# Modeling and Interpreting Multimodal Inputs: A Semantic Integration Approach

Minh Tue Vo and Alex Waibel

December 1997
CMU-CS-97-192

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

## Abstract

Modern user interfaces can take advantage of multiple input modalities such as speech, gestures, handwriting... to increase robustness and flexibility. The construction of such multimodal interfaces would be greatly facilitated by a unified framework that provides methods to characterize and interpret multimodal inputs. In this paper we describe a semantic model and a multimodal grammar structure for a broad class of multimodal applications. We also present a set of grammar-based Java tools that facilitate the construction of multimodal input processing modules, including a connectionist network for multimodal semantic integration.

This research was sponsored by the DARPA under Department of the Navy, Naval Research Office under grant number N00014-93-1-0806, and Project GENOA under grant number 97047-ISX/SOW-600066. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DARPA or the U.S. Government.

# 1. INTRODUCTION

Recent advances in speech recognition technology have created a great deal of renewed interest in alternative input modalities to replace or supplement the keyboard and mouse of traditional user interfaces. However, human communication does not consist solely of speech; we convey a wealth of information through other modalities such as gesturing, writing, drawing, facial expressions, gaze... We believe that the joint processing of such multimodal input sources can take advantage of the redundancy and complementary information across modalities to increase flexibility and robustness [1][7][8][12].

Our multimodal research has focused on building practical applications which support multiple input modalities [15][16][17][18]. In the course of implementing these applications, we have concluded that there is a need for methods to characterize multimodal user inputs, both syntactically and semantically. In the absence of extensive input data collected for a given task domain, an application developer may have to construct a model of what the user might say, write, draw... in order to implement a prototype of the application. Even if data is available to train speech recognizers for instance, the application developer may want to create a semantic model that describes how the application would respond to each set of multimodal stimuli drawn from the collected data. The usual approach for speech-enabled applications is to develop a grammar that characterize spoken inputs [19]. We need to generalize this notion to encompass more than just the speech modality.

Given a semantic model, the task of interpreting a multimodal input event consists of mapping the event to its corresponding semantic characterization in the model. One way to interpret a spoken utterance is to use a parser driven by a grammar to break the input utterance into semantically significant parts, each contributing to the overall interpretation. When this notion is generalized to multiple input modalities, we must also deal with the problem of aligning parts from different modalities if those parts need to be combined to make sense. Temporal alignment is an obvious approach that does not work in practice because it is difficult to constrain user input such that input segments that go together semantically always occur close to one another in time. We believe that semantic alignment is a more promising approach.

In this paper we describe a semantic model for a broad class of multimodal applications, as well as a multimodal grammar structure that embeds this semantic model in a context-free modeling language. We also present a set of grammar-based Java tools that facilitate the construction of input processing modules in multimodal applications. In addition to input modeling facilities, the tool set includes a connectionist-network-based parsing engine capable of segmenting and labeling multimodal input streams to ease the multimodal interpretation task. This flexible, trainable parsing network represents a multimodal interpretation approach based on the semantic alignment and integration of multiple input streams.

# 2. A MULTIMODAL SEMANTIC MODEL

One popular approach to speech understanding is to write a grammar that covers as many utterances as possible in the designated task domain. Given an input utterance to be interpreted, a parser attempts to match the utterance against grammar rules to break it down into components that carry semantic tags, which are assembled to form the interpretation of the input utterance. This approach is exemplified by the Air Travel Information Service (ATIS) system described in [19].

It is possible to employ the grammar used for interpretation as a user input model, i.e., a model that characterizes the kind of sentences we can expect from the users. However, we believe it is advantageous to put some separation between the notions of modeling and interpreting user input. An input model should characterize the most likely input combinations we may encounter, whereas an interpretation engine should maximize coverage and be flexible enough to handle even input combinations that may not be anticipated by the input model. The development of our multimodal semantic model was driven by the necessity of handling multiple input modalities and the parallel need for distinguishing the processes of modeling and interpreting multimodal inputs. This section describes the input modeling process; the interpretation process will be explored in Section 3.

A multimodal semantic model must answer two questions:

- What is the meaning assigned to a multimodal input event?
- How is this meaning derived by combining inputs from different modalities?

## 2.1. Meaning of Multimodal Input Events

In multimodal applications, the emphasis is on the way the system responds to user input. The user is trying to accomplish a task, and if the application carries out the right action then we are justified in saying that it has successfully interpreted whatever the user said, wrote, drew... In practical applications the available actions may also be characterized by *parameters* that could cause actions with the same name to have different effects. For instance, a **delete** action in a word processor would not be meaningful unless we specify what is to be deleted by supplying some kind of parameters. Thus we define the meaning of a multimodal input event as the *parameterized action* that the application should perform in response to the input event.

## 2.2. Combination of Multimodal Input Signals

A multimodal input event may be composed of input signals from one or more sources, such as digitized speech data and point coordinates from handwritten strokes. For each input modality we must decide when to start and stop recording a single input event. We also have to select a policy that determines when to group input events from different modalities into a combined multimodal input event. These design decisions may depend on the target multimodal application. In the applications we developed [15][16][17], we usually start recording input when the user starts speaking or drawing, and stop when no further input has been received within a predefined time-out interval. Input events from different modalities are grouped if they occurred close together in time, i.e., if their durations overlap or if one event starts within a time-out interval after another event ends. In the remainder of this discussion we will assume multimodal input events are collected and grouped according to some criteria similar to the above.

In deciding how to assign meaning to multimodal input events, inputs from different modalities may be associated with one another at many levels. At the lowest level the raw signals might be combined somehow before a meaning is derived from the aggregate signal. At an intermediate level the raw signals might be converted to a more convenient representation (such as a text string for speech input or a sequence of gesture shapes for pen input) before being combined and interpreted together. At a higher level the input from each modality might be assigned a partial interpretation, and these partial results could be merged in some way to form a joint interpretation. Combining inputs at the signal level is difficult at best and does not present any obvious way of characterizing semantics in a general, domain-independent manner. Combining partial interpretations is possible and practical [17]; in fact, the semantic representation employed in this process (as described in [17]) was the inspiration for the semantic model described here. We will show that it is possible to derive a semantic interpretation from the combination of intermediate symbolic input representations.

## 2.3. Action Frames and Parameter Slots

Our proposed model is applicable if the input from each modality can be represented as an *information stream*, i.e., the input consists of a sequence of tokens which may contribute information towards determining the output action and its parameters. We look at a multimodal input event as a set of parallel streams that can be aligned and jointly segmented such that each part of the segmented input influences part of the interpretation (see Figure 1). We call

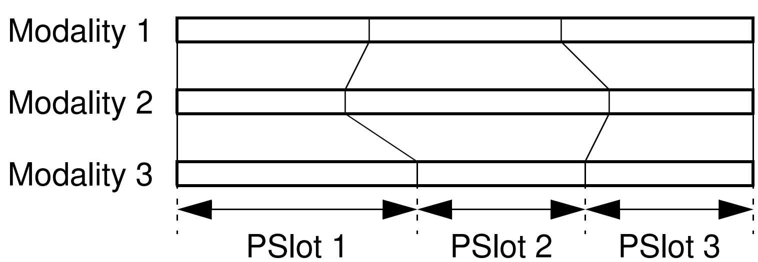



**FIGURE 1. Alignment and Joint Segmentation of Multimodal Inputs**

the overall interpretation an *action frame* since it specifies the action to be carried out in response to the input. Each part of the segmented input is a *parameter slot* that specifies one action parameter. The input segments in each parameter slot should contain enough information to determine the value of the corresponding parameter.

