Multimodal Event Parsing for Intelligent User Interfaces

Will Fitzgerald
Kalamazoo College
Kalamazoo, MI USA
+1 269 337 5721
wfitzg@kzoo.edu

R. James Firby
Michael Hannemann
I/NET, Inc.
Chicago, IL USA
+1 773 255 4702
firby,hannemann@inetmi.com

ABSTRACT
Many intelligent interfaces must recognize patterns of user activity that cross a variety of different input channels. These multimodal interfaces offer significant challenges to both the designer and the software engineer. The designer needs a method of expressing interaction patterns that has the power to capture real use cases and a clear semantics. The software engineer needs a processing model that can identify the described interaction patterns efficiently while maintaining meaningful intermediate state to aid in debugging and system maintenance.

In this paper, we describe an input model, a general recognition model, and a series of important classes of recognition parsers with useful computational characteristics; that is, we can say with some certainty how efficient the recognizers will be, and the kind of patterns the recognizers will accept. Examples illustrate the ability of these recognizers to integrate information from multiple channels across varying time intervals.

Categories & Subject Descriptors: I.5.5 [Pattern Recognition]: Implementation – special architecture

General Terms: Human Factors

Keywords: Multi-modal parsing, event recognition, CERA

INTRODUCTION
Many intelligent user interfaces need to be *multimodal*, that is, allow the user of the interface to interact with a system using multiple channels. For example, an effective in-car navigation device may allow the driver to click on an onboard map display and say "take me here;" the navigation system may use event data from the internal car networks and an onboard global position system to determine car location and path planning, and then to provide appropriate (audio and visual) driving directions.

System designers and implementers often find it useful to model interfaces in terms of user events that trigger system actions. As systems become more complex, it is often patterns of user activity that dictate which actions should be performed, rather than single events. The ability to represent and identify complex user input patterns is key to the design of interfaces to such systems.

IUI'03, January 12–15, 2003, Miami, Florida, USA.

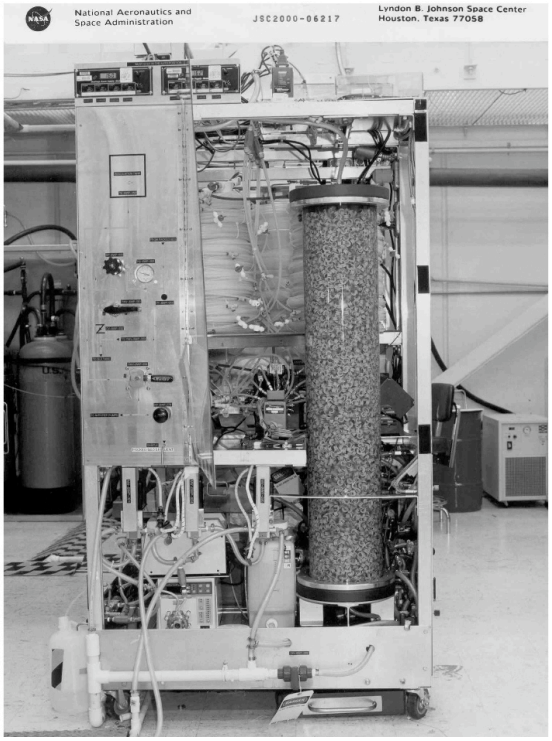

Figure 1: Biological Water Processor Component of Water Recovery System (NASA Photo)

In addition to modeling user input, system designers must also monitor system changes and present useful information back to the user. Often this requires filtering or distilling system state information into notifications with a more abstract meaning. Again, as systems become more complex, detecting and classifying state changes often involves monitoring multiple data streams for semantically meaningful, multimodal patterns.

For example, consider a human/computer interface built for the control and monitoring of an integrated Waste Recovery System (which recycles urine and waste water into water usable for drinking and other functions) at NASA's Johnson Space Center [14]. The system in question has several large subcomponents, including a biological water processor (see Figure 1), a reverse osmosis system, an air evaporation sys-

tem for brine recovery, and a post-processing system. All in all, nearly 200 different data streams are available.[1]

Detecting interesting patterns is complicated because there are so many data streams, and each subcomponent's data are logged asynchronously. For example, in case of a loss of communication from the control system, the entire waste recovery system must begin procedures to ensure that it is in a safe state. Each subsystem has its own safing procedures. Detecting that safing is complete varies among the subsystems. Each subsystem has its own measurement or combination of measurements which indicate it has gone into safe mode. Further, each subsystem arrives at a safe condition independently from the others. For example, the reverse osmosis system is in a safe mode if its effluent flow meter is below a certain threshold value; the post-process system is in a safe mode if flow pressure is below a certain value, the O_2 concentrator is off and O_2 flow drops below a threshold.

