The following paper was originally published in the
Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)
Monterey, California, June 1995.

Media-Independent Interfaces in a Media-Dependent World

Ken Arnold
Sun Microsystems Labs

Kee Hinckley
Utopia, Inc., Winchester, MA

Eric Shienbrood
Wildfire Communications, Lexington, MA

Media-Independent Interfaces in a Media-Dependent World

Ken Arnold	Kee Hinckley	Eric Shienbrood
Ken.Arnold@east.sun.com	*nazgul@utopia.com*	*ers@wildfire.com*
Sun Microsystems Labs	*Utopia, Inc.*	*Wildfire Communications*
2 Elizabeth Dr.	*25 Forest Circle*	*20 Maguire Rd.*
Chelmsford, MA 01824	*Winchester, MA 01890*	*Lexington, MA 02173*

Abstract

Wildfire is a communications assistant that uses speech recognition to work over phone lines. At least that's what it is today. But in the future it wants to run on desktops, PDAs (like the Newton Message Pad), and who knows what all. To provide a level of media independence, we designed a subsystem to isolate the communications knowledge of the assistant from the mechanisms of prompt/response. This layer is called the MMUI. It provides abstractions of input and output that let the assistant ask questions and get responses without knowledge of the specifics of the communication channels involved. The specifics of speech recognition, as well as the degree of abstraction desired, make this an interesting case of presentation/semantic split using object polymorphism. This presentation will cover the design of the MMUI, its fundamental weaknesses, and furious handwaving over future directions to mend them.

1. Introduction

The Wildfire communications assistant is designed to use computer analysis and assistance to enhance communication, both with other Wildfire users and with the outside world. To do this, the interface is critical: it must be natural and easy to use, engaging without wasting your time, and so on. There is nothing in any reasonable requirement list that says it must *only* work over voice interaction, and in fact, future expansion pretty much demands that it be more flexible than that, for example to operate with tty lines for the deaf, text-based two-way pagers, and eventually pen-based PDAs and GUI-based desktops.

However, much of the value added of the system has nothing in particular to do with the presentation of the interface. The primary value is provided using knowledge of how and when to get in touch with a person, who has called and when you need to call them back, who is currently on hold and who is important, and how to weave these facts into more effective assistance to the user.

The concept of a presentation/semantic split in application design is well established [1,2,3,4,5], so it was obvious that a presentation layer needed to be provided for developing the Wildfire assistant. This layer, however, has several additional requirements:

- It needs to be able to handle completely linear interactions. A voice interaction is like a conversation, in which the assistant asks questions, waits for a response, and then either gives feedback or asks another question.
- Voice interactions have a different quality to the interaction than a GUI does. As an example, because of the nature of speech recognition, a choice from a menu is not a single selected thing, but a list of probability-ordered *possible* responses.
- In any single interaction, user input can come from a variety of source. For example, you can either speak your responses, or you can use touchtone shortcuts.
- Speech recognitions systems can require training, resulting in a significant collection of user-specific data associated with the general menu.

So beyond the normal presentation/semantic considerations, there was a need to interact with the user independent from the specific ***media*** through which the interaction was taking place. Presentation could be through recorded voice, text-to-speech, plain or internationalized text, or a GUI presentation mixing images and text. Input could be recorded voice, recognized voice, text, images (such as faxes) and so on.

The independence of the need for interaction from the media through which the interaction takes place, and the great value represented by the underlying communications enhancement independent of the interface, made it attractive to create an abstraction capable of isolating the assistant code from the particular media(s) through which the user was interacting.

We also added the following requirements:

- It must be easy to add new types of media. It is impossible to predict who will make the next advancement in price or functionality in speech recognition, or to predict the winner in the two-way pager marketplace. Adding new kinds of media should require relatively minimal work that is completely hidden from the assistant.
- It should be possible to select particular media based on other attributes. While providing a layer of abstraction for the variability of media, adding a generic variability mechanism to select for, say, the desired prompt verbosity seemed like a small addition for a large gain.

2. The MMUI

The abstraction we designed is called the MMUI (***M***ulti-***M***edia ***U***ser ***I***nterface). It lets the assistant interact with the user through a very detached abstraction using units of meaning.

