Generating Remote Control Interfaces for Complex Appliances

Jeffrey Nichols[*], Brad A. Myers[*], Michael Higgins[†], Joseph Hughes[†], Thomas K. Harris[*], Roni Rosenfeld[*], Mathilde Pignol[*]

[*] School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{jeffreyn, bam, tkharris, roni}@cs.cmu.edu,
mpignol@andrew.cmu.edu
http://www.cs.cmu.edu/~pebbles/puc/

[†]MAYA Design, Inc.
Suite 702
2100 Wharton Street
Pittsburgh, PA 15203
{higgins, hughes}@maya.com

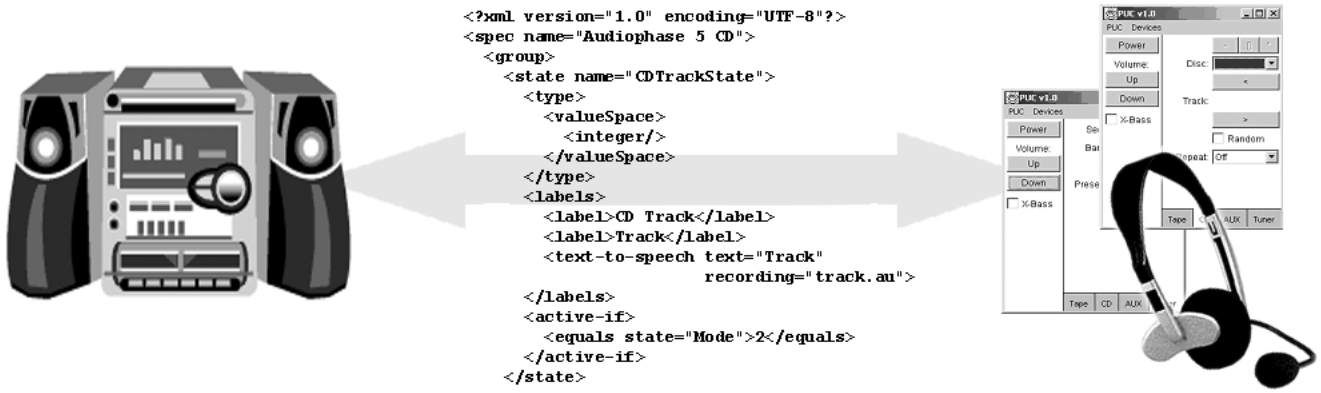


Figure 1. A diagrammatic overview of the personal universal controller system, showing an appliance, a snippet from our specification language, and two graphical interfaces generated from the specification.

ABSTRACT

The *personal universal controller* (PUC) is an approach for improving the interfaces to complex appliances by introducing an intermediary graphical or speech interface. A PUC engages in two-way communication with everyday appliances, first downloading a specification of the appliance's functions, and then automatically creating an interface for controlling that appliance. The specification of each appliance includes a high-level description of every function, a hierarchical grouping of those functions, and dependency information, which relates the availability of each function to the appliance's state. Dependency information makes it easier for designers to create specifications and helps the automatic interface generators produce a higher quality result. We describe the architecture that supports the PUC, and the interface generators that use our specification language to build high-quality graphical and speech interfaces.

Keywords: handheld computers, remote control, appliances, personal digital assistants (PDAs), Pebbles, Universal Speech Interface (USI), personal universal controller (PUC)

INTRODUCTION

Increasingly, home and office appliances, including televisions, VCRs, stereo equipment, ovens, thermostats, light switches, telephones, and factory equipment, are designed with many complex functions, and often come with remote controls. However, the trend has been that as appliances get more computerized with more features, their user interfaces become harder to use [2].

Another trend is that people are increasingly carrying computerized devices, such as mobile phones, pagers, and personal digital assistants (PDAs) such as the Palm Pilot or PocketPC. Many of these devices will soon contain processors powerful enough to support speech applications and hardware for wireless networking. Short-distance radio networks, such as 802.11b and Bluetooth [6], are expected to enable many devices to communicate with other devices that are within close range.

We are investigating how handheld devices and speech can improve the interfaces to home and office appliances, using

an approach we call the *personal universal controller* (PUC). A PUC provides an intermediary interface with which the user interacts as a remote control for any appliance. We envision PUCs running on a variety of platforms, including handheld devices with graphical interfaces or hidden PCs with speech recognition software. The PUC differs from today's universal remote controls, such as the Philips Pronto [17] or the inVoca speech remote [10], because it is self-programming. This means that the PUC engages in a two-way exchange with the appliance, downloading a description of the appliance's functions and then automatically creating a high quality interface. The PUC and the appliance exchange messages as the user interacts with the interface. The two-way communication allows the appliance to update the interface and provide feedback to the user.

Another difference from the Pronto and similar devices is the PUC's ability to provide an interface to the complete functionality of each appliance. Most universal remotes achieve their "universal" nature by providing an interface to only a subset of the most commonly used functions. In our preliminary studies, we have found that doing tasks on a complete PUC interface was still twice as fast as using the actual manufacturer's interfaces and users made one-half the errors.

Using a high-level description to generate interfaces also gives the remote controllers flexibility to choose their interface modality. We have created interface generators that can create graphical and speech interfaces. On a PUC that supports it, this might allow the user to switch freely between multiple interaction modalities, perhaps using a graphical interface to browse a list of songs on an MP3 player and a speech interface to pick a particular one. Multiple PUCs that each supported a different modality could also be used simultaneously to achieve the same effect.

