Requirements for Automatically Generating Multi-Modal Interfaces for Complex Appliances

Jeffrey Nichols, Brad Myers, Thomas K. Harris,
Roni Rosenfeld, Stefanie Shriver
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
jeffreyn@cs.cmu.edu
http://www.cs.cmu.edu/~pebbles/puc/

Michael Higgins, Joseph Hughes
MAYA Design, Inc.
Suite 702
2100 Wharton Street
Pittsburgh, PA 15203
higgins@maya.com

Abstract

Several industrial and academic research groups are working to simplify the control of appliances and services by creating a truly universal remote control. Unlike the preprogrammed remote controls available today, these new controllers download a specification from the appliance or service and use it to automatically generate a remote control interface. This promises to be a useful approach because the specification can be made detailed enough to generate both speech and graphical interfaces. Unfortunately, generating good user interfaces can be difficult. Based on user studies and prototype implementations, this paper presents a set of requirements that we have found are needed for automatic interface generation systems to create high-quality user interfaces.

Keywords: Multi-modal interfaces, speech recognition, handheld computers, remote controls, appliances, personal digital assistants (PDAs), Universal Speech Interface (USI), Pebbles, Personal Universal Controller (PUC)

1. Introduction

Home and office appliances are becoming more complex as embedded computers enable new kinds of functions. As complexity increases, appliance user interfaces usually get harder to use [2]. Many of these interfaces also have no accessibility options for helping impaired users, such as the blind, make use of the appliance.

Several groups, including our own [8], suggest that separating the interface from the appliance might solve these problems. Each user would carry a *personal universal controller* (PUC), a device that allows the user to interact with all the appliances and services in her environment. When the user wants to control an appliance, the PUC would communicate with the appliance, download a specification of the appliance’s functions, and then automatically generate a remote control interface suited to the PUC device and the user. The PUC and the appliance would continue to exchange messages as the user manipulates the interface and as the state of the appliance changes.

Separating the interface from the appliance has several benefits for users. A PUC device can take many forms. Impaired users could get controller devices that are specifically customized to their particular impairment. For example, a blind user might have an interactive Braille surface or a small headset that supports speech recognition and speech output. Unimpaired users benefit from controller devices with technology that would be too expensive to put in every appliance. For example, an unimpaired user could get a handheld controller with a high-resolution color LCD touch-screen. No matter what device is chosen for use as a PUC, the interfaces generated will be consistent with other applications on that device. Another benefit is that the user could also receive a consistent interface across appliances. For example, similar user interfaces could be presented for controlling the playback of audio and video from different appliances.

The benefits of separating the interface from the appliance may be reduced if the generated interface is not of sufficient quality. In order to ensure that interfaces generated by a PUC do not suffer from this problem, we started by investigating what is required for generating high-quality user interfaces from a specification language. We began by designing high-quality controller interfaces by hand, just as we would like the PUC to do automatically. We created graphical interfaces for an Aiwa shelf stereo and an AT&T telephone/answering machine, and speech interfaces for an Audiophase stereo and a phone-based movie database [14]. After many design iterations, when we felt that the interfaces were as good as we could make them, we conducted several user studies that showed, for example, that users were twice as fast and made half as many errors with our hand-designed interfaces as with the original appliance interfaces [9]. Besides confirming our designs, the user studies also helped us identify additional issues and determine the key properties that made the interfaces work well.

The main goal of first designing our own high-quality user interfaces was to discover what functional information about the appliance is required to design a good interface. We found, for example, that disabling the components for functions that were not available helped users tremendously. While this is consistent with user interface guidelines, it has never been followed for today's remote controls. Furthermore, we were surprised to discover that knowing when a function will be disabled based upon the rest of the appliance state is also very helpful for making layout decisions. For example, a set of components that are never active at the same time as another set could be placed on overlapping panels to minimize user confusion and make effective use of space in a graphical interface.

The most surprising discovery after developing our list of requirements is that no current specification system we are aware of fulfills each and every requirement. Some requirements, such as grouping components in a tree, are found universally, while others are always missing. This is discussed further in the requirements section.