(For those of you who want to follow along with pictures, Figure 1 shows the class hierarchy for most of the classes described in this paper.)

```
Meme
 └── Menu
Attributed
 ├── Media
 │    ├── TextMedia
 │    ├── NMSAudio
 │    └── VideoMedia
 └── ActiveGadget
Mogul
 ├── TextMogul
 ├── VPCMogul
 └── VideoMogul
Port
 ├── InputPort
 │    ├── NMSInputPort
 │    └── VideoInputPort
 ├── OutputPort
 │    ├── NMSOutputPort
 │    └── VideoOutputPort
 └── RecogPort
      ├── VPCPort
      └── VideoRecogPort
Gadget
 ├── PhoneGadget
 └── NetworkGadget
Channel
 ├── PhoneChannel
 └── NetworkChannel
```

FIGURE 1. MMUI Class Hierarchy

The basic meaning abstraction is the ***Meme***. When the assistant wants to say "Hello", it doesn't care if the presentation is in text, voice, or video, or even what language is used. What matters is that the user is presented with representation of the concept of "Hello" that is meaningful to them.

Each representation of the concept "Hello" does, in the end, have to be presentable in some way via a specific representation understood by a particular device, such as an audio board or an ASCII stream. Specific presentable data are represented by ***Media*** objects. The media class is an abstract base class from which specific media representation classes are derived.

The "Hello" meme would thus contain several media objects of various types, such as a TextMedia object containing the string "Hello" and/or an AudioMedia object that describes a recording of someone saying "Hello". Thus, Meme is a concrete class that has a set of media that all represent the same concept through specific presentation possibilities. When new kinds of media are added to the system, the Meme class does not change, and hence the assistant need not change either.

Note that, of the two requirements on the media within a meme (that it be presentable and equivalent in meaning), the MMUI is concerned only with presentability, not meaning. It assumes that some human took responsibility for properly correlating the meme's contents to the meaning it is intended to represent.

All this is sufficient for output, but output is the simplest part of the system. Interpreting user choices gets more complicated, but builds on the same basic concept of grouping identical conceptual meanings that require different specific interactions into a single conceptual unit.

So we extend the basic Meme concept with a derived class that represents a set of choices. This is called a ***Menu***. It contains an ordered list of rows, each of which has a meme to represent its presentation and program-supplied data to be returned if it is selected. In order for it to be presented, and the user input interpreted, some expert code is required that understands how to use the data in the menu to recognize. This expert is called a ***Mogul***. A TextMogul would understand how to present the text media from each meme to the user as, say, a numbered list and let the user type in the number of the choice they wanted. A voice-recognition Mogul will, as we shall see, be more complicated.

Presentation and recognition are done through an abstraction called an ***ActiveGadget***. This represents an active one- or two-way channel between a process (the assistant) and a gadget (like a telephone) that is currently active, i.e., able to be interacted with (interacting with a telephone that does not have a person using it, and is thus not “active”, is beyond the scope of this paper). Memes are presented on the channel, and recognition is done through it.

ActiveGadget and the base Media class are both derived from a base ***Attributed*** class that allows setting arbitrary name/value pairs on an object. The attributes on the ActiveGadget are used to store the output filtering preferences. The attributes on a Media object describe the attributes that particular media have. Matching between these values allows further refining of the list of Media that can be presented, once irrelevant Media (like voice media on a tty line) have been weeded out. This could be used to select verbose vs. non-verbose prompts, male vs. female speakers, etc. More on this later.

Before we talk about how this is translated into classes, objects, and other code-related realities, we will need a quick overview of the Wildfire architecture.

3. Wildfire Architecture

For the purposes of this paper, we will describe a simplified view of the Wildfire architecture from a MMUI-centric point of view. Basically, the Wildfire system is broken into a set of applications (of which the most visible is the assistant) that run on a specialized kernel called the Wildfire OS (WFOS for short). This exports functionality for communicating via ActiveGadgets to devices that support particular capabilities (such as speech recognition or recording).

The assistant gets an ActiveGadget by handing a Gadget description to the WFOS and getting back an ActiveGadget object that represents a channel of communication to that gadget. The assistant then requests capabilities of that ActiveGadget, which (if the addition is successful) will ensure that a port is attached to support that capability.

There are three major kinds of ports, all derived from an abstract ***Port*** class: ***InputPort*** for recording user input, ***OutputPort*** for presenting media, and ***RecogPort*** for ports that turn input into recognized selections. Currently a channel that represents normal communication with an assistant over a telephone line has the following ports attached to it:

- An InputPort that can record sound, such as a person leaving a message.
- An OutputPort to play pre-recorded sounds, such as system prompts.
- A speech recognition port that is derived from both OutputPort and RecogPort so that it can coordinate playing the sounds with setting up the recognition.
- A RecogPort to recognize touchtone selections.