The description of the appliance's functions must have enough information to allow a PUC to generate a high quality user interface, but not contain specifics about how the interface should look or feel. Those choices should be left up to the interface generator. Our specification language contains, like similar systems [16, 29], a hierarchical grouping of all appliance functions that we call the group tree. This tree is determined by the designers of the appliance specifications based upon their understanding of how the appliance's functions relate to each other.

A novel feature of our specification language is that it also includes *dependency information*, which describes the availability of each function relative to the appliance's state. Dependency information is useful for two reasons: 1) it allows the interface to provide feedback to the user about the availability of a function, such as "graying out" a button in a graphical interface, and 2) it helps the interface generators organize functions. Organization improves because dependency information is objective, rather than subjective like the group tree. For example, a graphical interface generator might place sets of functions on overlapping panels because the dependencies indicate they will never be available at the same time. Using a similar technique, the grammar of a speech interface may be simplified because the generator knows that a certain phrase will only make sense when the appliance is in a particular state.

The dependency information also makes it easier for designers to create the group tree because there is no need for the tree to exactly match the structure of the resulting interfaces. The designer can approximately match the structure of the appliance in the group tree and the interface generator can infer the rest from the dependencies. Another benefit of this approach is that it allows the designer to include more structure in the group tree than is necessary for most interfaces, e.g. making the tree deeper with smaller groups. This makes our specification more portable because a PUC generating a graphical interface on a small screen can take advantage of this extra detail, while other devices can safely ignore it by comparing the group tree to the structure inferred from dependencies. We have developed algorithms for generating graphical and speech interfaces by combining dependency information with the group tree.

Our speech interface generator uses Universal Speech Interface (USI) techniques. The USI project [19] at Carnegie Mellon University is creating a standardized interaction style for speech communication between humans and machines. A USI design consists of a few syntactic rules and a handful of application-independent keywords, such as *options*, *status*, *what_is_the*, and *more*. The rules are designed to create a semi-structured interaction style—mnemonic but not necessarily identical to natural language (NL). Unlike full NL interfaces, data collection and skilled grammar creation are not needed, and the small vocabulary and syntactic space ensures high recognition accuracy. Unlike command-and-control speech interfaces, the user need not master any particular application, and can become immediately productive in any new USI application by leveraging their existing knowledge of the universal keywords and rules. Studies have shown that the interaction style can be learned within five minutes of training [21]. Although USI design is guided by analysis of a variety of application types, so far USI designs have been tested primarily in information access applications. The work reported here is the first exploration of USI for device-control applications.

Another important piece of the PUC system is the ability to control actual appliances from the interfaces created by our generators. To enable this, the PUC system provides appliance adaptors: software and hardware that translate from the proprietary communication protocols found on many appliances to our PUC protocol. This architecture has allowed us to use PUCs to control a shelf stereo, camcorders with IEEE 1394 support via the AV/C protocol, the WinAmp media player, a Mitsubishi VCR that supports HAVi [7],

and several others. We are working to support emerging industry standards such as UPnP [26] and JINI [24].

The next section of this paper examines related work. In the sections following we describe the architecture in greater detail, and discuss the specification language and our approach to its design. We continue by describing our communication protocol, followed by a detailed section discussing the graphical and speech interface generators and their use of dependency information. We conclude with thoughts on future directions for this project.

RELATED WORK

A number of research groups are working on controlling appliances from handheld devices. Hodes, *et. al.* propose a similar idea to our PUC, which they call a "universal interactor" that can adapt itself to control many devices [8]. However, their research focuses on the system and infrastructure issues rather than how to create the user interfaces. An IBM project [5] describes a "Universal Information Appliance" (UIA) that might be implemented on a PDA. The UIA uses an XML-based language called MoDAL from which it creates a user interface panel for accessing information. However, the MoDAL processor apparently only handles simple layouts and its only type of input control is text strings. The Stanford ICrafter [18] is a framework for distributing appliance interfaces to many different controlling devices. While their framework supports the automatic generation of interfaces, their paper focuses on hand-generated interfaces and shows only one simple automatically generated interface. They also mention the difficulty of generating speech interfaces.

UIML [1] is an XML language that claims to provide a highly-device independent method for user interface design. UIML differs from the PUC in its tight coupling with the interface. UIML specifications can define the types of components to use in an interface and the code to execute when events occur. The PUC specification language leaves these decisions up to each interface generator.

The XWeb [16] project is working to separate the functionality of the appliance from the device upon which it is displayed. XWeb defines an XML language from which user interfaces can be created. Unlike the PUC specification language, XWeb's language uses only a tree for specifying structural information about an appliance. Their approach seems to work well for interfaces that have no modes, but it is unclear how well it would work for remote control interfaces, where modes are commonplace. XWeb also supports the construction of speech interfaces. Their approach to speech interface design, including emphasis on a fixed language and cross-application skill transference, is quite similar to ours, as it is derived from a joint philosophy [20]. XWeb's language design allows users to directly traverse and manipulate tree structures by speech, however they report that this is a hard concept for users to grasp [16]. Our Universal Speech Interface design differs by trying to stay closer to the way people might talk about the task itself, and is somewhat closer to naturally generated speech.