Consider the following illustrative example. Suppose we have a map navigation system that allows the user to ask for information by speaking and drawing on the screen. The user might say “How far is it from here to there?” while drawing an arrow between two points on the displayed map. The speech input stream consists of the words in the utterance whereas the pen input stream contains a pair of *arrow_start* and *arrow_end* tokens. The interpretation of this input combination is a *QueryDistance* action frame containing a *QueryDistanceSource* parameter slot followed by a *QueryDistanceDestination* parameter slot. The input streams are segmented and aligned as follows:

| | | |
|---|---|---|
| *Speech:* | `how far is it from here` | `to there` |
| *Pen:* | `arrow_start` | `arrow_end` |
| | *QueryDistanceSource* | *QueryDistanceDestination* |

If the destination point is somewhere outside the displayed area, the user might say “How far is it from here to Philadelphia?” and circle the starting point instead. In this case the input segmentation becomes

| | | |
|---|---|---|
| *Speech:* | `how far is it from here` | `to philadelphia` |
| *Pen:* | `circle` | |
| | *QueryDistanceSource* | *QueryDistanceDestination* |

For the utterance “How far is it from Pittsburgh to Philadelphia?” the parameter slots would consist of speech segments only.

Note that it is not necessary to have a segmentation of the form

| | |
|---|---|
| *QueryDistanceAction:* | how far is it |
| *QueryDistanceSource:* | from <somewhere> |
| *QueryDistanceDestination:* | to <somewhere> |

since the action is implicitly specified by the sequence of parameter slots. The *QueryDistanceAction* segment does not specify any action parameter and thus does not qualify as a parameter slot. It is possible to redefine our semantic model to include such segments but we have not found this to be advantageous in practice. In general it is only necessary to break the input into segments that actually contain information needed to extract action parameters.

## 2.4. Semantic vs. Temporal Alignment

Note that the alignment and segmentation process described above makes no mention of any temporal constraints. The various parts of the input streams are aligned purely based on their semantic contents. This way there is no time-alignment constraints on the way the user interacts with the system. It is also possible to force some kind of temporal alignment based on time-stamps assigned to input tokens. However, if this is done without regards to semantics, we may end up with cross-modal combinations that do not make sense, or we may have to impose constraints on the way the user interacts with the system (e.g., a gesture must be drawn right at the time certain words are spoken). This would severely compromise the flexibility of the user interface, which goes against the reason we wanted multimodal support in the first place. If temporal alignment is used at all, it should be used only to constrain the semantic alignment by restricting the alignment boundaries according to time-stamps. This would be properly part of the interpretation process and thus irrelevant in the semantic modeling stage.

# 3. MULTIMODAL INPUT INTEGRATION

Given the semantic model described above, the first step in interpreting a multimodal input event is to find an alignment and joint segmentation of the input streams. This *input integration* process produces input segments labeled as parameter slots, which can be postprocessed to extract the actual action parameters. The parameter-extraction postprocessing is necessarily domain-dependent, but it is possible to devise domain-independent algorithms to integrate multimodal input streams. We have implemented such an algorithm based on a connectionist network.

## 3.1. Basic Mutual Information Network

Suppose we have a sequence of input tokens $t_m$, $m$=1...$M$, that is to be associated with one of several output classes $c_n$, $n$=1...$N$. It is reasonable to select the *maximum a posteriori* (MAP) hypothesis, or the output class having the greatest *a posteriori* probability given the input:

$$c_{MAP} = \underset{c_n}{\operatorname{argmax}}\ P(c_n | t_1 t_2 \ldots t_M)$$

If we make the simplifying assumption that the input tokens are independent as well as conditionally independent given the target output, the above equation can be rewritten as

$$\begin{aligned} c_{MAP} &= \underset{c_n}{\operatorname{argmax}}\ P(c_n | t_1 t_2 \ldots t_M) \\ &= \underset{c_n}{\operatorname{argmax}}\ \frac{P(t_1 t_2 \ldots t_M | c_n) P(c_n)}{P(t_1 t_2 \ldots t_M)} \\ &= \underset{c_n}{\operatorname{argmax}}\ P(c_n) \prod_m \frac{P(t_m | c_n)}{P(t_m)} \end{aligned}$$

This is essentially a naive Bayes classifier [10]. Since the logarithm function is monotonically increasing, we have

$$\begin{aligned} c_{MAP} &= \underset{c_n}{\operatorname{argmax}} \left( \log P(c_n) + \sum_m \log \frac{P(t_m | c_n)}{P(t_m)} \right) \\ &= \underset{c_n}{\operatorname{argmax}} \left( \log P(c_n) + \sum_m I(t_m, c_n) \right) \end{aligned}$$

where $I(t_m, c_n) = \log[P(t_m | c_n) / P(t_m)]$ is the *mutual information* of input $t_m$ and output $c_n$.

The right hand side of the above equation can be realized by a connectionist network. The activation of output unit $c_n$ is a weighted sum of input unit activations, where an input unit $t_m$ has activation 1 if the token $t_m$ is present in the input sequence, and 0 otherwise. The connection weight from input $t_m$ to output $c_n$ is $w_{mn} = I(t_m, c_n)$. There is also a bias connection with weight $w_n = \log P(c_n)$. The output with the highest activation is selected as the most probable hypothesis given the input sequence. This network architecture was first proposed by Gorin et al. [3].

Although the simplifying independence assumption does not usually hold in practice, this mutual information network has been shown to learn input-output associations quite successfully [3][9][14]. The input independence assumption implies that classification does not depend on the order of the input tokens. To take into account the fact that adjacent input tokens sometimes form phrases or sentence fragments having significant information contents, we can introduce higher-order input units which are activated when particular token sequences occur [4]. Since the number of high-order input units can explode quite rapidly, we prune away units that are not useful for classification. This can be done using a measure called *salience* which is indicative of input relevancy with respect to classification [4].

The mutual information network architecture is depicted in Figure 2.

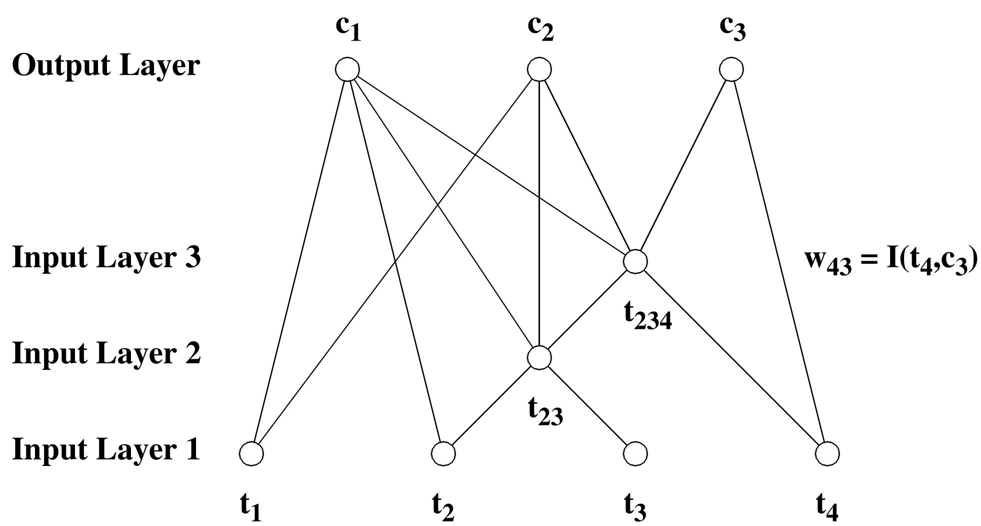



**FIGURE 2. Mutual Information Network Architecture**

## 3.2. The Multi-State Mutual Information Network

The basic mutual information network described above assigns a single label to each input sequence. Since our goal is to produce a sequence of labels (i.e., parameter slots) by segmenting the input, we need to extend the network to handle this case.

