Informing Automatic Generation of Remote Control Interfaces with Human Designs

Jeffrey Nichols
Human Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891 USA
jeffreyn@cs.cmu.edu
http://www.cs.cmu.edu/~jeffreyn/

ABSTRACT
Embedded processors are making it possible for common appliances, such as cable boxes, microwaves and fax machines, to provide even more functionality. Unfortunately, as these appliances become more complex, their interfaces are also becoming harder to use. At the same time, more people are carrying hand-held computerized devices that can communicate. We envision a future in which people will use their handhelds to communicate with and control common appliances in their environment. In this work we designed a specification language and built an automatic interface generator using lessons learned from analyzing a set of hand-created interfaces.

Keywords
Handheld computers, remote control, appliances, Personal Digital Assistants (PDAs), Pebbles

INTRODUCTION
We are exploring how handheld devices can be used to improve the interfaces for common home and office appliances using an approach we call the *personal universal controller* (PUC). The PUC is similar in concept to the universal remote controls that are available today, with two major differences: 1) it communicates with appliances using a two-way protocol and 2) it is self-programming. The PUC communicates with the appliance that the user would like to control, downloads information about the appliance's functions, and *automatically* generates an interface.

Two things are needed to create this vision of an automatically generated remote control interface: 1) a specification language that can enumerate the functions of the appliance, and 2) an interface generator that can parse the specification language and construct a usable interface from it. We are taking a multi-step approach to realizing these two goals. First, we manually constructed several interfaces by hand and evaluated them for usability [5]. Next, we analyzed these interfaces in order to understand what information about the appliance was needed to build them. Here, we describe what our analysis uncovered and discuss how this was used in the creation of our specification language and interface generator.

HAND-DESIGNED INTERFACES
We designed interfaces for two different appliances, an AIWA CX-NMT70 shelf-stereo and an AT&T 1825 office telephone, on two handheld computer platforms, Palm and PocketPC. We selected these two appliances because they are both multi-functional; the stereo has an integrated radio, tape, and CD player, and the telephone has an integrated answering machine. To be sure that our hand-designed interfaces for these appliances were acceptable for analysis, we conducted an evaluation comparing our interfaces to the physical interfaces provided by the manufacturer. We found that subjects using our interfaces performed tasks in half the time making half as many errors as compared to the manufacturer's interfaces [5].

INTERFACE ANALYSIS
Next we analyzed our hand-designed interfaces to understand the functional information that was needed for their creation. Clearly the PUC requires information about what it can manipulate on the appliance. We found that all the functions of an appliance can be represented by *commands* and *state variables*. For example, on a radio, the station would be represented by a variable with a numeric type. When the remote control wants to change the value of the station variable, it tells the appliance what the new value should be. The tuning function can be inferred from the variable because the interface generator knows how to manipulate a numeric variable. The seek function however, which tells the appliance to scan the radio spectrum until a new station is found, cannot be inferred from the radio station variable. Instead, this function must be represented by a command, because the knowledge needed to find the next clear channel on the spectrum cannot be specified at the time the interface is created (reception will surely change, etc.). The remote control must ask the radio to invoke the seek function, instead of telling it what the new station will be. The appliance would update the station variable on the PUC once it finds the next station.

The two-way communication feature of the PUC allows it to know when a particular function is unavailable. This makes interfaces easier to use, because widgets corresponding to a disabled function can be grayed-out. We have found that we can derive formulas that specify when a function will be disabled depending on the values of other appliance variables. This dependency information is useful for determining the structure of a remote control interface. While this is true for most interfaces, it seems to be especially important for remote controls, which tend to have different modes with different sets of properties. Our hand-designed interfaces use dependency knowledge to make very concise screens with overlapping panels of controls. When the user picks a particular mode, the panel with the controls for that mode is shown while the controls for other modes are hidden. The use of dependencies is one of the novel techniques that our generator uses to construct user interfaces.