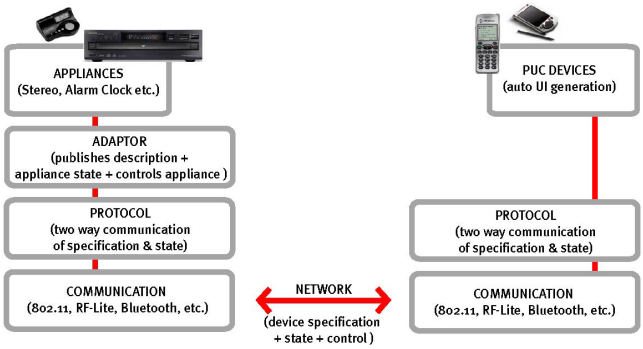


Figure 2. An architectural diagram of the PUC system showing one connection (multiple connections are allowed at both ends).

The INCITS V2 [30] standardization effort is developing the Alternative Interface Access Protocol (AIAP) to help disabled people use everyday appliances with an approach similar to the PUC. AIAP contains a description language for appliances that different interface generators use to create interfaces for both common devices, like the PocketPC, and specialized devices, such as an interactive braille pad. We have begun collaborating with the V2 group and plan to continue this in the future.

A number of research systems have looked at automatic design of user interfaces for conventional computers. These sometimes went under the name of "model-based" techniques [25]. Here, the programmer provides a specification ("model") of the properties of the application, along with specifications of the user and the display. This approach was moderately successful for creating dialog boxes [11, 27] and creating complete interfaces in a limited range [15, 25]. The ITS system from IBM was used to create all the screens for the information kiosks at the EXPO'92 worlds fair [29]. Of particular note is the layout algorithm in the DON system that achieved a pleasing, compact, and logical placement of the controls [11]. We extend these results to create panels of controls on significantly different handhelds, without requiring designer intervention after the interfaces are generated.

ARCHITECTURE

The PUC architecture has four parts: appliance adaptors, a specification language, a communication protocol, and interface generators (see Figure 2). There are several benefits to the design we have chosen:

- Interfaces can be automatically generated.
- The peer-to-peer connection model allows for scalability.

- The transport layer independence between PUCs and appliances makes communication easier.
- Proprietary appliance control protocols can be separated from the PUC framework.

The PUC architecture is designed to allow for automatic interface generation by a wide range of devices in a wide range of modalities. This is enabled by a two-way communication protocol and a specification language that allows each appliance to describe its functions to an interface generator. The specification language separates the appliance from the type of interface that is being used to control it, such as a graphical interface on a handheld or a speech interface on a mobile phone. A later section in this paper describes interface generators that we have created for the graphical and speech modalities.

A key part of the architecture is the network that PUCs and appliances use to communicate. We assume that each appliance has its own facility for accepting connections from a PUC. The peer-to-peer aspect of this choice allows the PUC architecture to be more scaleable than other systems, such as ICrafter [18] and UIA [5], which rely on a central server to manage connections between interfaces and appliances. A PUC could discover appliances by intermittently sending out broadcast requests, as in the service discovery portion of the Bluetooth [6] protocol. However, service discovery has not yet been implemented and is the subject of future work.

We have also attempted to make communication easier by making few assumptions about the underlying transport mechanism. It is unreasonable to expect that every controller device will be able to communicate over the 802.11b wireless Ethernet protocol, for example. The communication protocol, described later, has been designed with a small number of messages to operate over a wide variety of transport layers. The protocol is currently only implemented on TCP/IP, but we have plans to implement on a non-reliable protocol such as UDP, and also on Bluetooth if it becomes widespread.

Although we assume network independence for PUC communication, we cannot make any assumptions about how to communicate with actual appliances. Several standards exist for appliance control, including Microsoft's UPnP [26], Sun's JINI [24], and HAVi [7], but so far none of these have been widely accepted. Many consumer electronics manufacturers have their own proprietary methods for communicating with and between their appliances. Often these methods are not available for use by the general public, although there are Internet groups dedicated to reverse engineering these protocols [9]. Most devices in the low-end market have no means of being controlled externally, except for one-way IR-based remote control. These appliances must be altered at the hardware level to achieve two-way communication with a PUC.

(a) (b)

Figure 3. a) The Aiwa CX-NMT70 shelf stereo and b) the AT&T 1825 telephone/digital answering machine used in our research.

To connect the PUC to any real appliance, we must build an appliance adaptor, i.e. a translation layer to its built-in protocol (see Figure 2). We imagine that an adaptor would be built for each proprietary appliance protocol that the PUC communicates with. For example, we have built a software adaptor for the AV/C protocol that can control most camcorders that support IEEE 1394. We have also built adaptors for connecting to specific devices, such as an Audiophase shelf stereo that we modified with custom hardware to enable two-way communication. We have recently begun constructing an adaptor for the HAVi protocol, and plan to pursue UPnP in the future. This architecture allows a PUC to control virtually any appliance, provided the appliance has a built-in control protocol or someone has the hardware expertise to add one.