2. Related Work

There are many industrial and academic groups that are working on similar problems. Some of these groups [3, 7, 16] have decided to download pre-built interfaces to the controller device instead of making the device automatically generate user interfaces. This approach has the advantage of being easy to implement, but it requires that each appliance be pre-programmed with a set of hard-coded interfaces for every controller device that the appliance will encounter. Even if you allow for the possibility that the appliance can connect to the Internet and download interfaces for new controllers, it seems unlikely that manufacturers would continue to absorb the cost of creating a new interface for every new controller that becomes available. After all, most appliances are expected to last for many years whereas new handheld devices become obsolete within six to twelve months.

Several other projects have included some aspects of automatically generating interfaces. The ICrafter project [12] has support for both downloadable and automatically generated user interfaces. The interface generation system was not the focus of ICrafter however, and it can only generate very simple graphical interfaces. Work by Hodes, *et al.* speaks of a "universal interactor" [4] that adapts to control different devices. The primary focus of that research seems to be on the system and infrastructure issues, rather than the problems of creating high-quality user interfaces.

UIML [1] is an XML language that claims to provide a highly device- and modality-independent method for user interface design. Software generators are currently available that can convert UIML source into concrete interface code such as Java Swing, HTML, VoiceXML, and several others. Unfortunately, UIML seems to rely on explicit mappings between its own elements and the concrete form it will be translated into, negating most of its claimed device independence. Current UIML research is attempting to solve this problem [5, 11].

The INCITS V2 [18] standardization effort, which is creating the Alternative Interface Access Protocol (AIAP) [19], and XWeb [10] are two projects that automatically generate interfaces for multiple modalities. Both are capable of generating graphical and speech interfaces, and AIAP has also been implemented on an interactive Braille surface for blind users. Both of these systems also define an intermediate language for specifying the functions of the interface that will be generated. We will further discuss the strengths and weaknesses of these systems later in the requirements section.

3. Human Designed Interfaces

This section describes our observations from building and testing the interfaces we designed by hand. This work was performed separately for the graphical and speech interfaces.

3.1. Graphical Designs

We hand-designed graphical interfaces on the Compaq iPaq for two appliances (see Figure 1): an Aiwa CX-NMT70 shelf stereo and an AT&T 1825 telephone/answering machine. Since we could not control the actual stereo or telephone, we simulated control with a laptop that responded to signals from the iPaq by playing sounds that closely matched the responses expected of the actual appliances. The interfaces for both appliances were designed iteratively using heuristic analysis. We also conducted think-aloud studies with users to find problems that our own analysis had missed.

Once we were confident that the interfaces were of high quality, we conducted a study to compare our hand-designed interfaces with the actual appliances [9]. The purpose of this study was two-fold: to ensure that our designs were better than the actual interfaces and to find any additional problems that the analysis and think-aloud studies had missed. The study showed that users of our hand-designed interfaces completed tasks in half the time and with half the errors compared to users completing the same tasks on the actual appliance interfaces.

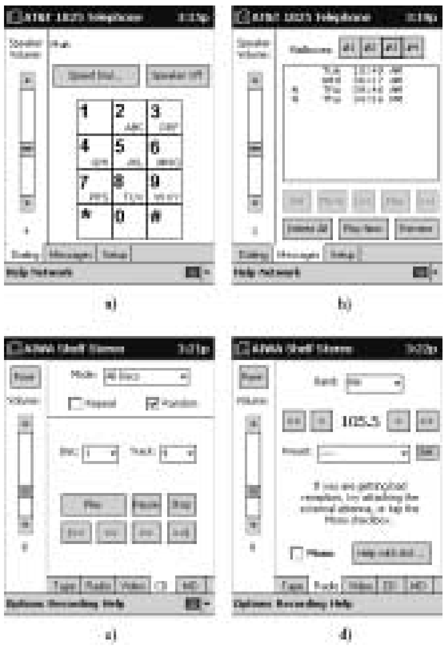


Figure 1. Hand-designed graphical interfaces for an AT&T telephone (a-b) and an Aiwa shelf stereo (c-d).