SPECIFICATION LANGUAGE

Before an interface can be generated, an appliance specification is needed. Our specification language is XML-based, and includes all of the information that we found in our analysis of the hand-designed interfaces. It also includes information for handling other problems that we did not experience with our hand-designed interfaces but anticipated encountering with the interface generator.

The generator must be equipped to handle different types of displays, including devices with different sizes than the hand-held platforms we looked at. Although dependency information is very useful for determining the structure of an interface, it may not be sufficient if a mode has many properties and the screen only has space for a small number of controls. To handle this difficulty, the specification includes a *group tree*, a hierarchical grouping of the state variables and commands that make up the functions of an appliance. Variables that must be placed together in the interface will be leaves of the same node, whereas controls that are not related will not be found in the same branch. A priority can also be attached to each variable to help the generator understand which elements are more commonly used. The interface generator can inspect the group tree to determine how to split the controls into sub-groups. For example, less important widgets might be placed into a dialog box.

INTERFACE GENERATOR

The interface generator takes a specification written in our language and creates an interface from it. The current version is implemented in Java 1.1.8 and runs on the Compaq iPAQ handheld device as well as desktop machines. The generator currently creates structure from dependency information and then assigns controls to variables and commands based upon a decision tree algorithm [2]. Figure 1 shows three different screens of an interface generated by our system for a shelf stereo appliance. In this stereo, all functions depend on the power being on. The left-most screen shows the interface when the power is off. The two screens on the right show the state of the interface in the Radio and CD playback modes. The top halves of these two screens show mode-independent controls and the bottoms show different sets of controls that are appropriate for the mode shown.

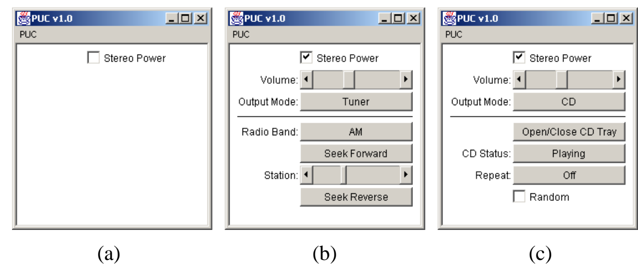

Figure 1. Three screens of a stereo interface created by our generator software. a) The screen shown when the power is off. b) The screen shown when the power is on and the radio is being used. c) The screen shown when the power is on and the CD player is being used.

FUTURE WORK

Work remains to be done on our interface generator. Specifically, we have not yet handled situations where there are too many controls to fit on a screen. We are also working on the aesthetic issues of interface design by exploring algorithms that can iteratively refine designs according to aesthetic properties. Simulated annealing or other iterative hill-climbing methods may be a part of our approach.

ACKNOWLEDGMENTS

This work was conducted as a part of the Pebbles project [4], directed by Brad A. Myers. It was funded 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 and under Grant No. IIS-0117658. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of the National Science Foundation.

REFERENCES

1. Brouwer-Janse, M.D. *et al.* “Interfaces for consumer products: ‘how to camouflage the user?’” *CHI’1992*: Human factors in computing systems. Monterey, CA. May 3-7, 1992. pp. 287-290.
2. de Baar, D., *et al.* "Coupling Application Design and User Interface Design," *CHI'1992*, May 3-7, 1992. Monterey, CA. pp. 259-266.
3. Haartsen, J. *et al*. “Bluetooth: Vision, Goals, and Architecture,” *ACM Mobile Computing and Communications Review*. 1998. **2**(4). pp. 38-45. Oct. www.bluetooth.com
4. Myers, B.A. “Using Hand-Held Devices and PCs Together,” *Communications of the ACM*. November, 2001. **44**(11): pp. 34-41.
5. Nichols, J. “Using Handhelds as Controls for Everyday Appliances,” *CHI’2001 Extended Proceedings*. Seattle, WA. March 31-April 5, 2001. pp. 443-444.