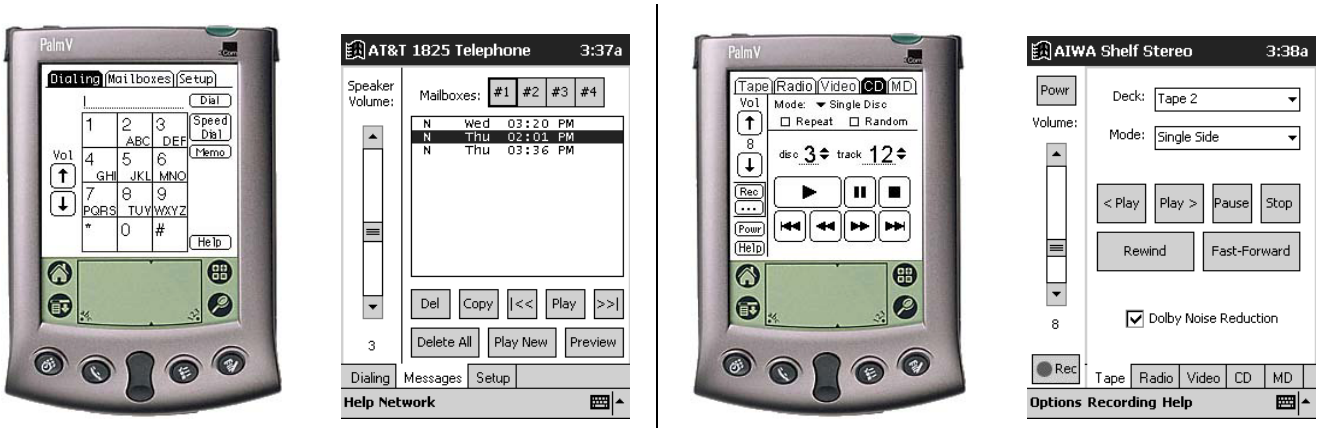
In order to make the construction of new appliance adaptors easy, we have created an adaptor development framework. The framework, implemented entirely in Java, manages all PUC communication for the adaptor, allowing the programmer to concentrate on communicating with the appliance. Using our framework, we implemented adaptors for the X10 protocol and the WinAmp media player in a matter of hours.

SPECIFICATION LANGUAGE

There must be a description of an appliance's functions so the PUC can automatically generate an interface. This description must contain enough information to generate a good user interface, but it should not contain any information about look or feel. Decisions about look and feel should be left up to each interface generator. Further requirements for our specification language are described elsewhere [13].

Approach

As a first step towards determining what information should be included in the specification language, we hand-designed control panels for two appliances. We evaluated them for quality with users, and then extracted the features of these control panels that contribute most to their usability.



(a) (b) (c) (d)

Figure 4. Hand-designed interfaces for the phone (a-b) and stereo (c-d) on the Palm and PocketPC. The Palm interfaces are paper prototypes, whereas the PocketPC interfaces were actually implemented in Microsoft's embedded Visual Basic.

We chose two common appliances as the focus of our hand-designs: the Aiwa CX-NMT70 shelf stereo with its remote control, and the AT&T 1825 telephone/digital answering machine (see Figure 3). We chose these two appliances because both are common, readily available, and combine several functions into a single unit. The AT&T telephone is the standard unit installed in many offices at Carnegie Mellon, and Aiwa-brand stereos seem to be common, at least among the user population that participated in our comparison studies. Ten of our twenty-five subjects owned Aiwa systems.

We created our hand-designed interfaces in two phases, initially on paper and later as Visual Basic implementations on a Microsoft PocketPC. Each interface supported the complete set of appliance functions. At each phase, we iteratively improved the interfaces with heuristic analyses and performed a user study. The user study in each phase was dual-purpose: to compare our hand-designed interfaces with the interfaces on the actual appliances and to see what problems users had with the hand-designed interfaces.

The comparison study in both phases showed that our hand-designed interfaces were much better than the manufacturer's interfaces on the actual appliances [14]. In both studies users were asked to perform a variety of simple and complex tasks. Some simple tasks were dialing the phone or changing the volume on the stereo, whereas some complex tasks were programming a list of tracks into the stereo's CD player or copying a message between two of the four mailboxes on the telephone's built-in answering machine. We found that for both hand-designed interfaces, Palm paper prototypes and PocketPC implementations, users completed tasks in one-half the time and with one-half the errors as compared to the actual appliances [14].

The large differences in this study can be attributed to problems with the appliance interfaces. Most of the problems users had with the built-in appliance interfaces could be traced to poor button labels and inadequate interface feedback. Both appliances had buttons with two functions, one when the button was pressed and released and one when the button was pressed and held. Our subjects rarely discovered the press and hold function. The stereo also had buttons that changed function with the appliance's mode.

Interface Analysis

Once we were confident that our interfaces were usable, we analyzed them to understand what *functional* information about the appliance was needed for designing the interfaces. This included questions such as "why are these elements grouped together?" or "why are these widgets never shown at the same time?" These are questions that might suggest what information should be contained in the specification language.

Language Definition

The PUC specification language is XML-based and includes all of the information that we found in our analysis of the hand-designed interfaces. The language has been fully documented and a DTD is available for validating specifications. The documentation can be downloaded from our project web site:
http://www.cs.cmu.edu/~pebbles/puc/specification.html

State Variables and Commands

Interface designers must know what can be manipulated on an appliance before they can build an interface for it. We discovered from our PocketPC implementations that most of the manipulable elements could be represented as state variables. Each state variable has a given type that tells the interface generator how it can be manipulated. For example, the radio station state variable has a numeric type, and the interface generator can infer the tuning function because it knows how to manipulate a numeric type. Other state variables include the current track of the CD player and the status of the tape player (stop, play, fast-forward, etc.).

After further exploration, we discovered that state variables are not sufficient for describing all of the functions of an appliance. Some elements, such as the seek button on a radio, cannot be represented as state variables. Pressing the seek button causes the radio station state variable to change to some value that is not known in advance. The seek function must be represented as a *command*, a function whose result cannot be described easily in the specification.

Commands are also useful when an appliance is unable to provide feedback about state changes back to the controller, either by manufacturer choice or a hardware limitation of the appliance. In fact, the remote control technology of today can be simulated on the PUC by writing a specification that includes only commands. This is exactly like building a remote control that has only buttons. Any feedback is then restricted to the appliance's front panel.