As another example, a channel communicating to a user across a file descriptor currently has the following ports:

- An InputPort that records ASCII text.
- An OutputPort that prints ASCII text.
- A RecogPort that lets the user select an entry from a text menu.

4. And Now, Back to the MMUI

Now we can discuss in greater detail what the MMUI actually does.

First of all, let’s examine what’s in a Meme. A Meme has a set of Media-derived objects, each of which describes a representation of a single concept. Different media objects might be of the same C++ type, but still be logically distinct due to attribute differences. For example, the “Call Whom?” Meme might have all the following Media objects:

- A TextMedia object with the string “Call Whom?”
- A TextMedia object with the attribute “Length=Tutorial” and the string “Call Whom? Please say the name of one of your contacts.”
- An NMSAudio[1] object with a pathname to a recording of a person saying “Call Whom?”
- An NMSAudio object with the attribute “Length=Tutorial” and a pathname to a recording of a person saying “Call Whom? Please say the name of one of your contacts.”

There might be other recognized values for the Length attribute, such as “Brief” for which the representation might be “Whom?” or (for the audio) nothing at all. There might be other attributes, such as “Speaker=Female/Male” to allow the user to select a

1. NMS is the particular vendor whose board we use for basic audio presentation.

female or male voice for the prompts. For prompts that include the gender of a person (like “She’s not here”) that could be encoded in a “Subject=Male/Female” attribute.

There are a set of system memes, which are simply well-known memes within a particular name space. These are referenced through MemeID objects that uses the name of a meme to get a pointer to the actual meme from a table of system memes. These memes are not special in any other way beyond being entered in this name space. Many memes are created on the fly, such as user names, contact names, and message bodies.

4.1. Presentation

Now let us examine how the actual presentation of a Meme works. We will work on the example

```
ag << WfCallWhomP;
res = ag.prompt_response(WfOutCallM);
```

Memes are buffered until an explicit flush or a request for input. Thus, the `prompt_response()` invocation causes the `WfCallWhomP` prompt meme to be presented. Here is how that happens:

1. The `WfCallWhomP` MemeID is translated into a Meme reference.
2. ActiveGadget’s `operator<<` method sends the Meme list (of one Meme in this case) to the WFOS to present through the channel.
3. The channel asks each attached OutputPort to present the Memes that have Media which that OutputPort understands.
4. Each OutputPort searches the Meme for Media objects of types that it understands. If it finds more than one such object, it does an attribute match to find the one whose attributes best match those set in the ActiveGadget. If there is a tie for best match, one of the best is picked randomly.[2]
5. When no more Memes are left to present, processing is complete.

When reducing the mass of possible Media down to the one that will be presented, the primary filtering is, of course, on presentable Media. For many Memes this may be enough. Attribute matching only matters if more than one presentable Media exists in that Meme. Attribute matching does not take adjacent Memes into account to smooth matching across a list of Memes (mostly because we’ve never found a use for it).

Ports find Media-derived objects that they understand using a runtime typing system. The NMSOutputPort will try and find Media that are at-least-a SimpleAudio media, which is a derived type that describes basic mulaw audio. The polymorphic behavior of Media and the various Port classes gives the MMUI great adaptive power. We will describe this in detail below, when we discuss how one would go about adding a new type of interaction to the system.

Also note that there is no 1-to-1 requirement between Ports and specific Media types. A Port may handle more than one kind of Media, and any number of Ports may understand a specific Media type. Since Ports are attached to channels because of requests for capabilities, it can be quite useful if a Port handles more than one Media, since it might reduce the number of Ports required to support a given capability.

Attribute matching is done in a very simplistic way. First, we take a list of the attributes meant to qualify the presentation, i.e., the attribute set from the ActiveGadget is overlaid with the set overriding the current presentation. (This can done with an option object analogous to the iostream manipulators.) For each attribute specified in the qualifying list, each presentable Media object is assigned a score: 2 for a match, 0 for a conflict, and 1 for a non-conflict (in other words, if the attribute in the qualifying list is absent from the Media, and hence, neither set correctly nor incorrectly). The presentable Media object with the highest score wins. This can mean that an object that directly conflicts with the qualifying list is presented, on the theory that the wrong output is better than no output at all. Wrong output tends to lead to complaints, which can often lead to getting the problem fixed. Silence tends to merely baffle.