Most of the problems users had with the actual appliance interfaces could be traced to poor button labels or inadequate feedback. Both appliances had buttons associated with multiple functions, so that pressing a button quickly might invoke one function while pressing and holding a button invoked something else. Feedback was often ambiguous; the phone would beep once to indicate an error and twice to signal a positive result.

We noticed many features of our hand-designed interfaces that seemed to make them easy to use. Good organization was important for making the interface easy to understand. Commonly used functions were placed at the top and to the left. Related functions were placed next to each other on the screen. We were able to use the extra space that the iPaq screen afforded us to put helpful labels on each control. Sometimes we added additional help text when the interface needed to be explained.

The flexibility of the iPaq's LCD screen also allowed us to disable or hide controls that were not currently active (see Figure 1b). On the actual appliances there were more than fifty buttons, many of which were not active. Interfaces for appliances with several modes, such as our stereo, seem to particularly benefit from the technique of hiding inactive controls, because each mode has a set of controls that are not active in other modes. The controls for each mode can be placed on overlapping panels (as in Figure 1c-d), hiding many unusable controls that the user might find confusing.

We also observed that our hand-designed interfaces could be improved in some situations by using standard component configurations that people are familiar with. For example, every telephone has a standard number pad arrangement (see Figure 1a). PUCs must have some built-in knowledge of these standard arrangements.

3.2. Speech Designs

Our hand-designed speech interfaces were developed as Universal Speech Interface (USI) applications [13]. Each USI application uses a standard set of strategies for exploration and invoking functions that help users interact with any USI application. The core features are a small set of keywords and a standard syntax for input and output across all applications. We have hand-designed USI applications for controlling a stereo and for querying telephone-based databases providing movie times, apartment availability, and airline flight information. We conducted a user study on the movie database interface to discover the problems users encounter with USI applications [14].

Building the stereo application taught us that the two biggest problems for a PUC speech interface are *exploration* and *disambiguation*. Exploration is problematic because most USIs have a tree structure that the user must navigate to find the function they want to use. Each node in the tree must have a label, and the number of children at each node must be limited so that the user is not overwhelmed when trying to remember a long list of options. The navigation problem can be partially overcome by allowing shortcuts. The user can say the name of any function and the system will try to match it to a function somewhere in the tree. This creates a problem of disambiguation because it is possible for multiple functions to have the same name. Fortunately, it seems that multiple functions with the same name are rarely active at the same time, so dependency information can be used to determine which function the user intended.

We also conducted a user study comparing the phone-based movie database application built on USI principles with a natural language interface to the same database. All of the subjects in the USI case were able to adapt easily to the syntactic structure of the USI, typically forming a correct USI phrase within their first three utterances. Subjects did have problems with the lack of explicit feedback for each action. This feature has since been added to the system to help users identify and correct recognition errors, and help users identify the context of the current dialog. Of the subjects who used both the USI and natural language interfaces, 80% preferred the USI interface. We believe this was because the capabilities of the USI system were readily apparent at each step, while the natural language system was more ambiguous about its available functionality.

4. Requirements

Our hand-design work with both speech and graphical interfaces led us to develop a list of requirements that our PUC system must fulfill in order to generate high-quality interfaces. This section describes those requirements and discusses how they are fulfilled (or not) by the PUC and other systems.

4.1. Two-Way Communication

One of the most important requirements for any PUC-type system is two-way communication between the controller and the appliance. This is an obvious requirement for any system where the controller downloads an appliance specification before constructing an interface that issues commands back to the appliance. It is important that this two-way communication be maintained throughout the entire session however, so that the controller and appliance can keep their state synchronized.

State synchronization allows graphical interfaces to display information about the current state of the interface that might not be visible on the actual appliance. Graphical interfaces can also use the current state coupled with dependency information (discussed below) to disable components that are not currently active. Knowledge of the current state is also very important for speech interfaces, which must be able to respond to user queries about the current state even if the information is available visually on the appliance. This is especially helpful for blind users or when the user is not near the appliance.