Type Information

Each state variable must be specified with a type so that the interface generator can understand how it may be manipulated. For example, in Figure 1 the state shown has an integer type. We define seven generic types that may be associated with a state variable:

- boolean
- integer
- fixed point
- floating point
- enumerated
- string
- custom

The most interesting of these types is the custom type, which is provided to allow for the specification of standard widget arrangements, such as the play-stop-pause button groups seen in Figure 4c-d. Each of these button groups represents one state variable, the status of the CD and tape players respectively. Such a widget arrangement presents two problems: the first is that there is a complex mapping desired between the state of the appliance and the interface elements presented to the user; the second is that this complex mapping can and should be reused. Ideally, interface generators will recognize custom types and present a familiar set of interface elements. This is always what happens in the current implementation. It is unreasonable, however, to expect every interface generator to understand every custom type. Therefore we intend to provide a type interrogation feature in our protocol which will involve breaking down a custom type into simpler component types that can be guaranteed to be understood across all interface generators.

Label Information

The interface generator must also have information about how to label the interface components that represent state variables and commands. Providing this information is difficult because different form factors and interface modalities require different kinds of label information. An interface for a mobile web-enabled phone will probably require smaller labels that an interface for a PocketPC with a larger screen. A speech interface may also need phonetic mappings and audio recordings of each label for text-to-speech output. We have chosen to provide this information with a generic structure that we call the *label dictionary*.

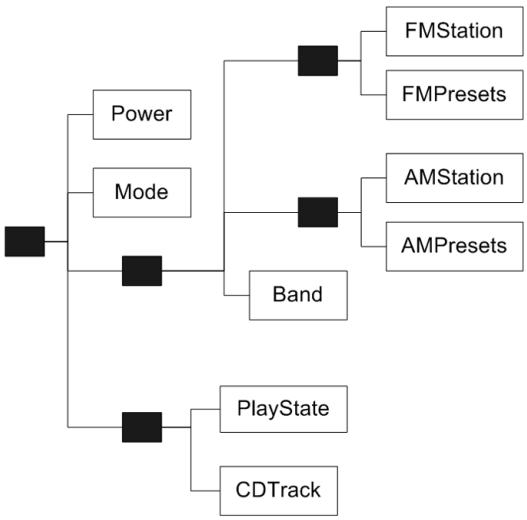


Figure 5. A sample group tree for a shelf stereo with both a CD player and radio tuner. The black boxes represents groups and the white boxes with text represent state variables. The mode variable indicates which source is being played through the speakers.

Each dictionary contains a set of labels, most of which are plain text. The dictionary may also contain phonetic representations using the ARPAbet (the phoneme set used by CMUDICT [3]) and text-to-speech labels that may contain text using SABLE mark-up tags [23] and a URL to an audio recording of the text. The assumption underlying the label dictionary is that every label contained within, whether it is phonetic information or plain text, will have approximately the same meaning. Thus the interface generator can use any label within a label dictionary interchangeably. For example, this allows a graphical interface generator to use a longer, more precise label if there is lots of available screen space, but still have a reasonable label to use if space is tight. Figure 1 shows the label dictionary, represented by the `<labels>` element, for a CD track state with two textual labels and a text-to-speech label.

Group Tree

Interfaces are always more intuitive when similar elements are grouped close together and different elements are kept far apart. Without grouping information, the play button for the CD player might be placed next to the stop button for the Tape player, creating an unusable interface. We avoid this by explicitly specifying grouping information using a group tree.

We specify the group tree as an *n*-ary tree that has a state variable or command at every leaf node (see Figure 5). State variables and commands may be present at any level in the tree. Each branching node is a "group," and each group may contain any number of state variables, commands, and other groups. We encourage designers to make

the group tree as deep as possible, in order to help space-constrained interface generators. These generators can use the extra detail in the group tree to decide how to split a small number of components across two screens. Interface generators for larger screens can ignore the deeper branches in the group tree and put all of the components onto a single panel.

Dependency Information

The two-way communication feature of the PUC allows it to know when a particular state variable or command is unavailable. This can make interfaces easier to use, because the components representing those elements can be disabled. The specification contains formulas (see the `<active-if>` element in Figure 1) that specify when a state or command will be disabled depending on the values of other state variables, currently specified with three types of dependencies: equal-to, greater-than, and less-than. Each state or command may have multiple dependencies associated with it, combined with the logical operations AND and OR. These formulas can be processed by the PUC to determine whether a component should be enabled when the appliance state changes.

We have discovered that dependency information can also be useful for structuring graphical interfaces and for interpreting ambiguous or abbreviated phrases uttered to a speech interface. For example, dependency information can help the speech interfaces interpret phrases by eliminating all possibilities that are not currently available. The processing of these formulas will be described later in the interface generation section.

COMMUNICATION PROTOCOL

The communication protocol enables appliances and PUCs to exchange information bi-directionally and asynchronously. The protocol is XML-based and defines six messages, two sent by the appliance and four sent by the PUC. The PUC can send messages requesting the specification, the value of every state, a change to a particular state, or the invocation of a command. The appliance can send the specification or the current value of a particular state. When responding to a PUC request for the value of every state, the appliance will send a current value message for each of its states. Full documentation for our communication protocol can be downloaded from our project web site:
http://www.cs.cmu.edu/~pebbles/puc/protocol_spec.html

INTERFACE GENERATION

The PUC architecture has been designed to be independent of the type of interface presented to the user. We have developed generators for two different types of interfaces: graphical and speech. This section describes our implementation of these two interface generators, including our algorithms for working with dependency information.