4.2. Recognition

Recognition is somewhat more complex than presentation. This is partly due to the fact that more stuff has to go on in a two-way interaction than a one-way data dump, but voice menus also have an attribute rarely found in other menu systems: probability. When someone selects the third entry in a menu, that’s what they’ve selected. When a speech recognition system tries to determine what you’ve selected, it can only return a *list* of probable selections. Its matching is only the best it can do. (The details of translating the actually output from the speech recognition algorithm into a probability is an interesting problem, but it is not in the scope of this paper; see [9,10,11].)

2. This is useful for areas where it is best to have the output vary. For example, the “You’re welcome” Meme might have “You’re welcome”, “Don’t mention it”, “Happy to help”, and so on, leading to a more personal interface.

This means that the output of a menu selection in a media-independent interaction is a list of candidate selections with some probability assigned to each. If the highest probability falls below some threshold, then the recognition must be considered a failure and handled appropriately. We notify the user in increasingly verbose ways that we didn't understand them, and then we finally give up on the whole command. (The issues surrounding the human factors of deciding how to handle these cases is another interesting, yet uncovered, topic; see [6,7,8].)

The menus can usually handle this kind of reprompting given the appropriate options (such as what to say when reprompting, and what level of probability is acceptable). There are, though, instances where the selection must be returned to a higher level.

Let's walk through the recognition request:

1. `WfOutCallM` is translated into a Menu reference. (There are system menus just like there are system memes, and the lookup is handled analogously.)
2. The menu, along with any specified options (none shown) are sent to the WFOS.
3. The WFOS waits for any asynchronous output request to finish.
4. It then asks each OutputPort to prompt/recognize on the pending output. This allows recognition ports to coordinate the ending of the prompt and the start of recognition.
5. The first RecogPort to say it is complete wins, i.e., its recognition result is the result of the overall recognition (no averaging or other inter-port data merge is done).
6. The responses are sent back to the MMUI, which asks the winning Mogul to translate the results into a canonical form, which is a list of ***MenuPick*** objects ranked by probability. (A MenuPick object holds the probability for that pick, an index into the Menu, and a pointer to the assistant-specified data associated with that row of the menu).
7. This list of MenuPicks is returned as the result, along with an overall confidence in the recognition, and (if appropriate) a failure status to distinguish failure due to timeout from failure due to unrecognizable noise.

If the recognition fails, step 6 instead consists of "bonking" the user with any other corrections necessary, all as specified by options provided to the menus via the ActiveGadget (which maintains the current default option settings) or specific overrides which can be given as an optional parameter to the `prompt_response()`.

Step 4 is where Moguls come into play. Speech recognition systems typically need a block of data that represents the entire menu of choices. This describes (in some device-dependent way) the differentiating aspects of the legal things a person can say. A common term for this is the ***vocabulary***. When you want to recognize words from a particular vocabulary, you must download the data into the device, and only then can you start recognition.

This means that a simple list of the possible choices is insufficient to represent a Menu on all input systems. While a list of Memes containing TextMedia would be sufficient to build a text or GUI menu on the fly, vocabulary building is a time-consuming process that offloads overhead from the recognition phase to a vocabulary building phase that must precede it. The Mogul was introduced to represent any overall menu-related information required by a RecogPort, such as vocabularies. We will deal more with how vocabularies are built below.

The coordination in step 4 is needed because, on some speech recognizers, if you do not coordinate the input and output, you can get very bad effects. On some systems, starting to recognize a person's voice while sound is still being played over the phone can lead to very bad effects, since you can start recognizing your own prompts as commands. On the other side, the prompt may have finished, but the recognition port may not yet be ready, i.e., the vocabulary may not have yet finished being downloaded to the device. Wildfire plays a little *blip* sound when it is ready to start recognition. This lets you know that, should the prompt finish too soon, you still need to wait. Otherwise people would start talking before the recognition window began, and Wildfire would start trying to understand them midway through their utterance. So this *blip* is coordinated with the output so that it does not play until the recognizing hardware is prepared.

We do not attempt to average results from multiple ports because we cannot see any meaningful way to do this in general. It is hard to even imagine that this can be usefully attempted. Imagine that the user had, when asked whom to call, said "Georgina Whit" and, before the requisite pause to signal the end of speaking, pushed the touchtone for "Never Mind" (the cancel command). How would one average such input?