4.2. Simultaneous Multiple Controllers

It is also important that multiple controllers can communicate with the same appliance simultaneously. Users will expect this feature, and it has the added benefit of allowing different interface modalities to be freely mixed together by using several different controller devices in tandem. For example, a user might combine a handheld controller with a headset to create a multi-modal graphical and speech interface. Most current systems seem to fulfill this requirement.

4.3. No Specific Layout Information

The appliance specification should include information about the functions of that appliance, but it should *not* include specific information about how controls should be positioned on the screen. We share this philosophy with the AIAP and XWeb projects but not with UIML, which can include concrete information in its description about layout. This requirement enforces modality independence by limiting how detailed a designer can specify the functions of an appliance. If it were possible to describe concrete interfaces within an appliance specification language, designers would be tempted to include too many details about how each interface should be implemented. This has several disadvantages:

- Appliance specifications will get much longer because each one may turn into a complete description for several different types of concrete interfaces.
- Appliance specifications might lose their forward compatibility to PUC devices of the future. It seems likely that variety will increase in the devices of the future as non-rectangular screens and different interaction styles become more common. For example, specific information for a dialog box-style interface would probably not be useful for a new watch with a circular screen and several nested dials for interaction.
- Some of the other advantages of automatic generation might be lost. For example, a PUC can ensure interface consistency by making certain interactions the same across multiple appliances. This is not possible if the PUC does not have the freedom to choose the interaction style and positioning for representing given functions.

4.4. Hierarchical Grouping

A fundamental requirement of any user interface is good organization, because users must be able to intuitively find a particular function. An appliance specification can easily define organization using a tree to group similar functions. This makes the interface generation process easier, because most concrete interfaces can also be represented as a tree. The utility of trees for grouping seems to be universally accepted; all current systems use some kind of tree for grouping functions.

4.5. Actions as State Variables and Commands

Each action the user can take must be represented in the appliance specification. We found, as have many others, that state variables and commands are a succinct way to represent the manipulable elements of an appliance. Some systems, such as AIAP and Microsoft's UPnP [17], separate the state variables from the commands that act upon them. This means that a specification for a radio might include a station variable, and also tune up, tune down, seek up, and seek down commands associated with the variable. Our PUC system infers as many functions from the state variable as possible, but still uses commands for those functions that cannot be inferred, such as seek up and seek down.

Not every command can be associated with a state variable however, and specification languages must support unassociated commands. Unassociated commands are required for representing functions where there is no notion of state, such as pressing the "flash" button on a telephone. Commands are also useful for situations in which state is not available, perhaps by manufacturer choice or an inherent limitation of the appliance hardware.

4.6. Dependency Information

In most graphical interfaces there is a visual indicator when a control is disabled, such as the typical “grayed out” appearance. We have found that information about when a function is active can be specified concisely in terms of the values of state variables. Not only does this allow graphical interfaces to display an indicator of whether the function is available, but it can also be useful for inferring information about the panel structure and layout of the interface. Appliances with modes especially benefit from this approach, because each mode is typically associated with several functions that are active only in that mode. If the dependency information is in a form that can be analyzed, the interface generator can search for sets of controls that are never enabled at the same time, and then create a graphical interface that saves space and prevents user confusion by displaying only the controls for the active mode. This knowledge can also be used by USI applications to solve the problem of disambiguation.

Dependency information may also be useful for generating help information, as in the UIDE system [15]. UIDE used built-in pre- and post-condition information to determine why a particular function is not available and to generate instructions for making the function available. In our comparison study with the graphical interfaces we observed that users most often sought help when they wanted to use a function that was currently inactive. Dependency information is similar to pre- and post-condition information and could be used to generate the same kind of help as UIDE.

As mentioned above, it is important that dependency information be in a form that can be analyzed by the interface generators. AIAP includes dependency information, but the dependencies are defined as arbitrary ECMAScript expressions which are difficult, if not impossible, to analyze. This would preclude the dependency information from being used for graphical layout, speech generation or command help. The PUC avoids this problem by specifying dependency information as a concise set of equals, greater-than, and less-than relations joined by logical AND and OR operations. The PUC is the only system we are aware of that uses dependency information as an input to its automatic interface generator.