Graphical Interface Generator

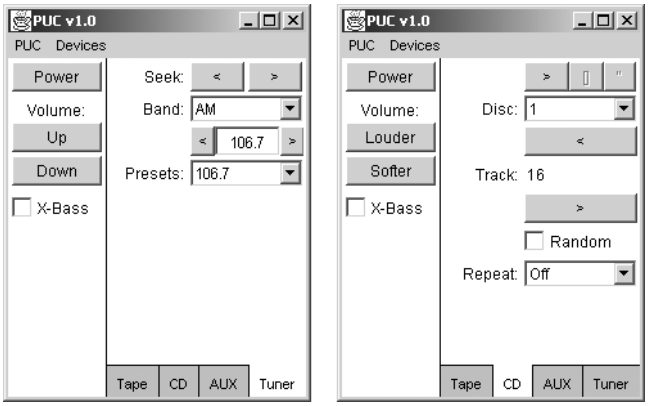
We have implemented a graphical interface generator for the Compaq iPAQ handheld computer using the PersonalJava API. This generator takes an arbitrary description written in our specification language and makes use of the group tree and dependency information to create a high quality interface. The actual UI components that represent each state variable and command are chosen using a decision tree [4]. The components are then placed into panels according to the inferred structure, and laid out using the group tree. The final step of the generation process instantiates the components and places them on the screen.

A key focus of the graphical interface generator is the structure portion of the interface layout. When we looked back at our hand-designed interfaces, we noticed that they could be broken down into small groups of components that were placed relative to each other. For example, Figure 4c shows an interface for the CD player of a shelf stereo. This interface can be broken down into five structural groups: the global functions in the vertical sidebar, the tabs for controlling stereo mode, the less-used CD controls (e.g. Random), the disc and track indicators, and the play controls. Each of these groups has a small number of controls and each control is aligned with the others in their small group, creating a combined layout that is complex. Focusing on the structure portion makes the layout problem solvable and allows us to create complex layouts.

Inferring Panel Structure From Dependencies

The use of different panels can be beneficial to a graphical interface. Commonly-used or global controls can be made available in a sidebar where they are easily accessible. Controls that are only available in the current appliance mode might be placed on a panel in the middle of the screen that changes with the mode, hiding functions from the other modes that are not available.

The graphical interface generator uses dependency information to determine how to divide the screen into panels, and to assign branches of the group tree to each panel. We have found it useful to compare the dependencies of different state variables and commands to determine if they are never available at the same time. If two variables are mutually exclusive, then they might be placed on separate panels. Instead of using dependency information to find mutual exclusion, we could instead have required the specification designers to place markers on all group tree nodes that have mutually exclusive branches. Determining how to place markers requires designers to not only determine what the dependencies are, but then discover all mutually exclusive situations themselves. Instead the designers can enumerate the dependency information and choose a group tree structure that seems intuitive. They can rely on the interface generator to discover relationships within the dependency information and infer panel structure, even if no groups have been specified.



(a) (b)

Figure 6. Interfaces produced by the graphical generator for our Audiophase shelf stereo.

Unfortunately, the task of determining mutual exclusiveness for an arbitrary set of state variables is NP-hard. We reduce the difficulty of the problem by considering mutual exclusion with respect to a single given variable. Our algorithm starts by obtaining a list of all state variables that other commands and states depend upon. Our experience shows that this is a small number of variables for most specifications. We iterate through this list, starting with the state that is most depended upon. Usually the states that are most depended upon represent higher-level modes and we prefer to create high-level structure earlier in the algorithm.

For each state, the algorithm finds the state's location in the group tree and gets a pointer to its parent group. Dependencies on the state are collected from each of the children and are analyzed to find mutual exclusion. If mutual exclusion is found, rules are applied to determine the panel structure to use. We currently have three rules at this level:

1. If the state has an enumerated type and there are mutually exclusive groups of components that are active for each value of the state, create a tabbed panel component with a panel for each value of the enumeration.
2. If the state has a boolean type and all the other states and commands are active for one value of the state, create two full-screen overlapping panels. Each panel has a button for toggling the boolean type. This rule is primarily used for the power button.
3. Create an overlapping panel for every mutually exclusive group and make sure a component exists at a higher level of the tree for changing this state's value.

We plan to create additional rules in the future that create pop-up dialogs or keep all components in the same panel.

Making the Interface Concrete

Once the initial structure has been defined for the interface, our generator can begin making the interface concrete. The first step is choosing what kind of component to assign to each state variable and command. The generator uses a decision tree to make these choices [4]. The decision tree takes into account the following questions when choosing a component:

- Is this a command or a state variable?
- What it the type of the state variable?
- Is the state variable read-only?
- Was a panel structure rule (see above) applied because of this state variable?

For example, in our current system a command is always represented by a button component. Read-only states are almost always represented by a label component. Boolean states are represented by checkboxes, unless the dependency algorithm applied panel structure rule #2 because of this state. In this situation, a button that toggles the state's value is used instead.

Once the components have been selected, the interface generator recursively traverses the group tree and inserts each component into a structure that we call the interface tree. The interface tree represents the panel structure of the generated interface and is used for translating abstract layout relationships to a concrete interface. Each leaf node in the tree is a panel and each branching node specifies how the child panels are placed relative to each other. Branching nodes may represent a set of panels separated by horizontal or vertical edges, or a set of overlapping panels. The particular branching nodes used are chosen by the rules described in the dependency inference section above.