The output activations in the basic mutual information network can be regarded as estimates of the *a posteriori* probabilities $P(c_n|t_1 t_2 \ldots t_m)$. We want to develop a similar score for input segmentations and label assignments. Assume we have an input segmentation $s_1 s_2 ... s_k$ where each $s_i$ is a group of adjacent input tokens (in the general case the group may consist of subgroups from different modalities). To evaluate a possible label assignment $c_1c_2...c_k$, we want to estimate

$$P(c_1c_2\ldots c_k | s_1s_2\ldots s_k) = P(c_k | s_1s_2\ldots s_k \wedge c_1c_2\ldots c_{k-1})P(c_1c_2\ldots c_{k-1} | s_1s_2\ldots s_k)$$

We can estimate $P(c_k | s_1s_2\ldots s_k \wedge c_1c_2\ldots c_{k-1})$ as follows. Assuming that the first $k$-1 input segments have been correctly labeled as $c_1c_2...c_{k-1}$, we replace them by their respective labels to get the equivalent input sequence $c_1c_2...c_{k-1}s_k$. The *a posteriori* probability $P(c_k | c_1c_2\ldots c_{k-1}s_k)$ with respect to this new input sequence is an estimate of $P(c_k | s_1s_2\ldots s_k \wedge c_1c_2\ldots c_{k-1})$. The advantage of this transformation is that $P(c_k | c_1c_2\ldots c_{k-1}s_k)$ can be estimated by feeding the transformed input sequence $c_1c_2...c_{k-1}s_k$ to a mutual information network.

We also make the simplifying assumption that the partial sequence $c_1c_2...c_{k-1}$ only depends on the input tokens in the first $k$-1 segments; thus

$$P(c_1c_2\ldots c_{k-1} | s_1s_2\ldots s_k) = P(c_1c_2\ldots c_{k-1} | s_1s_2\ldots s_{k-1})$$

Combining the above transformations yields

$$P(c_1c_2\ldots c_k | s_1s_2\ldots s_k) \approx P(c_k | c_1c_2\ldots c_{k-1}s_k)P(c_1c_2\ldots c_{k-1} | s_1s_2\ldots s_{k-1})$$

Let $\hat{P}(c_1c_2\ldots c_k | s_1s_2\ldots s_k)$ denote an estimate of $P(c_1c_2\ldots c_k | s_1s_2\ldots s_k)$ that satisfies the above recurrence relation exactly. We can use recursive decomposition to obtain

$$\begin{aligned}
\hat{P}(c_1\ldots c_k | s_1\ldots s_k) &= P(c_k | c_1\ldots c_{k-1}s_k)\hat{P}(c_1\ldots c_{k-1} | s_1\ldots s_{k-1}) \\
&= P(c_k | c_1\ldots c_{k-1}s_k)P(c_{k-1} | c_1\ldots c_{k-2}s_{k-1})\hat{P}(c_1\ldots c_{k-2} | s_1\ldots s_{k-2}) \\
&= \ldots \\
&= P(c_k | c_1\ldots c_{k-1}s_k)P(c_{k-1} | c_1\ldots c_{k-2}s_{k-1})\ldots P(c_1 | s_1)
\end{aligned}$$

Taking the logarithm of both sides yields

$$\log\hat{P}(c_1\ldots c_k | s_1\ldots s_k) = \log P(c_1 | s_1) + \log P(c_2 | c_1s_2) + \ldots + \log P(c_k | c_1\ldots c_{k-1}s_k)$$

Each term in the above sum is estimated by an output activation in the mutual information network, hence the sum can be interpreted as the score of a path that goes through the segment labels $c_1c_2...c_k$ in order, as illustrated in Figure 3. Using a dynamic programming algorithm similar to the Viterbi search or Dynamic Time Warping in speech recognizers [11], we can find an input segmentation and a corresponding label assignment that maximize the path score. Multiple input modalities are accommodated by implementing the path score maximization algorithm over more than one input dimension, where each dimension extends along one input stream. Figure 4 shows a path over two input dimensions.

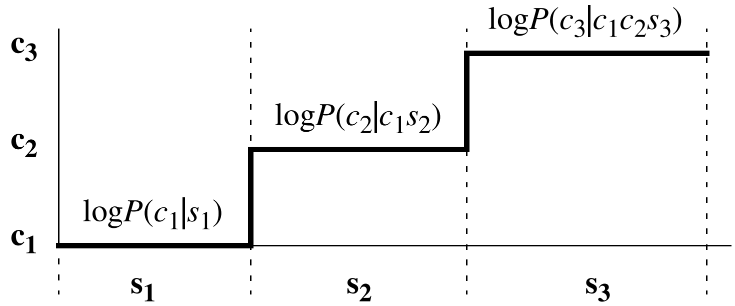


**FIGURE 3. Path Score of Input Segmentation and Labeling**

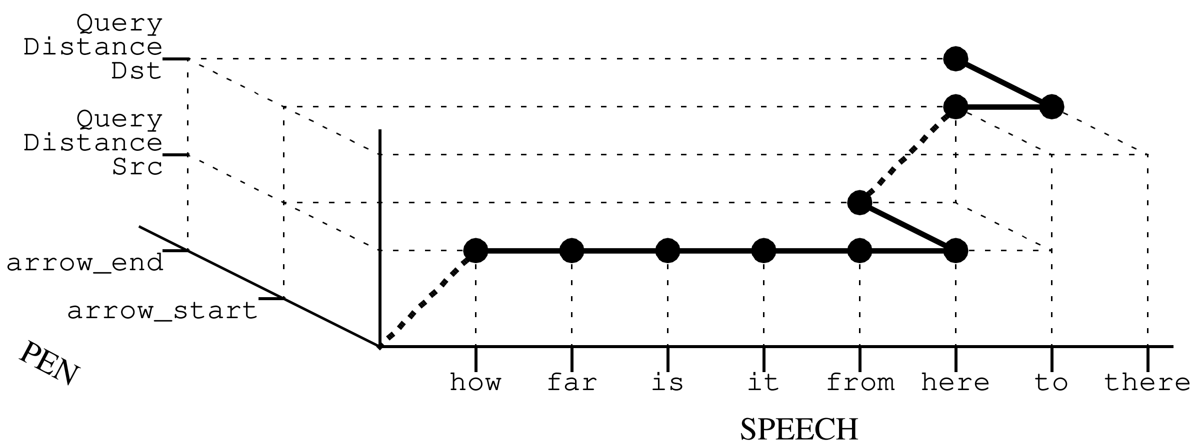


**FIGURE 4. Output Path Over Multidimensional Inputs**

The above path score maximization procedure effectively adds an extra layer on top of the basic mutual information network (see Figure 5). Each output unit of the mutual information network constitutes an output state, and the top layer produces the best sequence of states that fits the input, in a manner reminiscent of the Multi-State Time Delay Neural Network [5][6]. We call this architecture the *multi-state mutual information network*, or MS-MIN.

