the gesture, the canvas believes user is painting and removes all gesture feedback.

input events. Another source of uncertainty is recognized input, as in the case of gesture recognition. This uncertainty can be especially problematic when the recognizer is inaccurate or the gesture set is large.

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 (as in the OctoPocus system [1]) because of the high degree of ambiguity that results.

Our 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 interpretation. During event sampling, this event is divided into event samples for individual gestures (i.e., a circle gesture, a ‘c’ gesture, etc.), weighted by likelihood.

We developed a canvas interactor for our paint application that handles both drag (for painting) and gesture events. Figure 9 shows the gestures the canvas recognizes. The canvas also provides feedback about the canvas state (paint and interpreted gestures) using transparency. Figure 8 illustrates what happens when the user paints a figure 8 on the canvas. Because 8 shares a common prefix with ‘c’ and ‘circle’, the top hypothesis shifts from ‘c’ to ‘circle’ to simply painting on the canvas as the user draws.

Because the canvas provides visual feedback about the top recognized gestures, the user can change her gesture in real time to disambiguate. The feedback also allows the user to know when her gesture is ‘good enough’ to be interpreted, or (if she intends to paint something that looks like a gesture) the user can “cancel” recognition without affecting

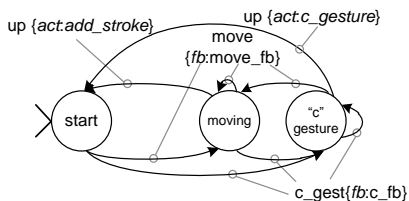


Figure 10: Partial state machine description for canvas interactor.

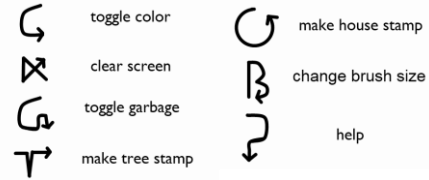


Figure 9: Gestures recognized by the canvas.

the intended drawing by reversing direction and retracing part of the drawing before lifting the finger.

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 our 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, as shown in Figure 10. The system handles all logic relating to tracking probabilities and deciding between paint and gesture events. Because each gesture is handled in a separate state, the state machine for the canvas is quite large. Figure 10 illustrates a portion of the machine.

Beyond Touch: Accelerometer-based interaction

Although all of the input discussed so far is based on touch, our framework is not limited to touch. Any probabilistic event can be handled using the same mechanisms. For example, a shake event can be modeled with probability based on the vigor of the shake. We have built a recognizer that generates probabilistic shake events from accelerometer readings. We built a shake interactor which takes these shake events, provides feedback about the shake vigor and executes actions when shake events are above a developer-specified threshold.

Putting it all together

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 (until the user lifts his/her finger). The framework we 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 our framework, this complex application simply works.

Consider the situation when the user intends to do a circle gesture but accidentally selects a stamp (Figure 1). This demonstrates uncertainty about the touch target (canvas or

stamp), type (gesture or drag) and location. Since both possible targets process the input during dispatch, both the canvas and stamp will show feedback based on how likely each corresponding action is. The user continues the circle gesture and releases when she sees that the system will correctly interpret her results. If instead the user accidentally draws a circle when she intends to move the stamp, she could simply move her finger back and forth so that the circle gesture is no longer recognized.

The framework's support for tracking the *compatibility* of actions easily supports multitouch. As with any multi-touch interface, each interactor must include logic to make sure that it responds only to touch events that have the correct touch ID. For example, in our implementation, multiple stamps may be moved simultaneously, and any number of simultaneous fingers may be painting on the canvas. Because touch events with different IDs are considered compatible, our framework handles all logic for accepting and canceling events correctly.

The power of our framework is clear when all of these interactors are combined together. While it is feasible to write these interactors individually using conventional input, writing an application which correctly handles uncertainty *across* interactors would be extremely difficult, and not reusable. Our framework provides a general method not only for handling uncertain input, tracking interactive state and resolving ambiguous actions.