Each panel node in the tree also contains a set of row objects, which describe how components should be placed relative to each other. Most components are placed in the panel using a one-column layout with an adjacent label, but other row layouts are also used. The following rules are used to select different layouts:

- If a group is found that contains a label and two components that do not need their own labels (such as buttons), then a row will be created using the label with the two components in the space typically reserved for a single component. An example is the seek buttons in Figure 6a.
- Some of the components chosen by the decision tree may prefer to occupy the full width of their panel. In this case, a row is created that allows the component to fill the full width of its container, including the space typically reserved for labels.
- If a group is found that contains two components that both need their own labels (such as text fields or selection lists), then a row will be created that has two columns. Each component and its label will be placed in a separate column.

After the group tree has been traversed, the interface is made concrete by recursively traversing the interface tree twice. The first traversal allocates space for each component and determines the size and location of every panel. The second traversal places and sizes the components within their rows, and then assigns labels by picking the largest label that will fit in the space allocated. When this traversal is complete, an interface is presented to the user. Example interfaces generated for controlling our shelf stereo are shown in Figure 1 and Figure 6.

Speech Interface Generator

The speech interface generator creates an interface from a PUC specification, using USI interaction techniques [22].

Generation Procedure

The speech interface generator differs from the graphical interface in that it connects to multiple PUC adaptors (if multiple adaptors can be found on the network), requests the device specifications, and then uses those specifications collectively to configure itself so it can control all devices with the same speech grammar. This includes building a grammar for the parser and a language model and pronunciation dictionary for the recognizer.

The generated grammar is compatible with the Phoenix parser [28], which is used by the USI library to parse user utterances. A grammar is generated for each device that contains phrases for *query* and *control*. Query phrases include determining what groups are available, what states and commands are available, what values the states can take, as well as what values they currently hold. Control phrases allow the user to navigate between groups, issue commands and change states. All of the device-specific grammars, together with a device-independent USI grammar, are compiled into a single combined grammar representing everything that the speech interface will understand. This has been implemented in a test system that is capable of controlling a shelf stereo, a Sony camcorder via the AV/C protocol and multiple X10 devices.

Lessons Learned

We encountered several challenges generating an interface using USI techniques from the PUC specification language. The language makes a distinction between state variables and commands. But what is best described as a variable in a visual interface (e.g. speaker volume) might be better thought of as a command in a spoken interface (e.g. “louder”). Secondly, in a visual environment, groupings of functionalities or widgets need not have a name; such grouping can be implied by adjacency or by a visual cue such as a frame. In speech interfaces, grouping must be assigned a word or phrase if they are to be directly accessed. Occasionally, appropriate names are hard to find, and may not even exist. In the other direction, choosing from a long list of names is easy with speech, yet poses a special challenge to visual interfaces. These challenges are consistent with the observations of others [18] and is a topic of current research.

We have addressed these interface generation challenges in our USI-PUC implementation. Consistent with the graphical interface’s translation from the group tree to the graphical interface tree, the speech interface translates the group tree into a USI-interaction tree. This tree, like the graphical interface generator’s tree, is a more concrete representation of what interactions can take place between the system and the user. The tree consists of nodes with phrasal representations. Each node is either *actionable* or *incomplete*. An actionable node contains a phrase that, when uttered, will cause some device state change. An incomplete node contains a phrase that requires more information before it can be acted upon; uttering the phrase causes the system to prompt the user with completion options, derived from that node’s children. Grouping information is represented in the structure of the tree, and influences exploration, disambiguation, and scope.

FUTURE WORK

There are numerous directions for us to pursue with our work on the personal universal controller in addition to those mentioned previously. There are several outstanding issues with the specification language, and a number of directions in which to take the interface generators.

One problem with the current specification language is that it does not include a list type. Lists are important for many appliances, such as the messages for an answering machine and the songs on an MP3 player. There are many issues to address before the PUC framework can handle lists adequately. One of the most difficult problem is inferring what functions can be performed on the list. Unlike for an integer or boolean state variable, where inferring the possible manipulations is simple, there are many different ways to operate on lists, and it is rare to find a list that uses every one. Some lists, such as a play list for an MP3 player, support the addition and deletion of elements at arbitrary locations. Others do not, such as the answering machine message list shown in Figure 4b, in which new messages are always appended to the end but can be removed from any location. It does not seem reasonable to enumerate all of the possible list operations that might be supported, and then specify which of those operations are supported for each instance of a list on the appliance. We are currently working on a more general solution for this problem.

Finally, one goal of the personal universal controller is to provide consistent interfaces across appliances with similar functions. This requires an interface generator that is adaptive, based upon interfaces that it has created in the past. It must also support some kind of pattern recognition for determining from a specification that two appliances have similar functions. These are both difficult problems that we will be addressing in our future research.

CONCLUSION

We have described the design and architecture of the personal universal controller, a system for automatically generating high-quality multi-modal remote control interfaces for complex appliances. The system includes a two-way communication protocol, adaptors for translating from proprietary appliance protocols to the PUC protocol, a specification language for describing the functions of an appliance, and generators that automatically build interfaces from specifications. A novel element of our system is the use of dependency information for helping generators create high quality interfaces. We have presented generators that use our specification language to create both graphical and speech interfaces.