The current implementation of the MS-MIN produces a parameter slot sequence in only one action frame for each multimodal input event. However, it would be straightforward to modify the dynamic programming algorithm in the state layer to produce a path through more than one action frame. Such an enhanced network would be able to parse a multimodal input event that maps to a sequence of two or more actions. This could accommodate input sentences such as “Cancel the meeting with Fred and schedule a meeting with John” in a multimodal appointment scheduler that supports *CancelMeeting* and *ScheduleMeeting* actions.

## 3.3. Training the MS-MIN

In backpropagation neural networks, the connection weights are incrementally adjusted during training by a gradient descent algorithm to minimize a classification error function [13]. In contrast, the weights in a mutual information network can be computed directly from occur-

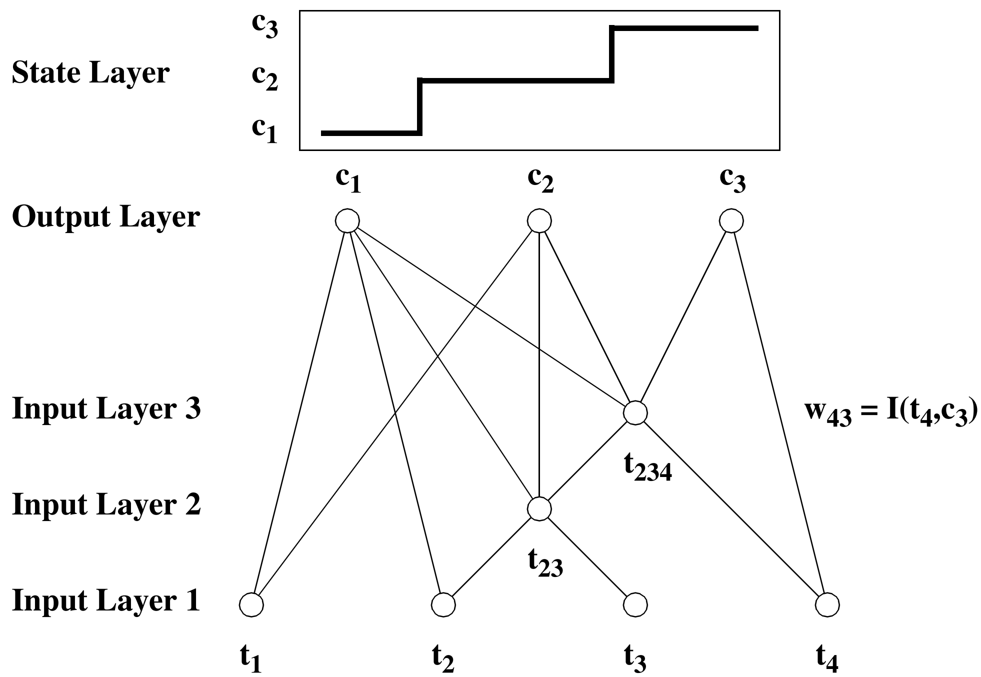



**FIGURE 5. Multi-State Mutual Information Network Architecture**

rence probabilities observed in the training data. These probabilities can be estimated by a simple counting procedure [3]. In Section 5.5 we will describe how the weights can be automatically generated from a grammar-based input model. Thus our network architecture enjoys a definite advantage over backpropagation networks since the training time is drastically reduced or eliminated altogether.

In principle the mutual information network is capable of learning incrementally during actual use, as demonstrated by Gorin et al. [3]. The MS-MIN inherits this capability; however, we need to conduct more experiments to determine to what degree this is true in practice, when the task domains are complex and a lot of training data must be used to achieve adequate coverage.

# 4. GRAMMAR-BASED MULTIMODAL INPUT MODELING

The semantic model described in Section 2 can serve as a basis for a more comprehensive multimodal input modeling language. After deciding what action frames to include and what parameter slots to put in each action frame, we can use a context-free grammar to specify the various possible things the user might say, write, draw... and their corresponding segmentation into parameter slots. The intent of this grammar structure is to capture syntactic elements in relation to their semantics. Our parameter slot segmentation model allows us to do this across modalities. In addition, we can incorporate a probabilistic model that specifies the likelihood of each input combination modeled by the grammar.

## 4.1. Grammar Components

Our form of multimodal grammars consists of two basic entities: *nodes* and *sequences*. A node may be a *literal* or a group of related constructs forming a single entity. A sequence consists of a series of nodes and represents one of possibly many alternatives within a complex node. Each sequence has a weight specifying the relative likelihood that the sequence will be chosen within the parent node.

There are six types of nodes:

- a *Toplevel* represents the entire grammar and contains one or more sequences, each of which contains exactly one AFrame;
- an *AFrame* represents an action frame and contains one or more sequences, each of which consists of one or more PSlot;
- a *PSlot* represents a parameter slot and contains one or more UnimodalNodes (at most one for each modality);
- a *UnimodalNode* specifies a sub-grammar for one single modality and has the same structure as a *NonTerm*, with the addition of a label specifying the modality;
- a *NonTerm* is a non-terminal node consisting of one or more sequences, each of which contains one or more NonTerm or Literal;
- a *Literal* is a terminal node containing a text string.

Figure 6 shows a simple grammar that illustrates all the basic building blocks. The grammar structure described here is equivalent to a recursive transition network formulation [19] which can represent any context-free grammar.

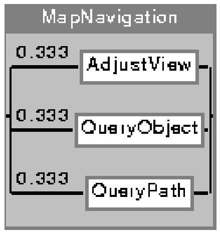
There exist syntactic descriptions of context-free grammars that include notations for optional or repeating elements. These are only "syntactic sugar" notations that are convenient but not necessary, since they can be expressed using only the basic notations. For example, a node in our multimodal grammar formulation can be made optional by replacing it with a NonTerm containing an empty sequence and a second sequence that includes the original node. Similarly, a repeating node is equivalent to a NonTerm that recursively references itself; for instance,

```
N ::= a | aN
```

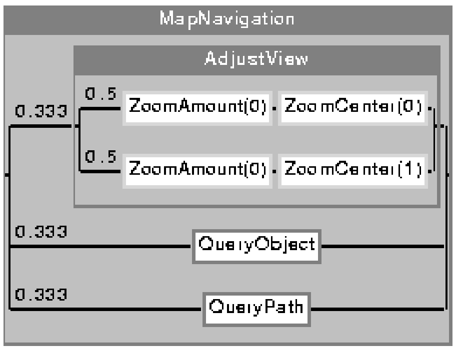
represents any number of repetitions of the letter `a`. The current implementation of our multimodal grammar structure does not include shortcut notations.