As Oviatt states [13], it is a myth that "multimodal" primarily means speech plus pointing. Even direct user input can take many forms: other body gestures, drawings, and orthographic symbols are common human-to-human interactions in addition to speech and pointing. Input that comes from sources other than the user can vary practically without limit.

Recognizing patterns in data from multiple channels is made more difficult by this wide variety of potential input types. Further, the temporal relationships among data are not simple. For example, Oviatt cites another myth: that a deictic gesture always occurs simultaneously with a deictic utterance. Rather, the gesture often comes before or after the utterance: "synchrony does not imply simultaneity."

By making certain simplifying assumptions about the form input data will take, we can describe classes of recognition parsers that are useful for finding patterns in both user input and system output. These parsers have well defined computational characteristics; that is, we can say with some certainty how efficient the recognizers will be, and what kind of patterns the recognizers will accept. We turn to describing an input model, a general recognition model, and a series of important recognizer classes.

EVENT RECOGNITION

Our model for understanding multimodal input is to recognize interaction patterns both within and across input modes. The model draws on a tradition of "understanding as recognition" that has been used extensively for natural language processing [4, 10]. The key to this approach is to look for hierarchical patterns associated with conceptual representations in an input stream of individual words, much like chart parsing. We extend this approach to include patterns that combine "events" from multiple input modes using a variety of temporal relationships beyond the simple total order of words in natural language. When a pattern is recognized it may generate additional events to be recognized by other patterns, allowing a rich mixture of hierarchical, recursive recognition.

[1] From the biological water processor, 76 data values; from the reverse osmosis system, 42 data values; from the air evaporation system, 51 data values; from the post-processing system, 30 data values.

Typical multimodal user interfaces process each mode of interaction as a separate input stream and attempt to build structured representations of the meaning of each stream separately. These representations are then combined by some sort of integration module. Our approach allows individual recognizers to process input from any combination of modalities. System processing can use separate patterns within each mode when that makes sense, or patterns can combine events from different modes right from the beginning. In either case, the meaning of various user interactions is constructed by pattern recognition alone, with no post-recognition integration module.

We have chosen to define pattern recognizers as separate computational units, rather than as input to a multi-dimensional chart parser, to simplify the characterization of the computational properties of each type of recognizer (or pattern) and to simplify system development by allowing custom event matching algorithms within individual recognizers when necessary.

Example

Consider the standard example: speaking "Go here" while synchronously tapping on a map display. A speech recognition system produces the stream of temporally ordered word events "go" and "here." The tapping produces a tapping event containing the x, y coordinates of the tap. A typical event stream, with each event's duration identified by a start and finish time, might look those in Table 1:

Start	Finish	Event
100	100	(word "go")
105	105	(tap 22 87)
112	112	(word "here")

Table 1: Example multimodal event stream with start and finish times in milliseconds.

Here, the tapping is recorded as being simultaneous with the first word recognition, but it could also precede or follow or be simultaneous with either word event; to be meaningful, it only has to be synchronous with the word events; i.e., occur within some time window.

In the system we have built, we could recognize this synchrony with the pattern[2] in Figure 2.

```
(within (all (tap ?x ?y)
     (in-order (word "go")
          (word "here")))
    P0.250S)
```

Figure 2: Example multimodal event recognition pattern

This pattern is written to predict that, within a duration[3] of 250 ms., both a tap event and the recognition of the sub-pattern of the words "go" "here" in order, will occur, with

[2] For illustration, we insert the words in the patterns directly in this example. It is likely that we would define a patterns which represented the semantic content of "go here" would be more useful.

[3] P*y*Y*m*M*w*W*d*DT*h*H*m*M*n*S is the ISO-8601 standard for representing a duration.

the x, y coordinates bound to the variables ?x and ?y respectively.

GENERAL EVENT MODEL

We begin by assuming that all input to a recognizer (user or otherwise) takes the form of *discrete, time stamped, typed events*. By *discrete* we simply mean that incoming events are individually distinct, non-continuous data. Of course, continuous data can be transformed into discrete data in various ways; for example, a speech stream can be converted into a sequence of discrete words using speech recognition techniques; other continuous waveforms can be effectively converted to discrete data via the Fourier transform, etc.