One could imagine cases where combining various inputs you could increase the correctness of the recognition. On a video phone, for example, being able to match lip movements against what the speech recognition thought was said might be able to help discrimination between possibilities. (I *did* say one could "imagine" such a thing.) To do this requires sophisticated interactions between the data available from multiple sources, and logically, within the MMUI, belongs in a single RecogPort that examines and correlates the relevant data. Neither the WFOS nor the MMUI could possibly broker this interaction in a general, abstract way applicable to other types of ports balancing multiple inputs.

4.3. Training

There are speech recognition systems that work on "speaker-independent" recognition. This means that any person should be recognized without having to train the system about how they personally speak. Although this is ideal in theory, even in the best current systems there are people with heavy accents or speech disabilities who cannot successfully be recognized.[3]

For this and other reasons, Wildfire (and hence, the MMUI) must support user-specific training for vocabularies. A large subsection of the MMUI is devoted to training, and the requirements of speech recognition training constrained parts of the design.

The MMUI supports a training call that sorts the user's menus based on what could use the most training, and presents them to the user one-at-a-time for training. To train a single menu, the user is asked to say each word.[4] Each mogul then uses that data to update its information.

Currently, only the speech recognition mogul VPCMogul[5] uses this step, but it is critical to its operation. The user's provided training for each word is added incrementally to the vocabulary so it can take effect immediately after the training is finished. At a later time, a batch process notices that there are new trainings for a menu, and it rebuilds its vocabularies in a more compact, and effective, form.

If you had, in some way, a recording of the user saying a word, you could add this without interacting with the user, but the vocabulary rebuilding work would still be required, which relies on access to the speech recognition device. This means that it is not possible to simply add a new meme to a menu, or to modify its contents by adding a new AudioMedia to change the recognition for that menu entry. Such a change requires active intervention by some code that understands how the device works. This is the Mogul's job. When a new row is added to a menu, each associated mogul is "introduced" to the new row's meme. This also happens when a row's meme is replaced; there is no mechanism for changing the contents of a meme in a menu except by wholesale replacement. It would clearly be possible to design a method to do so, but we have not yet needed to. And, obviously, when a row is deleted, all the menu's moguls are notified of that, too.

Currently, whenever a menu is created, all Mogul types are created and attached. There is a design, not yet implemented, to make this more dynamic based on content, but this wholesale approach has not yet proved to be a problem.

5. Extending the MMUI

One of the important design centers for the MMUI was the ability to extend the system by adding new kinds of interactions. There were several reasons for this requirement:

- Replacement hardware is constantly becoming available. If a new audio-playing board comes out that is much cheaper to use, it should be easy to change the system to use it, thus allowing a quick reduction in the price of shipping systems.
- New capabilities are coming quickly, too. The sophistication of speech recognition systems is increasing rapidly, and the MMUI should not be a bottleneck when deciding how quickly we can use better solutions.
- Completely new interactions should be quick to add. As two-way pagers become lighter and more widespread, we should be able to add them to the full system (for text-based ones) or for specific interactions (such as saying who is calling and asking if you'd like to take the call). The list of other possibilities is as long as your wired imagination can make it. Again, the MMUI side of this should not be the controlling factor in the schedule.

So we designed a system in which the code that must be added is isolated under five primary abstract base classes, Media, Mogul, Port, Gadget, and Channel.

3. Note that all discussions here pertain to speech recognition over the telephone. This introduces are large degree of variations due to noise, line quality, dropping, and so on. Voice recognition based on high-quality microphones directly connected to computers have an easier time of it.
4. Note that "word" here indicates a recognizable utterance that appears in a menu's vocabulary; in this sense, "tell me" is one word; the computer can't tell and doesn't care that it is, to humans, actually a phrase.
5. VPC is the particular vendor we use for most voice recognition systems.

- ***Media:*** If presentation will be done on the new device, a new Media will probably be needed to describe presentable data.
- ***Mogul:*** If recognition will be done on the new device, a new Mogul type will probably be needed to manage that complexity.
- ***Port:*** New input, output, and/or recognition ports will be required that understand how to work within the WFOS to drive the device.
- ***Gadget:*** Adding a new device may require adding a new Gadget to describe an address for the device
- ***Channel:*** A channel understands how to establish and terminate connections to Gadgets (e.g., how to dial the phone and hang up), and coordinate between multiple ports on a particular kind of gadget.