## 4.2. Grammar Implementation

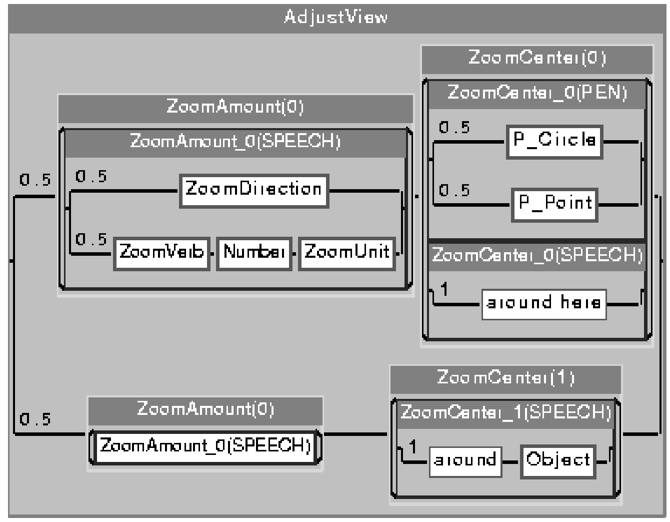
The grammar structure described above can be easily implemented using the object-oriented formalism in Java. Figure 7 shows the resulting class hierarchy. This design permits a great deal of code reuse; for instance, the code for manipulating sequences is shared among Toplevel, AFrame, UnimodalNode, and NonTerm. Polymorphism greatly simplifies the code; thus Sequence only has to deal with Node instead of all its different subclasses. All the com-



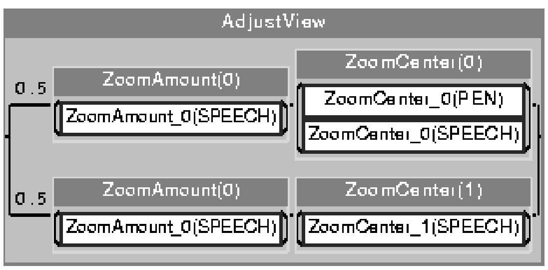

a) Toplevel *“MapNavigation”*



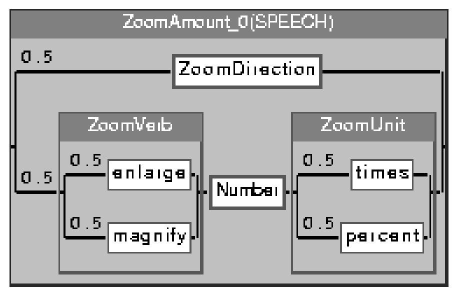

b) AFrame *“AdjustView”*




d) UnimodalNodes in *“Zoom”* PSlots




c) PSlots in AFrame *“AdjustView”*




e) NonTerms and Literals

**FIGURE 6. Multimodal Grammar Structure**

mon functionality is factored out and put into a common root class, GObject, which presents a uniform interface to all external entities that have to manipulate grammar objects.

## 4.3. Grammar Traversal

Most of the grammar-based tools described in later sections do their work by traversing the grammar graph structure and performing some operation on each component. It we were to implement this as a polymorphic recursive method on GObject, it would be necessary to add one method for each operation we want to support. We chose instead to apply the Visitor design pattern [2]. The Visitor pattern is ideal for this because the number of grammar object types is fixed, whereas the number of possible operations is unlimited. The list of operations we need to support includes loading and saving grammar objects as well as generating various types of information from grammars: language models, random samples, preprocessors, integration networks...

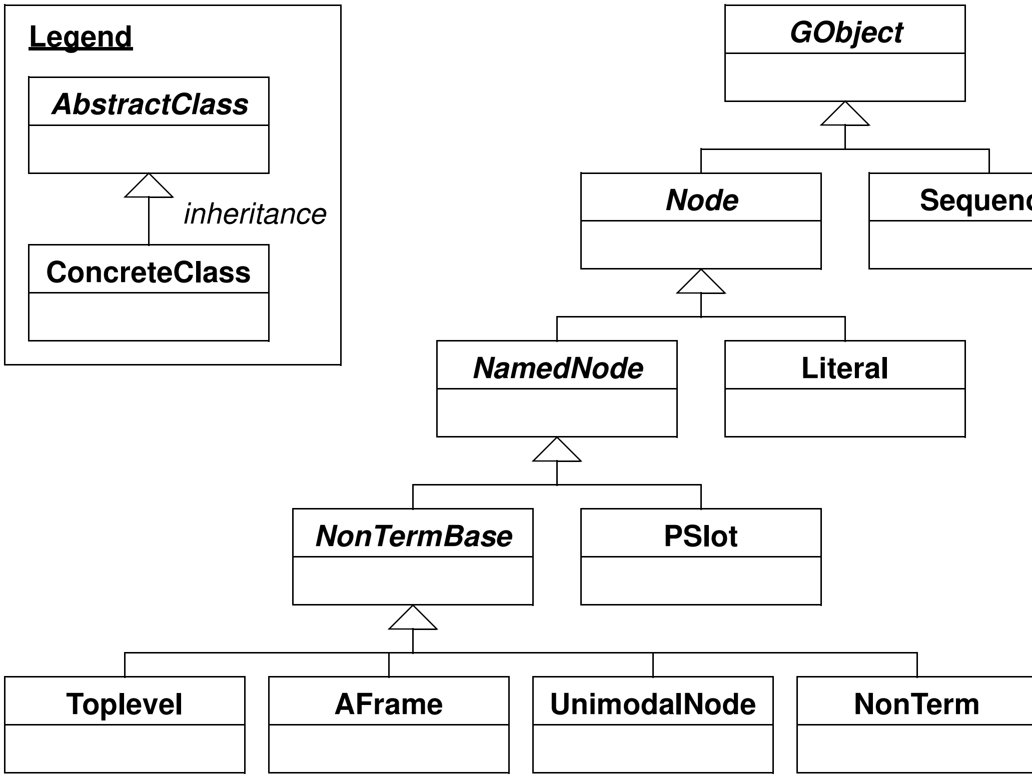



**FIGURE 7. Class Hierarchy For Multimodal Grammar Components**

In the Visitor pattern, the only necessary modification to the target object hierarchy (GObject and its subclasses in this case) is a polymorphic method that accepts an instance of an abstract *Visitor* class. Using a well-know object-oriented technique called *double-dispatch*, the same method invocation ends up calling different functions depending on the actual types of two objects: the target of the visit and the Visitor instance. Different operations are implemented by different subclasses of the Visitor base class. The Visitor pattern forms the basis for the implementation of many algorithms that work on multimodal grammars, without requiring *ad hoc* modifications of the GObject class hierarchy.

# 5. MULTIMODAL GRAMMAR TOOLS

This section describes a collection of grammar-based tools that are useful for user input modeling in multimodal applications. For the most part the tools are written in Java to maximize platform independence and to make it possible to deploy them on the World Wide Web as applets running inside Web browsers.

## 5.1. Visual Grammar Designer

Traditional grammars are usually represented textually in some descriptive language such as Backus-Naur Form (BNF) or Phoenix [19]. It is rather difficult to follow these textual descriptions at a glance and keep track of grammar rewrites, especially as the size of the grammar increases. A graphical display that represents the grammar components visually as in Figure 6 makes it much easier to understand the grammar by visual examination. Furthermore, creating or editing a large grammar in textual form sometimes requires the skills of a computer programmer; in contrast, designing a grammar visually by dragging and dropping graphical components is much more intuitive and requires less training. We have implemented an object-oriented, drag-and-drop grammar editor that employs exactly this paradigm.

### 5.1.1 Graphical Display of Grammar Components

In our implementation the graphical user interface (GUI) elements are cleanly separated from the underlying grammar representation using the Observable/Observer design pattern [2], similar to the model-view approach in the Smalltalk programming environment. Each GObject has one or more associated "observers" or "views", represented by subclasses of a GObjectView root class (Figure 6 shows some views captured from a computer display). The views know how to update themselves whenever the "observable" or "model" object broadcasts a change in its data. External entities do not need to know about views; when they manipulate the underlying grammar structure the screen will be automatically updated.

The views for Node objects can be expanded to show the internal structure or collapsed to display only the node labels. This way the overall grammar structure can be grasped instantly while still allowing for detailed examination of any section, down to the level of Literals. When connected nodes are expanded, phrases modeled by the grammar can be easily read off the display as in part d) and e) of Figure 6.