By *time stamped*, we mean that each input event has associated with it a start time and a finish time, such that the start time is ≤ the finish time. Given time stamps, events can be put in a total order with respect to the intervals the time stamps define [1]. We further assume that, in general, events are signaled in monotonically non-decreasing order of their finish times. For example, the three words in the sentence "take me here" are signaled in temporal order. This assumption is not a very strong one; the greatest practical difficulty is ensuring that event signals do, in fact, arrive in proper order and that the various input channels have the same clock.

By *typed*, we mean that events can be placed into distinct classes, and that each class defines an equality predicate, =, such that if *a,b* are events of the same class, and $a=b$, then *a* and *b* are the same, otherwise they are not the same[4]. Again, this assumption is not very strong; it merely forces the recognition system to distinguish among different kinds of events (for example, words vs. mouse clicks) and distinguish when a looked-for event has been seen. For example, if a recognizer predicts that the word "here" will occur next, it can recognize "here" when it arrives, or distinguish it from another word[5].

GENERAL RECOGNIZER MODEL

We model a parser, or (more generally) an event pattern recognizer, as a function mapping events to {*ignore, active, futile, complete*} × {[*start, finish*]}, where *start* and *finish* are times. In other words, the function maps an event to a completion state plus a start time and a finish time. The semantics of this function depend on the value of the completion state. If the value is *ignore*, the event is irrelevant to the recognizer. If the value is *active,* the event is relevant to the recognizer, but the recognizer (perhaps after changing some internal state) still requires additional events before recognizing the pattern. If the value is *futile*, the presence of this event indicates that this recognizer is not able to recognize the pattern. If the value is *complete*, the pattern has been recognized.

Practically, recognizers will usually maintain some kind of internal state. For example, a recognizer for the pattern *a, b, c* in order will need to maintain (at a minimum) how much of the pattern has already been recognized so it can determine when it is *complete* and when it is still *active*. From a software engineering point of view, the ability to inspect this state is a crucial element in monitoring and debugging recognizer-based interfaces.

The start and finish times define the interval over which the recognizer completely recognized the pattern, or the interval over which it remained active or became futile.

TYPES OF RECOGNIZERS

The general recognizer model presented is an abstract one, to be sure. However, specific classes of recognizers can be identified that are of more concrete use. Several useful recognizer classes are *one* recognizers, *binding* recognizers, *in-order* recognizers, *one-of* recognizers, *all* recognizers, *Allen* recognizers, *within* recognizers, and *without* recognizers.

In general, we use [*type* $element_1$ $element_2$ ... $element_n$] to define a recognizer, where *type* is the type of recognizer (*one*, *binding*, etc.) and the $element_n$ are the elements of the pattern to be recognized. For all of the recognizer types we will define, an element of a pattern can either be an event form (i.e., an appropriate parameter to an event equality predicate) or another recognizer. This will allow us to compose recognizers and recognize complex patterns of events.

One Recognizers

One recognizers look for exactly one event (or sub-recognizer). They typically form the basis of event transducers that convert events from one type to another or simple event handlers, that take action when an event is seen. Given the recognizer [one *i*] where *i* is an event form, and a probe event *p,* the recognition function is *complete* if $i=p$, and *ignore* otherwise, with the start and finish times taken from *p*. Given the recognizer [one *r*], where *r* is a recognizer, the recognition function has the same value as *r*.

For a given probe event, the computational complexity of a *one* recognizer is clearly the same as the complexity of the event equality predicate if the target is an event form, or, if the recognizer is for a sub-recognizer, the same as the complexity of the sub-recognizer. In general, the computational complexity of an event recognizer depends on either the complexity, *e*, of the equality predicate or the complexity of the constituent sub-recognizers. To simplify the discussion, we give complexity in terms of *e*.

Binding Recognizers

Binding recognizers are similar to *one* recognizers, but they return state in addition to recognizing an item. In essence, they *bind* the value of the recognized item to an attribute. These can be represented as [binding *attribute constraint*], where *attribute* is the name of an attribute, and *constraint* is a function which maps from events to the Boolean set {true, false}. Given a recognizer of this form, and a probe event *p*, the recognition function is *complete* if *constraint(p)* is true, and *ignore* if *constraint(p)* is false. Further, if the recognition function is *complete*, then the attribute is bound to *p*.

Binding recognizers are especially useful for recognizing hierarchical events, such as natural language syntax and semantics, especially in conjunction with the multiple-item recognizers that follow. Indeed, if the *constraint* is a stan-

[4] This is without regard to the start and finish times of the event.