We have only briefly touched on the Gadget class. Gadget is an abstract class whose derived classes contain addresses that can be used to connect to specific targets, establishing a channel for an ActiveGadget. The details of that handshake are not terribly interesting here, but any device we talk to must be contacted at an address, and hence, must have a Gadget-derived type that contains that address. For a PhoneGadget the address is a phone number; for a NetworkGadget it is an internet address and port number that will be used for the network connection.

We have also not discussed Channels in any detail either, but they are rather straightforward. They are created to carry media to destinations specified by particular Gadget types. If the new device does not require a new Gadget type (e.g., it operates at a network address, for which Wildfire already has NetworkChannel type), it will not require a new Channel. If it does, the Channel will have to be able to resolve the address described in the Gadget to a Channel that can juggle the communication needs of that type of Gadget. Like Port, a new Channel is not very complicated beyond whatever is required by the device that connects to the machine. The NetworkChannel is trivial, since sockets are easy to manage. The PhoneChannel is more complicated because telephony has more intermediate and failure states.

Not all new systems will require all of these. Besides the obvious fact that output-only systems will not require recognition Moguls, new devices that work with phone lines for addresses would not require a new Gadget or Channel type. This is likely to be common; a fax system would need a new Media and Port type, but no new Gadget would be required.

To give a flavor of how we would add a new device to the MMUI, we will describe how to do this for a putative video-based system.

It is quite likely that the new video would be reachable either on the local network via an internet address, or over the phone like a video-conferencing system. Both of these Gadget and Channel types already exist in the Wildfire system, so we will just piggyback on them.

If a new address *was* required (some video systems, for example, require two phone lines to handle the volume of data), a new Gadget type would have to provide a way to hold such an address. A new channel type that recognized that address and knew how to establish a connection to the addressed video system would also have to be added. Since this is not the primary topic of the paper, we will skip the details of this mechanism, but suffice it to say that it is not very complicated, except whatever complication the video system itself may impose.

For this one needs a VideoMedia class, derived from the abstract Media class. The Media class doesn't require much of its derived classes; it mostly (being derived from the Attributed class) ensures that Media can be attributed. Almost all functionality is added in the specialized classes. The VideoMedia would presumably store a pathname of a file that contained the media, and probably a start and end frame within the file.

Providing input and output ports for the video system is again relatively straightforward. New classes would be derived from the InputPort and OutputPort classes that overrode the pure virtual `record()` and `present()` methods respectively. The `record()` method's main job would be to dump the video into an appropriate place, and create a VideoMedia object that described it.

The `present()` method would paw through the Media in the list of pending output, looking for VideoMedia objects. If it found one, it would do the same for the next meme in the list, continuing on until it reached either the end of the meme list, or a meme that didn't contain any VideoMedia. It would then peel off the memes it *could* reasonably present, and, for each one, find the best attribute matched VideoMedia and present it.

As it currently stands, we could use the system described only for input and output, but not for recognition. We could take a video message, or play a video clip, but we couldn't ask any questions. Let us presume that this video system has an attached keyboard for answering questions (since I suspect a sign-language gesture recognition system is currently a tad beyond even the limits of handwaveware).

It is possible even here that we can avoid any hard work, since there already is a text recognition port, and if the video system understands simple ASCII I/O, a TextRecogPort would fulfill our engineering needs. It would not, however, fill our pedantic needs, so we will assume that this is not so.

So we need to add a VideoRecogPort and a VideoMogul. The Mogul class requires its derived classes to handle notification of changes in the Memes of the Menu, and to be able to canonicalize the list of MenuPicks into a form presentable to the application. The VideoMogul would be interested in the TextMedia of the menu so it could create a menu on demand (any Mogul can look at any Media to do its work). It might also be interested in VideoMedia that presented the choices in a video form, should the user want some help understanding the available choices.

The VideoRecogPort would present the list of choices as text and allow the user to select one in some way.

Notice that this does *not* preclude doing simultaneous speech recognition on the video system's microphones. Just as touchtones and voice can coexist on a single ActiveGadget, so can voice and the video systems text selection mechanism (and touchtones too, if desired). Whichever got an answer first would govern any particular recognition, but each recognition could be responded to in any available way.

6. Current State

The current state of the MMUI allows a linear presentation style with quite a lot of separation between the internals of the code and the particular media presentations required to present, record, or recognize in a particular case. However, as currently implemented, the MMUI still has major weaknesses if one is to consider it as a general interface abstraction.