### 5.1.2 Drag-And-Drop Editing

The view objects provide convenient handles to manipulate grammar entities visually. It is relatively easy to implement a drag-and-drop GUI in which the handles may be moved around by moving the mouse while holding down a button ("dragging"), and inserted into other objects by letting go of the mouse button when the cursor is over the desired location ("dropping"). This kind of direct manipulation is ubiquitous in modern GUIs and familiar to most computer users. It permits the rapid construction of a grammar with convenient, continuous visualization of various grammar parts and their relationships.

Our Multimodal Grammar Designer program supports exactly this kind of grammar construction and editing. Figure 8 shows a screen capture of the Designer. The main window shows a graphical grammar display. Views of different node types are color coded so that the type of any grammar entity can be grasped instantly. Expanded and collapsed states of grammar nodes are also distinguishable visually. Double-clicking on any node expands the node or collapses it if the node is already expanded. Clicking with the right mouse button displays a context-sensitive popup menu tailored to the object beneath the cursor. Certain menu items are common across all objects, such as a "Properties" menu item that pops up a dialog allowing the user to modify object attributes.

On the right side of the Designer window is a palette of grammar object "prototypes" that can be used to construct a grammar visually. The prototypes are also color coded and grouped by

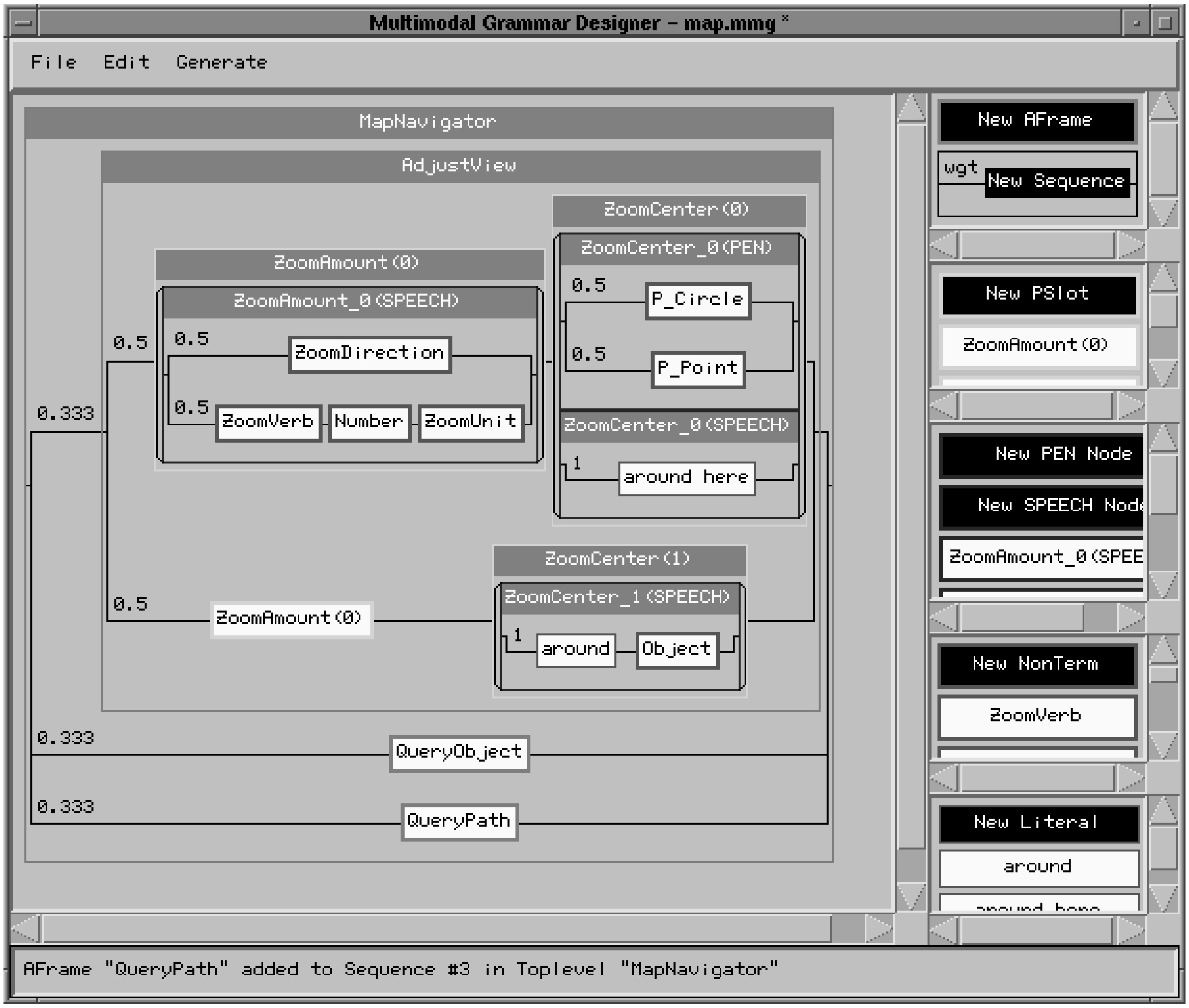



**FIGURE 8. The Multimodal Grammar Designer**

object types. There are prototypes that allow the creation of new objects, while the rest correspond to existing objects and permit their reuse in difference places. The prototypes can be "grasped" and dragged with the mouse. The cursor changes shape when a dragged prototype passes over potential drop sites to indicate whether a drop at that location will be allowed. For example, AFrame can only be dropped into Toplevel, PSlot can only be dropped in AFrame, and so on. Visual feedback in the form of a dashed line indicates where the new object (or reference to an existing object) would be inserted in the drop target. When a newly created object is dropped, a Properties dialog pops up to allow the user to change object attributes from the default values. After the object has been successfully added to the grammar, this dialog can be accessed again at any moment by right-clicking and choosing "Properties" from the context menu, as described above.

## 5.2. Random Sample Generator

Components of multimodal applications usually need to be validated by computing certain evaluation functions over a set of test input data. If little or no actual data is available, as may be the case during the construction of an application prototype, it is still possible to obtain a set of artificial data that reflects a user input model constructed by the application developer. This kind of model is precisely what multimodal grammars are suppose to encode. Using the probability distribution represented by sequence weights, we can generate random multimodal input samples that follow such a distribution.

The sample generator is a Visitor subclass (see Section 4.3) that traverses the grammar graph and selects sequences at random. For each non-terminal node, the weight of each sequence is divided by the total weights of all sequences in the node to produce the selection probability. The literal tokens from selected sequences are concatenated to form a token stream for each modality. The output is encoded in a format that retains the parameter slot segmentation information in case this may be useful to the evaluation procedures that process the generated data.

## 5.3. N-gram Language Model Generator

A large vocabulary, continuous speech recognizer is usually customized for a particular task domain using a statistical language model, normally a bigram or trigram model. A statistical language model helps guide the search for the correct speech-to-text mapping by predicting the likelihood of encountering a word based on preceding words. A bigram model uses a single preceding word whereas a trigram model uses two preceding words.

N-gram language models are normally generated by counting trigrams (sequences of 3 words) in a training corpus and computing bigram/trigram probabilities from the trigram count table. Generating the language model directly from a grammar means performing the equivalent of generating a very large number of random samples from the grammar to form the training corpus. Since generating random samples involves traversing the grammar structure, we can count the trigrams during the traversal without storing any samples. Using the sequence weight distribution built into the grammar, we can compute an exact trigram weight for each trigram permitted by the grammar. If we imagined generating an enormous number of random samples and dividing the number of times a certain trigram occurs by the number of samples, the limit of this ratio as the corpus size tends to infinity would be exactly the trigram weight. The language model generated from these trigram weights should be more accurate than one generated from any random corpus, no matter how large the corpus size is.