4.7. Sufficient Labels

Our comparison study of the hand-designed interfaces with the actual appliance interfaces showed that good labels are an important part of creating a high quality user interface. Labels are an even more important part of speech interfaces, because there are no graphical hints to assist the user’s understanding of the interface. To give flexibility to the interface generator, a label in an appliance specification should not be a single text string but instead a collection of text strings, pronunciation keys, and text-to-speech recordings. Pronunciation keys and text-to-speech recordings help improve the quality of the speech interface. Multiple text strings give a graphical interface generator the flexibility to select the label with the most information that can be fit in the allotted space. The PUC is the only current system that provides more than just single string text labels.

4.8. Shared High Level Semantic Knowledge

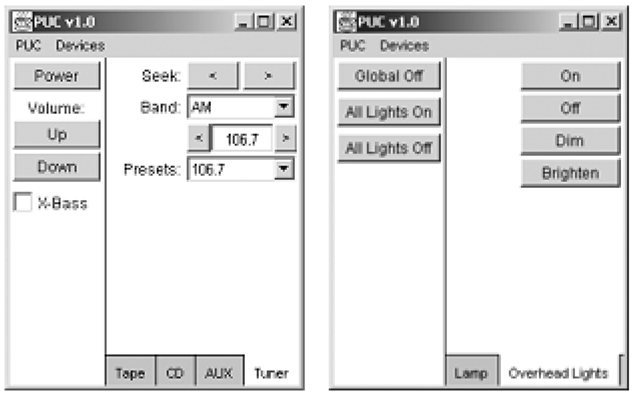
Despite all of the previous requirements, we must concede that it is impossible to encode all the information into an appliance specification that a human would use to design an interface. In addition to functional information about the appliance, a human designer will also use his knowledge of conventions when creating an interface. There are many such conventions, such as the arrangement of buttons on a telephone number pad or the country-specific format for specifying dates. Defining every such convention in an appliance specification is simply not possible.

We can work around the problem by storing this information in each interface generator and creating a standard set of high-level elements in the specification language. If the interface generator understands the high-level element, then it can apply that knowledge to create a better user interface. If not, the generator must be able to ask the appliance for detailed functional information that will allow the generator to create an interface using the normal controls. This approach has the side benefit of making the initial transfer of a specification to the controller shorter, because only the high-level elements are transferred.

Current systems do not seem to address the problem of conventional layouts, although they do make exceptions for handling the most common cases, such as dates and times. The PUC currently implements high-level elements in its specification language, but has no protocol support for generators requesting detailed function information.

5. Requirements in the PUC System

We have created a specification language and implemented two PUC interface generators based upon this list of requirements. Our graphical interface generator is written in Java and runs on a Compaq iPaq handheld computer. Figure 2 shows some example interfaces created by this generator, and further details are available elsewhere [8]. We have also built a preliminary speech interface generator that creates USI controller interfaces. Unfortu-



(a) (b)

Figure 2. Interfaces produced by the graphical generator for *a)* an Audiophase shelf stereo and *b)* an X10 system.

nately, our system does not yet fulfill all of the requirements listed in this paper. For example, we have not completely implemented our approach for handling standard widget layouts as described in section 4.8. Our interfaces also have no facility for generating help text at this time. Despite this, we feel that our generated interfaces are better than can be achieved by any other system, and user studies are in progress to confirm this.

6. Conclusion

In this paper we have described a list of requirements for automatically generating speech and graphical interfaces for complex appliances. These requirements were generated through a human-centered process; we designed interfaces by hand, evaluated them with users for quality, and analyzed them to determine what was required for building a high-quality interface. We are currently using these requirements as the basis for building an automatic generation system for remote control interfaces. We feel that other systems will also need to incorporate these requirements if they want to generate high-quality user interfaces.