ACKNOWLEDGMENTS

This work was conducted as a part of the Pebbles [12] project. The speech interface was also conducted as part of the Universal Speech Interfaces project [19]. 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 Mitsubishi Electric Research Laboratories, VividLogic, 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., Phanouriou, C., Batongbacal, A.L., Williams, S.M., and Shuster, J.E. “UIML: An Appliance-Independent XML User Interface Language,” in *The Eighth International World Wide Web Conference*. 1999. Toronto, Canada
2. Brouwer-Janse, M.D., Bennett, R.W., Endo, T., van Nes, F.L., Strubbe, H.J., and Gentner, D.R. “Interfaces for consumer products: "how to camouflage the computer?"” in *CHI'1992: Human factors in computing systems*. 1992. pp. 287-290.
3. CMU, “Carnegie Mellon Pronuncing Dictionary,” 1998. http://www.speech.cs.cmu.edu/cgi-bin/cmudict.
4. de Baar, D.J.M.J., Foley, J.D., Mullet, K.E. “Coupling Application Design and User Interface Design,” in *Conference on Human Factors and Computing Systems*. 1992. Monterey, California: ACM Press. pp. 259-266.
5. Eustice, K.F., Lehman, T.J., Morales, A., Munson, M.C., Edlund, S., and Guillen, M., “A Universal Information Appliance.” *IBM Systems Journal*, 1999. **38**(4): pp. 575-601.
6. Haartsen, J., Naghshineh, M., Inouye, J., Joeressen, O.J., and Allen, W., “Bluetooth: Vision, Goals, and Architecture.” *ACM Mobile Computing and Communications Review*, 1998. **2**(4): pp. 38-45. Oct. www.bluetooth.com.
7. HAVi, “Home Audio/Video Interoperability,” 2002. http://www.havi.org.
8. Hodes, T.D., Katz, R.H., Servan-Schreiber, E., and Rowe, L. “Composable ad-hoc mobile services for universal interaction,” in *Proceedings of ACM Mobicom'97*. Budapest Hungary: pp. 1-12.
9. homeautonz, “Home Automation Webring,” 2002. http://c.webring.com/webring?ring=homeauto;list.
10. inVoca, “inVoca Universal Remote,” http://www.invoca.com.
11. Kim, W.C. and Foley, J.D. “Providing High-level Control and Expert Assistance in the User Interface Presentation Design,” in *Proceedings INTERCHI'93: Human Factors in Computing Systems*. 1993. Amsterdam, The Netherlands: pp. 430-437.
12. Myers, B.A., “Using Hand-Held Devices and PCs Together.” *Communications of the ACM*, 2001. **44**(11): pp. 34-41.
13. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Shriver, S. “Requirements for Automatically Generating Multi-Modal Interfaces for Complex Appliances,” in *ICMI*. 2002. Pittsburgh, PA:
14. 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.
15. Olsen Jr., D.R. “A Programming Language Basis for User Interface Management,” in *Proceedings SIGCHI'89: Human Factors in Computing Systems*. 1989. pp. 171-176.
16. Olsen Jr., D.R., Jefferies, S., Nielsen, T., Moyes, W., and Fredrickson, P. “Cross-modal Interaction using Xweb,” in *Proceedings UIST'00: ACM SIGGRAPH Symposium on User Interface Software and Technology*. 2000. pp. 191-200.
17. Philips, *Pronto Intelligent Remote Control*. Philips Consumer Electronics, 2001. http://www.pronto.philips.com/.
18. Ponnekanti, S.R., Lee, B., Fox, A., Hanrahan, P., and T.Winograd. “ICrafter: A service framework for ubiquitous computing environments,” in *UBICOMP 2001*. pp. 56-75.
19. Rosenfeld, R., “Universal Speech Interfaces Web Site,” 2002. http://www.cs.cmu.edu/~usi/.
20. Rosenfeld, R., Olsen, D., Rudnicky, A., “Universal Speech Interfaces.” *interactions: New Visions of Human-Computer Interaction*, 2001. **VIII**(6): pp. 34-44.
21. Shriver, S., Black, A.W., Rosenfeld, R. “Audio Signals in Speech Interfaces,” in *ICSLP*. 2000.
22. 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. pp. 247-248.
23. Sproat, R., Hunt, A., Ostendorf, P., Taylor, P., Black, A., Lenzo, K., Edgington, M. “SABLE: A Standard for TTS Markup,” in *International Conference on Spoken Language Processing*. 1998. Sydney, Australia:
24. Sun, *Jini Connection Technology*. Sun Microsystems, http://www.sun.com/jini/, 2000.
25. Szekely, P., Luo, P., and Neches, R. “Beyond Interface Builders: Model-Based Interface Tools,” in *Proceedings INTERCHI'93: Human Factors in Computing Systems*. 1993. Amsterdam, The Netherlands: pp. 383-390.
26. UPnP, “Universal Plug and Play Forum,” 2002. http://www.upnp.org.
27. Vander Zanden, B. and Myers, B.A. “Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces,” in *Proceedings SIGCHI'90*. 1990. pp. 27-34.
28. Ward, W. “The CMU Air Travel Information Service: Understanding Spontaneous Speech,” in *DARPA Speech and Natural Language Workshop*. 1990.
29. Wiecha, C., Bennett, W., Boies, S., Gould, J., Greene, S., “ITS: A Tool for Rapidly Developing Interactive Applications.” *ACM Transactions on Information Systems*, 1990. **8**(3): pp. 204-236.
30. Zimmermann, G., Vanderheiden, G., Gilman, A. “Prototype Implementations for a Universal Remote Console Specification,” in *CHI'2002*. 2002. Minneapolis, MN: pp. 510-511.