### 5.3.1 Basic N-gram Counting Algorithm

It is possible to compute a trigram weight table for each grammar node by recursively combining the trigram weight tables of the node's components. Since each node is composed of weighted alternative sequences, and each sequence is a series of nodes, we define three basic operations on trigram weight tables:

1. *Scaling*. This operation multiplies all trigram weights in a table by a scaling factor representing the probability that a certain sequence in a node would be randomly selected if we were to generate random samples from the grammar. This probability is simply the weight of the sequence divided by the total weight of all sequences in the node.

2. *Merge*. This operation takes the trigram weight tables for two sequences in a node and combines them to form a new trigram weight table that contains all the trigrams in the original tables. Weights for trigrams that occur in both component tables are added because if we were to generate random samples from the grammar, the trigram counts for samples that contain either of the two alternative sequences would accumulate additively.
3. *Concatenation*. This operation produces the trigram weight table for two nodes in a series by concatenating trigrams from the two nodes to form new trigrams. Two trigrams can be concatenated if one occurs at the end of the first node and the other occurs at the beginning of the second node. The weights of the concatenated trigrams are multiplied together.

The language model generator is a Visitor subclass (see Section 4.3) that does the following for each object it visits:

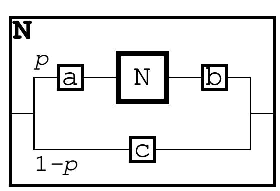
- if it's a Literal, count the trigrams that occur in the Literal's text;
- if it's a PSlot, visit the UnimodalNode for the designated modality (usually speech);
- if it's any other non-terminal node, visit each sequence of the node in turn and merge the resulting trigram weight tables;
- if it's a sequence, visit each node in the sequence and concatenate the resulting trigram weight tables, then scale the result by the normalized sequence weight.

Nodes can be reused in different parts of the grammar so we cache the computed trigram weight table for each node and do not recompute it if the node is visited again.

### 5.3.2 Dealing With Recursive Grammar References

The above algorithm works if the grammar is finite-state, but as soon as there is a recursive reference to a node, the algorithm goes into an infinite loop. The only practical way of dealing with this is to stop the recursion at some depth. In that case the computed trigram weights are no longer exact, but if the recursion were allowed to go deep enough, the multiplicative effects of scaling would reduce the probabilities to such small values that the depth-limited computation could produce results with any desired accuracy.

Consider the following grammar node:




This node represents sentences of the form $\mathtt{a}^n\mathtt{cb}^n$ for all integers $n \geq 0$. The possible trigrams are: <s><s>a, <s><s>c, <s>aa, <s>ac, aaa, aac, acb, cbb, bbb, cb</s>, bb</s>, <s>c</s> where <s> and </s> denote the beginning- and end-of-sentence markers. Imagining that each sentence starts with two beginning markers and ends with one end marker makes trigram counting more regular. It is easy to see that the trigram weight for <s><s>a must be $p$ and the weight for <s><s>c must be $1–p$, for example, but the trigram weight for aaa is distributed across all sentences $\mathtt{a}^n\mathtt{cb}^n$ for $n \geq 3$. For a specific $n$ the trigram `aaa` occurs $n-2$ times in the sentence, and the sentence has an occurrence probability of $p^n(1-p)$, hence the exact trigram weight is the sum of the infinite series

$$w = \sum_{n=3}^{\infty} (n-2)p^n(1-p) = \frac{p^3}{1-p}$$

If we stop the recursive expansion at depth $n_{max}$, we get the partial sum for $n=3...n_{max}$. One more level of expansion would add $(n_{max}-1)p^{n_{max}+1}(1-p)$ to the partial sum. We can stop if this new contribution divided by the new partial sum is smaller than an accuracy threshold, say $10^{-6}$. In general we maintain a running product of normalized sequence weights to compute the sentence probability and estimate the contribution of a recursive expansion to the trigram weights.

The above heuristic will not break the infinite loop if the recursion does not "bottom out", e.g, if the above grammar has no `c` but only the single sequence `aNb` with unit probability. In this case the trigram weights are not well defined anyway, so we supplement the stopping heuristic by imposing an absolute maximum recursion depth and raising an exception if the recursion unwinds completely without producing any trigrams.

### 5.3.3 Computing N-gram Probabilities

A trigram language model contains unigram, bigram, and trigram penalties, which are $\log_{10}$ of probabilities. Given a trigram count table, the various n-gram probabilities can be computed as follows:

$$P_{unigram}(a_n) = \frac{count(a_n)}{\sum_i count(a_i)}$$

$$P_{bigram}(a_m a_n) = \frac{count(a_m a_n)}{\sum_i count(a_m a_i)}$$

$$P_{trigram}(a_m a_n a_p) = \frac{count(a_m a_n a_p)}{\sum_i count(a_m a_n a_i)}$$

Since the n-gram weights computed by traversing the grammar are equivalent to the ratios of n-gram counts to the training corpus size, the weights can be directly substituted for the counts in the above equations.

Each unigram also has a back-off value used to estimate the probabilities of bigrams that did not occur in the training corpus. Similarly unknown trigram probabilities are estimated from bigram back-off values. This back-off scheme smooths out the n-gram distributions and lets the speech recognizer accept slight variations of sentences permitted by the grammar. We compute the back-off values using an absolute discount scheme in which a fixed discount (usually 0.5) is subtracted from each n-gram count to form a count for the unseen n-grams. Since this operation requires an actual count, we have to supply an *equivalent corpus size* which is then multiplied with the n-gram weights to produce the equivalent counts. A reasonable equivalent corpus size can be automatically computed to give the smallest weight an equivalent count of 1, i.e., to produce an equivalent training corpus in which each trigram allowed by the grammar appears at least once.

## 5.4. Input Preprocessor Generator

The input integration process described in Section 3 produces a labeled segmentation of multimodal input streams that breaks the input into a sequence of parameter slots. Actual parameter values still have to be extracted from the parameter slots in a postprocessing step. This could be much simplified if the most basic concepts in the input domain were represented by equivalent classes rather than simple words. For instance, if a parameter slot may contain a number that must be extracted, all the input phrases that represent a number (“one”, “two”, “twenty three” etc.) could be preprocessed and converted to a single *NUMBER* token with an attached data packet containing the actual number phrase. The advantages of this preprocessing are twofold:

- the integration network described in Section 3.2 can achieve higher accuracies because it only has to learn associations for the *NUMBER* token instead of all the possible number phrases;
- the parameter extraction postprocessing can readily identify which parts of the segmented input contain relevant, extractable information.

The preprocessing step can be implemented by a simple state-machine-based parser that identifies particular word groups in the target input stream (this is usually the speech stream but the same algorithm works for any modality that needs preprocessing). In the multimodal grammar we can mark appropriate NonTerm nodes as “parseable”, e.g., a *NUMBER* node that contains a sub-grammar for number phrases. Out toolkit includes a Visitor subclass (see Section 4.3) that functions as a grammar compiler and traverses the grammar structure to generate a state-machine matcher for each parseable node. The resulting collection of state machines can be used to match fragments of the input stream and convert them to tokens tagged as carrying parameter information.