[5] In fact, the recognizer might only make this distinction probabilistically, using standard statistical techniques.

dard unification algorithm, *binding* recognizers are essentially unification parsers. An example is given below for *in-order* recognizers.

For a given probe event, the computational complexity of a *binding* recognizer is the same as the complexity of the constraint predicate, assuming that creating a binding between an attribute and a value can be done in constant time.

In-order Recognizers

In-order recognizers recognize patterns of events (and sub-recognizers) that occur in temporal order. Here, we introduce the idea of *event contradiction*. The basic idea is this: in a complex, multimodal input environment, many events will occur that are not relevant to the completion or futility of an *in-order* recognizer. For example, if the recognition pattern is [in-order "take" "me" "here"], an event coming from a display click is not relevant to this particular pattern.[6] If, however, the recognizer has seen "take" and "me," and then sees, for example, "dog," the recognizer should fail. In other words, the event equality predicate is not sufficient; we need another predicate function which maps from events × events to the Boolean set {true, false}. This predicate can be as simple or as complex as necessary for a given application.

In-order recognizers need to maintain state about elements that remain to be seen, and elements that have been seen. We'll call the former the *remainder list* (*rem)* and the latter the *seen set* (*seen*). The semantics of the recognizer, for a probe event *p*, are:

Return *complete* if:

> $|rem|=1$ and $p = first(rem)$, or
>
> $|rem|=1$ and *first(rem)* is a recognizer and *p* completes *first(rem)*;

Otherwise, return *active* if:

> $|rem|>1$ and $p=first(rem)$, (adding *p* to *seen*, and setting *rem* to *rest(rem)*) or
>
> $|rem|>1$ and *first(rem)* is a recognizer and *p* completes *first(rem)* (setting *rem* to *rest(rem)*) or
>
> *first(rem)* is a recognizer and *p* makes *first(rem)* active

Otherwise, return *ignore* if:

> $|rem|>1$ and *first(rem)* is an event and $p \neq first(rem)$ and *p* contradicts no member of *seen*,
>
> *first(rem)* is a recognizer and *p* is ignored by *first(rem)* and *p* contradicts no member of *seen*;

Otherwise, return *futile*.

These conditions are somewhat complicated, but the general idea is to recognize the sub-elements in order, and fail if there is a contradiction to what has already been seen, unless, of course, what is being looked for is just that item at that point. The start and finish time of the recognizer is the start time of the first event placed into the seen set, and the finish time of the last event seen before the recognizer completes.

[6] It is likely to be relevant to a larger pattern, but not the linguistic pattern.

For a given probe event, the computational complexity of an *in-order* recognizer has a number of components. First is the complexity *e* of the equality predicate. Second, is the amortized cost of checking for a contradiction. In other words, if a recognizer has, on average, *m* events in the seen set, and the cost of checking that two events contradict is *c*, then this cost component is *m* times *c*. Thus the cost of a probe is $e+mc$, where *e* is the cost of the equality predicate, and *c* is the cost of the contradiction predicate.

In special cases, versions of *in-order* recognizers can be defined in which "contradiction" is defined in less general terms (as for example, in classic string-matching algorithms). These recognizers will be more efficient.

As an example, consider the event pattern [in-order a b c d e], where a, b, … e are event forms and -a, -b,…, -e indicate event contradiction. Table 2 shows three example streams of events, and the results after each event signal.

a	b	c	d	e	…	
ACT	ACT	ACT	ACT	COM		
a	b	c	c	d	d	e
ACT	ACT	ACT	IGR	ACT	IGR	COM
a	b	c	c	-b	…	
ACT	ACT	ACT	IGR	FUT		

Table 2: [*in-order* a b c d e] with three different event streams and recognizer results. ACT is *active*, COM is *complete*, FUT is *futile* and IGR is *ignore*.

All Recognizers

All recognizers are similar to *in-order* recognizers. The basic semantics are that all of the items must be seen for the recognizer to complete, but the order in which they happen is not important. The issue of event contradiction remains significant. Event contradiction causes an event that has been seen to be put back into the set of items remaining to be seen. This ensures that an *all* recognizer completes only when all of its constituent event forms have been seen together.