Acknowledgements

This work was conducted as a part of the Pebbles project [6]. The speech interface was also conducted as part of the Universal Speech Interfaces project [13]. Marc Khadpe did a portion of the work on the prototype phone interface as a part of a summer independent study project. This work was funded in part by grants from NSF, Microsoft and the Pittsburgh Digital Greenhouse, and equipment grants from Symbol Technologies, Hewlett-Packard, and Lucent. The National Science Foundation funded this work through a Graduate Research Fellowship for the first author, and under Grant No. IIS-0117658. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

References

[1] Abrams, M., *et al.* "UIML: An Appliance-Independent XML User Interface Language," in *The Eighth International WWW Conference.* 1999. Toronto, Canada

[2] Brouwer-Janse, M.D., *et al.* "Interfaces for consumer products: how to camouflage the computer?" in *CHI'1992.* 1992. Monterey, CA: pp. 287-290.

[3] HAVi, "Home Audio/Video Interoperability," 2002. http://www.havi.org.

[4] Hodes, T.D., *et al.* "Composable ad-hoc mobile services for universal interaction," in *Proceedings of ACM Mobicom'97.* 1997. Budapest Hungary: pp. 1-12.

[5] Mir Farooq, A., Abrams, M. "Simplifying Construction of Multi-Platform User Interfaces using UIML," in *European Conference UIML 2001.* 2001. Paris.

[6] Myers, B.A., "Using Hand-Held Devices and PCs Together." *Communications of the ACM*, 2001. **44**(11): pp. 34-41.

[7] Newman, M., Hong, J., Sedivy, J., Izadi, S., Neuwirth, C., Marcelo, K., Edwards, K.W., Smith, T. "Designing for Serendipity: Supporting End-User Configuration of Ubiquitous Computing Environments," in *Designing Interactive Systems.* 2002. London, UK

[8] Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Pignol, M. "Generating Remote Control Interfaces for Complex Appliances," in *Submitted for Publication.* 2002. http://www.pebbles.cs.cmu.edu/papers/PebblesPUCuist.pdf

[9] Nichols, J.W. "Using Handhelds as Controls for Everyday Appliances: A Paper Prototype Study," in *ACM CHI'2001 Student Posters.* 2001. Seattle, WA: pp. 443-444.

[10] Olsen Jr., D.R., *et al.* "Cross-modal Interaction using Xweb," in *Proceedings UIST'00.* 2000. San Diego, CA: pp. 191-200.

[11] Plomp, C.J., Mayora-Ibarra, O., "A Generic Widget Vocabulary for the Generation of Graphical and Speech-Driven User Interfaces." *International Journal of Speech Technology*, 2002. **5**: pp. 39-47.

[12] Ponnekanti, S.R., *et al.* "ICrafter: A service framework for ubiquitous computing environments," in *UBICOMP 2001.* 2001. Atlanta, Georgia: pp. 56-75.

[13] Rosenfeld, R., "Universal Speech Interfaces Web Site," 2002. http://www.cs.cmu.edu/~usi/.

[14] Shriver, S., Toth, A., Zhu, X., Rudnikcy, A., Rosenfeld, R. "A Unified Design for Human-Machine Voice Interaction," in *Extended Abstracts of CHI 2001.* 2001. Seattle, WA: pp. 247-248.

[15] Sukaviriya, P. and Foley, J.D. "Coupling A UI Framework with Automatic Generation of Context-Sensitive Animated Help," in *ACM UIST.* 1990. Snowbird, Utah: pp. 152-166.

[16] Sun, *Jini Connection Technology.* Sun Microsystems, http://www.sun.com/jini/, 2000.

[17] UPnP, "Universal Plug and Play Forum," 2002. http://www.upnp.org.

[18] V2, I., "Information Technology Access Interfaces," 2002. http://www.ncits.org/tc_home/v2.htm.

[19] Zimmermann, G., Vanderheiden, G., Gilman, A. "Prototype Implementations for a Universal Remote Console Specification," in *CHI'2002.* 2002. Minneapolis, MN: pp. 510-511.