The degree of preprocessing may vary depending on the task domain, but there is a trade-off between complexity and flexibility. If the preprocessing identifies high-level concepts, the complexity of the input space is reduced at the detriment of flexibility since input fragments must match the sub-grammars for the target concepts exactly. A good rule of thumb is to parse only the most basic concepts of the task domain needed for parameter extraction, e.g., *DAYS_OF_THE_WEEK* in an appointment scheduling task or *CITY_NAME* in a map navigation task. The job of matching higher-level constructs in a flexible manner should be left to the integration network which can be trained from actual user input.

### 5.5. Integration Network Generator

The multi-state mutual information network described in Section 3.2 has connection weights that can be computed from input-output occurrence probabilities in a training corpus. As with n-gram language models (see Section 5.3) it is possible to compute these probabilities directly by traversing the grammar structure, without generating any input sample. This algorithm is implemented in a Visitor subclass (see Section 4.3) that recursively computes count tables similar to the n-gram tables described in Section 5.3. In one experiment, this network generator took 2 minutes to produce a network which would have required the equivalent of 6 million training examples and more than 11 hours of training time to cover all possible input combinations.

### 5.6. Interpretation Code Generator

An integration network such as one produced by the network generator described above only segments and labels the input to identify the action frame and parameter slots. As explained in Section 5.4, the actual parameter values have to be extracted in a postprocessing phase to complete the multimodal interpretation process. Some part of this postprocessing must necessarily be domain-dependent, but the rest usually consists of repetitive code that branches based on the names of action frames, parameter slots, or preprocessed concept nodes. Much of this domain-independent code can be automatically generated from the grammar structure. Another Visitor subclass (see Section 4.3) accomplishes this by traversing the grammar and writing out template code customized by the grammar context at each node.

The output of the code generator is a Java class that contains methods to identify the interpretation context (e.g., the current action frame and parameter slot) and branch to the appropriate postprocessing code. The application developer only has to fill in the domain-dependent parts of the postprocessing template. To avoid losing the modifications if the template code is regenerated, the class produced by the code generator is usually retained unchanged and the modifications are put in a derived class by overriding the appropriate methods.

## 6. CONCLUSION

The multimodal semantic model and grammar modeling language presented in this paper are useful for characterizing user inputs that comprise multiple input channels. Our models represent the meaning of multimodal inputs as parameterized actions to be carried out by the target application in response to the inputs. Multiple input streams are integrated by an alignment and joint segmentation process that produces parameter slots in an action frame.

We described a multimodal integration algorithm based on a multi-state mutual information network that can be trained from examples or directly constructed from a multimodal grammar. The MS-MIN assigns scores to output nodes using mutual information weights from input nodes and produces an optimal segmentation and labeling of the input streams using dynamic programming. Under suitable assumptions, this can be interpreted as a maximum *a posteriori* decision.

Our multimodal grammar structure is easily implemented in Java and lends itself to a visual drag-and-drop construction paradigm that requires little or no programming skills on the part of multimodal application developers. We also produced a set of tools based on this grammar implementation to automate several steps in the construction of multimodal input processing modules. From a multimodal grammar that models user inputs for a given task domain, the tools can generate random input samples, a statistical language model for speech recognition, input preprocessors to parse macro concepts from the raw input tokens, an MS-MIN semantic integration network, and a postprocessor skeleton for parameter extraction. These components greatly facilitate the construction of multimodal applications, especially in the prototype stage when little or no actual user data is available.

# 7. ACKNOWLEDGEMENTS

We would like to thank Jie Yang for helpful insights and comments on this manuscript, Robert Malkin for testing the visual grammar editor and providing bug reports, and Klaus Ries for explaining the n-gram calculation.

# 8. REFERENCES

[1] Ando, H., Kitahara, Y., and Hataoka, N., "Evaluation of Multimodal Interface Using Spoken Language and Pointing Gesture On Interior Design System," *Proc. ICSLP'94* (Yokohama, Japan, Sept. 1994), Vol. 2, pp. 567-570.

[2] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[3] Gorin, A.L., Levinson, S., Gertner, A., and Goldman, E., "Adaptive Acquisition of Language," *Computer, Speech and Language*, Vol. 5, No. 2, April 1991, pp. 101-132.

[4] Gorin, A.L., "On Automated Language Acquisition," *J. Acoust. Soc. Am.*, Vol. 97, No. 6, June 1995, pp. 3441-3461.

[5] Haffner, P., Franzini, M., and Waibel, A., "Integrating Time Alignment and Neural Networks for High Performance Continuous Speech Recognition," *Proc. ICASSP'91* (Toronto, Canada, May 1991), Vol. 1, pp. 105-108.

[6] Haffner, P. and Waibel, A., "Multi-State Time Delay Neural Networks for Continuous Speech Recognition," *Advances in Neural Network Information Processing Systems 4*, Morgan Kaufmann Publishers, 1992, pp. 135-142.

[7] Hauptmann, A., "Speech and Gestures for Graphic Image Manipulation," *Proc. CHI'89* (Austin, Texas, April-May 1989), pp. 241-245.

[8] Koons, D.B., Sparrell, C.J., and Thorisson, K.R., "Integrating Simultaneous Input From Speech, Gaze, and Hand Gestures," *Intelligent Multimedia Interfaces*, Maybury, M.T. (Ed.) (MIT Press, 1993), pp. 257-276.

[9] Miller, L.G. and Gorin, A.L., "Structured Networks for Adaptive Language Acquisition," *Int. J. Pattern Recog. Artificial Intell.*, Vol. 7, No. 4, 1993, pp. 873-898.

[10] Mitchell, T.M., *Machine Learning*, WCB/McGraw-Hill, 1997.

[11] Ney, H., "The Use of a One-Stage Dynamic Programming Algorithm for Connected Word Recognition," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 32, No. 2, 1984, pp. 263-271.

[12] Oviatt, S.L., Cohen, P.R., and Wang, M., "Toward Interface Design for Human Language Technology: Modality and Structure as Determinants of Linguistic Complexity," *Speech Communication* (Netherlands), Vol. 15, Nos. 3-4, Dec. 1994, pp. 283-300.

[13] Rumelhart, D.E. and McClelland, J.L., *Parallel Distributed Processing: Exploration in the Microstructure of Cognition* (Vols. 1 & 2), MIT Press, 1986.

[14] Sankar, A. and Gorin, A.L., "Adaptive Language Acquisition in a Multisensory Device," *Artificial Neural Networks for Speech and Vision*, Mammone, R. (Ed.) (Chapman and Hall, London, 1993), pp. 324-356.

[15] Vo, M.T. and Waibel, A., "Multimodal Human-Computer Interaction," *Proc. ISSD'93* (Waseda, Japan, 1993).

[16] Vo, M.T., Houghton, R., Yang, J., Bub, U., Meier, U., Waibel, A., and Duchnowski, P., "Multimodal Learning Interfaces," *Proc. ARPA SLT Workshop 95* (Austin, Texas, 1995).

[17] Vo, M.T. and Wood, C., "Building an application framework for speech and pen input integration in multimodal learning interfaces," *Proc. ICASSP'96* (Atlanta, Georgia, May 1996).

[18] Waibel, A., Vo, M.T., Duchnowski, P., and Manke, S., "Multimodal Interfaces," *Artificial Intelligence Review*, *Special Volume on Integration of Natural Language and Vision Processing*, McKevitt, P. (Ed.), Vol. 10, Nos. 3-4, 1995.

[19] Ward, W., "Understanding Spontaneous Speech: the Phoenix System," *Proc. ICASSP'91* (Toronto, Canada, May 1991), Vol.1, pp. 365-367.