Here, *rem* is the *remainder set*, the remaining items to be recognized; *seen* is the *seen set*, the items which have been seen. The recognition semantics, for a probe event *p*, are:

Return *complete* if:

> $|rem|=1$ and $p=i$, an event form which is the only element of *rem,* or
>
> $|rem|=1$ and *p* completes *i*, a recognizer which is the only element of *rem*;

Otherwise, return *active* if:

> $|rem|>1$ and $\exists i$ such that *i* is an event form and $i=p$, (adding *p* to *seen*, and setting *rem* to $rem-\{i\}$) or

(algorithm continued)

> $|rem|>1$ and $\exists s$ such that *s* is an event form in *seen* and *s* contradicts *p*, (setting *seen* to $seen-\{s\}$) or

$|rem|>1$ and $\exists i$ such that i is a recognizer and p completes i (setting *rem* to *rem*-{i}) or

Otherwise, return *ignore*.

As for *in-order* recognizers, the start and finish time of the recognizer is the start time of the first event placed into the seen set, and the finish time of the last event seen before the recognizer completes.

a	b	c	d	…		
ACT	ACT	ACT	COM			
d	a	c	b	…		
ACT	ACT	ACT	COM			
a	b	c	b	-c	d	c
ACT	ACT	ACT	IGR	ACT	ACT	COM

Table 3: [*all* a b c d] with three different event streams and recognizer results.

The computational complexity of *all* recognizers is similar to *in-order* recognizers. The cost of the equality predicate must be amortized over the average number of items in the remainder, and the cost of the contradiction must be amortized over the average number of items in the seen set. Thus, the cost of a single probe is $ne+mc$, where n is the average number of items in the remainder, e is the cost of the equality predicate, m is the average number of items in the seen set, and c is the cost of the contradiction predicate.

As an example, consider the event pattern [all a b c d e], where a, b, … e are event forms and -a, -b,…, -e indicate event contradiction. Table 3 shows three example streams of events, and the results after each event signal.

One-of Recognizers

One-of recognizers complete as soon as one of their pattern items completes. No state needs to be maintained, and contradiction does not obtain. The recognition semantics, for a probe event p, are:

Return *complete* if:

There is some event form i such that $p=i$, or

There is some recognizer and it returns complete on p;

Otherwise, return *futile* if:

All pattern items are recognizers that have become *futile*, or

Otherwise, return *active*.

The start and finish times are the start and finish time of the event or recognizer that completed.

The computational complexity of a *one-of* recognizer with n elements is ne, where e is the cost of the equality predicate.

Allen Recognizers

James Allen described the relationships between temporal intervals in [1]; a list of these is given in the appendix. These form a super-class of recognizers of one of these forms:

[*relation element*$_1$ *element*$_2$] or

[*relation element*$_1$ *start finish*]

where *relation* is one of the Allen temporal relationships, *element*$_n$ is an event form or recognizer, and *start* and *finish* describe a temporal interval (a start time and a finish time).

The semantics of the *Allen* recognizers are *complete* if the relation holds between the elements or between the element and the specified interval; *futile* if an element is futile; *ignore* if all elements ignore an event, or *active*, otherwise. The start and finish times depend on the individual relationship involved. The computational complexity for an event probe p is $2e+\Box$, where e is the cost of the event equality predicate (for event forms) or for recognizers, the cost of the recognizer, and $\Box$ is the cost of comparing intervals.

Within Recognizers

A *within* recognizer succeeds if an element occurs whose duration (finish time less start time) is no longer than a specified amount of time. Specifically, given [*within element duration*], the semantics for a probe event p with start time s and finish time f:

If *element* is an event form, then

Return *complete* if:

p=*element* and (f-s)≤*duration,*

Return *futile* if:

p=*element* and (f-s)>*duration,*

Otherwise, return *ignore*.

If *element* is a recognizer, then

Return *complete* if:

p makes *element* complete and the start time-finish time of the *element* is ≤ *duration,*

Otherwise, return *futile* if:

p makes *element* futile, or *element* returns *complete* and the start time-finish time of the *element* is > *duration*

Otherwise, return *ignore* if:

p is ignored by *element*

Otherwise, return *active*.

The computational cost of a *within* recognizer $e+\Box$; that is, the cost of matching the *element* plus the cost of the interval comparison.

Without Recognizers

The basic idea of a *without* recognizer is that the recognizer will succeed if an interval passes without an event element succeeding, and it will fail otherwise. Specifically, given [*without element begin finish*], the semantics for a probe event p with start time s and finish time f:

If *element* is an event form, then

Return *complete* if:

p=*element* and s>*finish*, or

Otherwise, return *futile* if:

p=*element*, or

Otherwise, return *ignore*.