First, it has no concept of "sentence". The presentation of Memes is linear in nature, with the order specified by the assistant. This is one problem (of several) that makes porting the assistant to a different language difficult. The natural or allowed order of presentation could be quite different.

There is also no concept of "dialog". Most actions require more than one piece of data. Again, different languages may impose a different expected or required order on gathering the data. For example, in English it is natural to say "Call Gordwina at work", but another language may prefer "Call the workplace of Gordwina". Further, a particular piece of data may affect the valid values of another one; Gordwina may or may not *have* a work phone.

There is also the issue of context. In a conversation, much data can be left out, inferred by the context. Overall context (for example, the subject of discourse) is easily built into the assistant interactions, since it is obvious and shared — we are talking about calling people, for instance, by the nature of the dialog, and so the dialog designer can make the interface understand that context.

But specific context is harder to provide. It would be nice to use words like "it" or "them". But without a notion of the type of legal referent, and the history of previous interactions that might state or imply a referent of that type, such words can only be used in a highly constrained way.

These problems limit the use of the MMUI to linearly presented interfaces. If one wanted, for example, to add a desktop GUI interface, one could only do so using a series of single-question prompt-response dialogs. Graphical icons could be included as part of the dialog description media, but it would not be possible to pop up a full "Place A Call" dialog that let one specify both whom to call and where to call them.

Designs exist to address these problems by adding classes and protocols to represent each, but as yet no prototyping has been done to try and prove them actually useful in the cauldron of the real world.

On the other hand, the Wildfire system is currently available, and is built using this infrastructure. It is quite possible to interact with the assistant using intermixed voice and touchtone commands, to listen to data recorded in different formats (hence, having different Media objects representing them), and to use attributes to select particular media (tutorial vs. standard prompts). As an experiment, we added a new channel type able to talk across a text-based two-way pager. With no modification to the assistant, it was possible to see the prompts and respond via the pager's keyboard just as one did when communicating via a Network Gadget to a local terminal emulator.

The MMUI has proven to be a useful tool in isolating many details of the interface presentation from the duties of the assistant. It has proven its adaptability to different flavors of linear presentation. It is easy to add new types of presentation and recognition, and should be possible to extend to provide greater isolation to the presenter. These benefits make it a useful abstraction in designing media-independent interfaces that must be presented in a media-dependent world.

Acknowledgments

Tony Lovell, Vinnie Shelton, Keith Gabryelski, Rich Miner, Greg Cockroft, and Dave Pelland contributed significantly to the design and implementation of the ideas presented here. Bill Warner created the concept of the WIldfire Assistant which motivated this design.

References

[1] Eguene Cicarelli, "Presentation Based User Interfaces", Thesis, MIT AI Lab, Technical Report AI-TR-794, 1984

[2] Pedro Szekely, "Modular Implementations of Presentations", Proceedings SIGCHI+GI 1987, pp. 253-240.

[3] Pedro Szekely, "Separating the User Interface from the Functionality of Application Programs", Thesis, ECMU 1988.

[4] Scott McKay, William York, Michael McMahon, "A Presentation Manager Based on Application Semantics", Proceedings SIGGRAPH Symposium on User Interface Software and Technology 1989, pp. 141-148.

[5] H. Rex Hartson, Deborah Hix, "Human-Computer Interface Developments: Concepts and Systems", ACM Computing Surveys, 21:1, pp.5-92.

[6] Candace Kamm, "User Interfaces for Voice Applications" *Voice Communication Between Humans and Machines*, National Academy Press, Washington, D.C., 1994.

[7] Eric Ly, Chris Schmandt, "Chatter: A Conversational Learning Speech Interface", *AAAI Spring Symposium on Intelligent Multi-Media Multi-Modal Systems*, Stanford, CA, March 1994.

[8] Nicole Yankelovich, Gina-Anne Levow, Matt Marx, "Designing SpeechActs: Issues in Speech User Interfaces", Proceedings, SIGCHI '95 Conference on Human Factors in Computing Systems.

[9] Gordon E. Pelton, *Voice Processing*, McGraw-Hill, 1993.

[10] L. R. Rabiner, B. H. Juang, *Digital Processing of Speech Signals*, Prentice-Hall, 1978.

[11] Kai-Fu Lee, *Automatic Speech Recognition*, Kluwer Academic Publishers, 1989.