CONCLUSIONS AND FUTURE WORK

The arrival of new recognition-based input technologies requires that applications handle input with uncertainty. This presents new challenges for developers. Our framework supports managing interactive state, feedback and action without requiring developers to think probabilistically. Our contribution greatly simplifies the task of developing interactors that handle uncertain input, and brings us one step closer to developing an end-to-end toolkit for handling uncertain input.

The examples in the previous sections represent illustrations of the complex interactors that our system can accurately support including touch, gesture, and accelerometer input. In the future we hope to expand this library of interactors to make use of additional forms of uncertain inputs. Although our example feedback methods (*e.g.*, changing the transparency of various drawn objects) are effective, they only begin to explore the space of possible feedback for ambiguity. Future work will also consider more complex approaches to feedback.

ACKNOWLEDGEMENTS

Removed for anonymous review.

REFERENCES

1. Bau, O., Mackay, W. "OctoPocus: a dynamics guide for learning gesture-based command sets." in *Proc. UIST 2008*, 37-46.
2. Bourguet, M. L. "A Toolkit for Creating and Testing Multimodal Interface Designs." Posters and Demos from UIST'02, Paris, Oct. 2002, pp. 29--30.
3. Chang, B.-W., Mackinlay, J.D., Zellweger, P.T., and Igarashi, T. A negotiation architecture for fluid documents. in *Proc. UIST '98*, 123-132.
4. Dumas, B., Lalannel, D., Guinard, D., Koenig, R., and Ingold, R. "Strengths and weaknesses of software architectures for the rapid creation of tangible and multimodal interfaces." in *Proc. TEI 2008*, 47-54.
5. Dumas, B., Lalannel, D., Oviatt, S. "Multimodal Interfaces: A Survey of Principles, Models and Frameworks." *Human Machine Interaction*. vol. 5440, pp. 3-26, 2009.
6. Gordon, N. J., Salmond, D. J., Smith, A. F. M. "Novel approach to nonlinear/non-Gaussian Bayesian state estimation". *IEE Proceedings F on Radar and Signal Processing* **140(2)**:107-113.
7. Grover, D., King, M., Kushler, C., "Reduced keyboard disambiguating computer." U.S. Patent 5818437, Oct. 6, 1998.
8. Hudson, S. E., Newell, G. L. "Probabilistic state machines: dialog management for inputs with uncertainty." in *Proc. UIST 1992*, 199-208.
9. Jacob, R., Deligiannidis, L, Morrison, S. "A software model and specification language for non-WIMP user interfaces." *TOCHI* **6(1)**:1-46, 1999.
10. Mankoff, J., Hudson, S. E., Abowd, G. D. "Interaction techniques for ambiguity resolution in recognition-based interfaces." in *Proc. UIST 2000*, 11 - 20.
11. Mankoff, J., Hudson, S. E., Abowd, G. D. "Providing integrated toolkit-level support for ambiguity in recognition-based interfaces." in *Proc. CHI 2000*, 368-375.
12. Maskell, S., Gordon, N. "A tutorial on particle filters for on-line nonlinear/non-Gaussian Bayesian tracking," *Target Tracking: Algorithms and Applications IEEE* vol. 2, pp. 16-17, 2001.
13. Metropolis, N., Ulam, S. "The Monte Carlo Method". *Journal of the American Statistical Association* **44(247)**:335-341, 1949.
14. Myers, B., and Kosbie, and Kosbie, D. "Reusable hierarchical command objects." in *Proc. CHI 1996*, 260-267.
15. Odell, J. "The use of context in large vocabulary speech recognition." PhD thesis, University of Cambridge, England, 1995.
16. Olsen, D. "Propositional production systems for dialog description." in *Proc. CHI 1990*, 57-64.
17. Oviatt, S. "Ten myths of multimodal interaction." *CACM* **42(11)**:74-81, 1999.
18. Schwarz, J., Hudson, S., Mankoff, J. "A Framework for Robust and Flexible Handling of Inputs with Uncertainty." in *Proc. UIST 2010*, 47-56.
19. Wasserman, A.I. "Extending State Transition Diagrams for the Specification of Human-Computer Interaction." *IEEE Transac. Software Engineering*. **11(8)**:699-713, 1985.