If *element* is a recognizer, then

Return *complete* if:

p makes *element* return *futile,*

Otherwise, return *futile* if:

p makes *element* return *complete* and $s \leq finish$;

Otherwise, return *ignore* if:

p is ignored by *element*

Otherwise, return *active*.

On *complete*, the start time of the recognizer is *begin* and the finish time is *finish*.

The computational complexity for an event probe *p* is $e+\Box$, that is, the cost of matching the *element* plus the cost of the interval comparison.

SIGNALING ALGORITHM

Given a recognizer, it is useful to consider associating with it a *callback* procedure; that is, a procedure that is called when the recognizer completes. The parameters for this procedure are the start and finish times of the recognizer at completion as well as any state the recognizer might have built (through *binding* recognizers).

```
S ← {}; // states for recognizer callbacks
F ← {}; // the recognizers which have completed
For each r in R do:
    result, start, finish, state ← r(event);
    case result:
        futile: F ← F ∪ {r};
        complete:
            F ← F ∪ {r};
            S ← S ∪ {[r, start, finish, state]};
    end case;
end for;
R ← R – F;
for-each(λ(s)(apply first(s).callback, rest(s)),S);
```

Figure 2: Signaling Algorithm

A recognizer function is *active* if it is registered to receive event signals. Call the set of active recognizers *R*.

Signaling an event (with associated start and finish times) is done with the algorithm in Figure 2.

Essentially, the algorithm sends an event to all the registered recognizers. If a recognizer returns *futile*, it is marked for removal. If a recognizer *completes*, it is also marked from removal, but it and its data are stored so that its callbacks can be run. After all of the registered recognizers receive an event and return a result, the recognizers marked for removal are removed, and all of the successful callbacks are run. Thus, the computational complexity of signaling an event is the sum of the computational complexity of the active recognizers.

Note that one of the possible things to do in a callback is to signal another event; in this way, recognizers can affect one another and hierarchical, complex events can be recognized.

CURRENT IMPLEMENTATION

This model for event pattern recognition described has been implemented as CERA, the Complex Event Recognition Architecture. CERA is written in Common Lisp. In addition to recognizer definition and execution, CERA also provides an event visualization tool and an Integrated Development Environment based on the Eclipse extensible IDE [2].

Another example

CERA is being developed as a general event recognition tool. As mentioned above, it is currently being used in a demonstration project at NASA's Johnson Space Center in the context of a large, multi-station, multi-channel monitoring project for water recovery [14].

As an example of a event pattern recognizer definition in CERA, we consider what it means to recognize that the water recovery system has gone into safe mode. The waste recovery system is in safe mode when each of its four subsystems go into safe mode. Because the subsystems go into safe mode asynchronously from each other, determining that all of the subsystems have entered into safe mode is a good example of an *all* recognizer. The form used to recognize this is[7]:

```
(define-recognizer (safing-complete)
 (pattern
  `(all
   (safing (system pbbwp) (status on))
   (safing (system ro) (status on))
   (safing (system aes) (status on))
   (safing (system pps) (status on))))
 (on-complete (st end)
  (signal-event `(all-safed) st end)))
```

The *pattern* clause introduces a recognizer form. The *on-complete* clause introduces forms to execute when the recognizer completes. In this example, a synthesized *safing-complete* event is signaled. This is, in turn, the second step of the *in-order* recognition pattern of communication loss, safing completion communication required, and restart complete.

This pattern is part of a set of patterns used to recognize loss and recovery of communication in this application. On a 700 Mhz. Windows 2000 machine with 256 Mb of memory running Allegro Common Lisp 6.1, CERA processes an event in approximately 1.5 ms. (2346 base events, representing two days of data, in 3.55 seconds).

OTHER WORK

Multimodal human computer interfaces tend to be based on research in vision; task execution, planning and dialogue systems; natural language parsing; or combinations of these. For example, the recognition subsystems for the Intelligent Classroom [6] are inspired by the vision architecture of the Gargoyle vision system [3] and Horswill's context-dependent visual routines [7]. Recognition subsystems for the Intelligent Classroom are built using pipelines of perceptual routines that can be configured at run-time. Such processing pipelines are similar in spirit to CERA event pattern recognizers although much more specialized to visual processing. In fact, the two approaches to recognizing complex events are complimentary: CERA could benefit from a gen-

[7] This has been slightly edited to fit the space constraints of this article.

eral low-level event detection system like Gargoyle and the Intelligent Classroom could benefit from the abstract language for event pattern recognition that CERA provides.

TRIPS, a dialogue management system that integrates a spoken language interface and other modalities, has recently undergone an architectural reworking of its core functionalities [1]. Among the changes is the addition of a "behavioral agent" that monitors for internal and external asynchronous events. This change is motivated by the incremental nature of dialogue; a strict "turn-taking" model does not model human-human dialogue well, and makes human-computer interactions "unnatural and stilted." These behavioral agents are quite similar in motivation and functionality to the event recognizer architecture described here.

This idea of a separate module for monitoring input from multiple modalities is common. For example, the Open Agent Architecture [11, 12] has a "modality coordination agent" that is responsible for producing "a single meaning that matches the user's intention" from multiple inputs. In general, understanding a person's intentions involves much more than recognizing patterns of behavior. However, the event recognizers we propose define an approach to recognizing patterns across multiple modes on input that can then be used as the foundation for understanding intentions.

The idea of using parsing techniques for multimodal event pattern recognition is not unique to this paper. For example, Johnston argues both for unification-based multidimensional chart parsing [8] as well as finite-state parsing techniques [9]. Indeed, these two approaches are at two ends of a scale. On the one hand, unification-based parsing provides a very general declarative framework for multimodal parsing with relatively expensive computational properties. On the other hand, compiling multimodal grammars to finite state machines are very efficient, with a concomitant loss in generality and transparency. The approach to event parsing defines an intermediate strategy in which recognizers can be made reasonably efficient while maintaining both a clear semantics and meaningful internal state to aid in software system development and monitoring.

EVENT RECOGNITION AND SEMANTIC PARSING

Our own work is situated in research on semantic parsing techniques and task execution systems. DMAP [10], for example, is a system that parses natural language directly into semantic descriptions of a frame-based memory model. DMAP-style patterns strongly influence the *in-order* and *binding* recognizers described above. In fact, one can build semantic parsers in the DMAP style by combining *in-order* recognizer and *binding* recognizers.

In DMAP, concepts in memory are arranged in a hierarchy, and can have a set of (inheritable) "slots," or attribute/value pairs, in which the values are typically conceptual representations ("concepts") themselves (and thus form a partonomy). DMAP patterns are attached to the concepts in the memory model. Each pattern consists of a sequence of items: each item can either be a grounded literal or an attribute/constraint pair. Patterns create predictions that the items in the pattern will be seen *in order*; when the sequence is completed, the associated concept is referenced. Grounded literals are matched directly with user inputs. An attribute/constraint pair in a pattern for a concept creates a dynamic prediction for something which fulfills the constraint; if another concept is referenced which fulfills the constraint, it is used as the value of an attribute/value pair and passed as *bindings* to the base concept when it is eventually referenced[8].

Consider a frame LOVE with two attributes ACTOR and □-OBJECT, with ACTOR constrained to be an M-PERSON and □-OBJECT constrained to be in the set {PHYS.OBJ, ABSTR.OBJ}. Further, let *frame:pattern* be shorthand for "If *pattern* completes, signal *frame* with associated bindings." Let {*attribute*} be shorthand for [binding *attribute* □(x).*true* if *x* meets the constraints of *attribute*; *false* otherwise]. Let PAT and CHRIS be defined as instances of PERSON, a subset of PHYS.OBJ, and JUSTICE be an ABSTR.OBJ. Further, consider these frame patterns:

PAT: [one "Pat"]

CHRIS: [one "Chris"]

JUSTICE: [one "justice"]

LOVE: [in-order {ACTOR} "loves" {□-OBJECT}]

Then parsing "Pat loves Chris" results in the concept description:

LOVES with {ACTOR=PAT, □-OBJECT=CHRIS}

And parsing "Pat loves justice" results in the concept description:

LOVES with {ACTOR=PAT, □-OBJECT=JUSTICE}.

The work described in this paper is motivated our desire to extend the semantic parsing model in order to "parse the world." But parsing the world requires extending the parsing model to allow a much wider set of event pattern types, including parsing event patterns that come from multiple event streams, and the recognizer classes described above are an attempt to do this.

CONCLUSION

Our complex event parsing model makes relatively few commitments as to the structure of multimodal data, requiring only that the data be discrete (or discretized) typed time-stamped data which are signaled in order of their endpoints. Event pattern recognizers can be of arbitrary form (in fact, both unification and finite-state parsers can created using this model). We have identified, however, several classes of event pattern recognizers we have found to be especially useful in our own work. This model has allowed us to create an efficient event pattern recognition system for a large multi-channel project. The Complex Event Recognition Architecture provides a system for parsing multimodal input, integrating and parsing data from multiple channels. Future work will involve building this architecture into other multimodal human-computer interface systems and integrating it more fully into the Dynamic Predictive Memory Architecture [5], a software architecture for intelligent task and dialog control.

---

[8] Code for a basic frame system and a DMAP-style parser (in the Common Lisp programming language) can be found at http://kzoo.edu/~wfitzg/icp.html.

Relationship	Semantics
contains(i_1,i_2)	$s_1 < s_2 < f_2 < f_1$
finishedBy(i_1,i_2)	$s_1 < s_2$; $f_1 = f_2$
startedBy(i_1,i_2)	$s_1 = s_2$; $f_1 < f_2$
before(i_1,i_2)	$f_1 < s_2$
meets(i_1,i_2)	$f_1 = s_2$
overlaps(i_1,i_2)	$s_1 < s_2 < f_1 < f_2$
equals(i_1,i_2)	$s_1 = s_2$;$< f_1 = f_2$
overlappedBy(i_1,i_2)	$s_2 < s_1 < f_2 < f_1$
metBy(i_1,i_2)	$s_1 = f_2$
after(i_1,i_2)	$s_1 > f_2$
starts(i_1,i_2)	$s_1 = s_2$; $f_1 < f_2$
finishes(i_1,i_2)	$s_1 > s_2$; $f_1 = f_2$
during(i_1,i_2)	$s_2 < s_1 < f_1 < f_2$

Table 4: Allen's 13 Possible Relationships between two intervals i_1 and i_2, with start times s_1 and s_2 and finish times f_1 and f_2, respectively.

ACKNOWLEDGMENTS

CERA is being developed under NASA contract #NAS9-00122.

APPENDIX: INTERVAL RELATIONSHIPS

Allen [1] defined a covering set of the relationships that can obtain between two intervals. These relationships are listed in Table 4. Less strictly, two intervals can be ordered just by their start times or just by their finish times, using the standard relationship $\{< \leq = \geq >\}$. For example, it may just be of interest that the start time of one interval is equal to the start time of the second interval, without regard to the ordering of their finish times.

REFERENCES

1. Allen, J., Maintaining knowledge about temporal intervals. Communications of the ACM, 1983. 26(11): 832-843.
2. Eclipse.org, Eclipse Integrated Development Environment. 2002, http://www.eclipse.org.
3. Firby, R.J., et al. An architecture for vision and action. Proceedings of International Joint Conference on Artificial Intelligence. 1995.
4. Fitzgerald, W., Building Embedded Conceptual Parsers. Unpublished Ph.D. Thesis, Northwestern University, 1994.
5. Fitzgerald, W. and Firby, R.J. The Dynamic Predictive Memory Architecture: Integrating language with task execution. Proceedings of IEEE Symposia on Intelligence and Systems. 1998. Washington, DC.
6. Flachsbart, J., Franklin, D., and Hammond, K. Improving human computer interaction in a classroom environment using computer vision. Proceedings of Intelligent User Interfaces. 2000. New Orleans, LA: ACM.
7. Horswill, I., Specialization of Perceptual Processes. Unpublished Ph.D. Thesis, Massachusetts Institute of Technology, 1993.
8. Johnson, M. Unification-based multimodal parsing. Proceedings of COLING-ACL 98. 1998. Montreal, Quebec: ACL Publications.
9. Johnson, M. and Bagalore, S. Finite-state multimodal understanding and parsing. Proceedings of COLING-2000. 2002. Saarbrücken, Germany: ACL Publications.
10. Martin, C.E., Case-based parsing and Micro-DMAP, in Inside Case-Based Reasoning, C.K. Riesbeck and R.C. Schank, Editors. 1989, Lawrence Erlbaum Associates: Hillsdale, NJ.
11. Martin, D.L., Cheyer, A.J., and Moran, D.B., The Open Agent Architecture: A framework for building distributed software systems. Applied Artificial Intelligence, 1999. 13: 91-128.
12. Moran, D.B., et al. Multimodal user interfaces in the Open Agent Architecture. Proceedings of Intelligent User Interfaces. 1997. Orlando, FL: ACM.
13. Oviatt, S., Ten myths of multimodal interaction. Communications of the ACM, 1999. 42(11): 74-81.
14. Schreckenghost, D.C., et al., Intelligent control of life support for space missions. IEEE Intelligent Systems, 2002. 17(5): 24-31.