Controlling Home and Office Appliances with Smartphones

Jeffrey Nichols and Brad A. Myers

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891
(412) 268-5150
FAX: (412) 268-1266
{jeffreyn, bam+}@cs.cmu.edu
http://www.cs.cmu.edu/~pebbles/puc/

Abstract

Today most of the home and office appliances that we interact with contain microprocessors. All of these appliances have some user interface, but many users become frustrated with the difficulty of using the complex functions of their appliances. We are developing a framework that allows users to interact with appliances through a separate user interface device that they are already carrying. Smart phones are good candidates for providing interfaces because they are common, have communication capabilities to allow connection to appliances, and are already being used for a wide range of different applications. Our framework includes an abstract specification language for describing appliances, a two-way communication protocol, and automatic interface generation software that allows user interfaces to be customized to users and the devices they are using. This article overviews our *personal universal controller* system and describes in detail our design and implementation of automatic interface generation for Microsoft's Smartphone platform.

Keywords:

Personal Digital Assistants (PDAs), handheld devices, smart phones, mobile phones, mobile devices, Palm Pilots, PocketPC, remote controls, appliances, Pebbles, Personal Universal Controller (PUC).

Introduction

Increasingly, home and office appliances, including televisions, VCRs, stereo equipment, refrigerators, washing machines, thermostats, light switches, telephones, copiers, and factory equipment, have embedded computers, and often come with remote controls. However, the trend has been that as appliances get more computerized with more features, their user interfaces get harder to use [Brouwer-Janse 1992]. The Wall Street Journal reports that "appliances – TVs, telephones, cameras, washing machines, microwave ovens – are getting harder [to use]…. The result is a new epidemic of man-machine alienation" [Gomes 2003].

At the same time, it is becoming common for people to carry a smart phone that has better input-output capabilities than the average home appliance, such as high-resolution screens, text-entry technologies, and speech capabilities. Phones are likely to maintain this advantage over appliances, because improved hardware is a key differentiator between phones and is often marketed as an incentive to upgrade to a new phone. All phones also come with the ability to communicate over the cellular networks, and most have built-in short range communication capabilities, such as Bluetooth, that could allow them to communicate with and control appliances in their surrounding environment. Phones are also personal devices, which allow them to provide interfaces that are personalized. For example, a phone could provide interfaces that are consistent with previous appliance interfaces that the user has seen, or it might combine multiple appliance interfaces to create a single interface organized around tasks rather than appliances. Combining interfaces also deals with the familiar problem of needing a table full of remote controls for a home entertainment system.

We see future phones being the preferred mode of interaction with many appliances, because the phone is always available and can provide a better user interface with its improved hardware. There is precedent for people using their phones to remotely control their environment. The Salling Clicker [Salling 2005] is popular for controlling applications on the Macintosh from a Bluetooth-enabled smart phone, and RuttenSoft's Media Remote [RuttenSoft 2005] allows a phone to remotely control Windows Media Player on a desktop. In late 2002, NEC introduced a mobile phone in Japan that contains an infrared transceiver that allows the phone to act as a universal remote control [Agilent 2003].

If phones are to act as remote controls, then user interfaces will be needed, but where will these user interfaces come from? There are too many different kinds of appliances for each phone

manufacturer to provide hand-designed interfaces for each appliance on every phone. Appliance manufacturers could store pre-designed remote control user interfaces on each appliance, but there are too many different kinds of phones to provide a different interface for each. Appliances could also provide web-style interfaces that are rendered in the built-in web browser that most phones ship with (an approach used by UPnP [UPnP 2005] and others), but web-style interfaces do not support the level of interaction that people want to have with their appliances. For example, a web interface cannot support an interactive slider that adjusts a value in real-time, such as you might want for the volume on a stereo. Most phone web browsers also suffer from poor rendering of pages, which could lead to low quality user interfaces.

In this paper, we present a framework for automatically generating appliance interfaces from abstract specifications of appliances' functions that are stored on the appliances. These interfaces allow control of the full functionality of each appliance and are generated to be consistent with other interfaces that are provided on the phone. This allows users to leverage their existing knowledge of their phone to control appliances. Our interfaces are also fully interactive, which enables real-time incremental adjustment of the appliance's features, such as volume.

An important focus of our system is to generate high quality interfaces, but creating high quality user interfaces on a smart phone is challenging. The screens on existing phones are small and typically capable of displaying only 8-10 lines of text at one time. This results in a hierarchical list-based design for most user interfaces, which can become unwieldy when the list is longer than can be displayed on one screen. Another problem is that users have to remember their location in the hierarchy in order to navigate to any function. Our system addresses these problems with innovative rules that optimize each list screen and limit the depth of the hierarchy while maintaining the natural structure of the appliance. This results in an interface that is easy to search and requires a small number of navigation steps by the user. Our system also maintains the look-and-feel of the phone interfaces without introducing new interaction techniques so that our generated interfaces are consistent with other applications on the phone.

Related Work

There has been work on improving the interfaces for consumer electronics and work on automatically generating interfaces for phones, but as far as we know this work has never before been combined.

Omojokun et al. [Omojokun 2005] have collected usage data for consumer electronics in real home settings and applied a machine learning approach to discover the core set of functionality that is used by a particular user and to cluster these functions into task groups. They compared their automatic results to their users' intuition and discovered that neither approach was sufficient for building a complete user interface. In the future they propose to explore a mixed approach that combines automatic and user-oriented approaches to design user interfaces. Our approach differs because our interfaces include the full functionality for each appliance rather than a subset containing the most commonly used functions. In the future, we are interested in applying Omojokun's work to optimize the organization of our user interfaces to favor commonly used functions while still including the remaining functions.

DiamondHelp [Rich 2005] combines a task-based dialog interface with a direct manipulation interface to bring usability and consistency to consumer electronics interfaces. The interface is designed for display on a large screen in the home, such as a television or personal computer, and uses two-part design that would be difficult to adapt to today's mobile phones. The task-based portions of the user interface are automatically generated from task models, but the direct manipulation portions are currently hand-designed. The unique aspect of DiamondHelp is its combination of two different interface styles, which allows users to choose how to interact with the appliance while benefiting from structured support.

The automatic generation of user interfaces, also known as model-based user interface development, has a rich history of building interfaces for general computer applications [Szekely 1996]. Recently, researchers have explored how to apply this knowledge to mobile platforms like phones and PDAs. SUPPLE [Gajos 2004] uses optimization techniques to automatically generate WAP-based interfaces for phones with built-in browsers, though phone interfaces are not the focus of their system. These WAP-based interfaces would have the same problems for appliances as web-based interfaces (as described above). Eisenstein et al. [Eisenstein 2001] used their XIML description language to transform desktop interfaces into interfaces for mobile devices, including a phone. Their approach uses a set of transformation rules, many of which can be applied automatically, though the final interfaces usually need to be tuned by a designer.

Personal Universal Controller

The work presented in this paper is built within our personal universal controller (PUC) system, which is designed to allow users to control appliances in their environment through a remote user

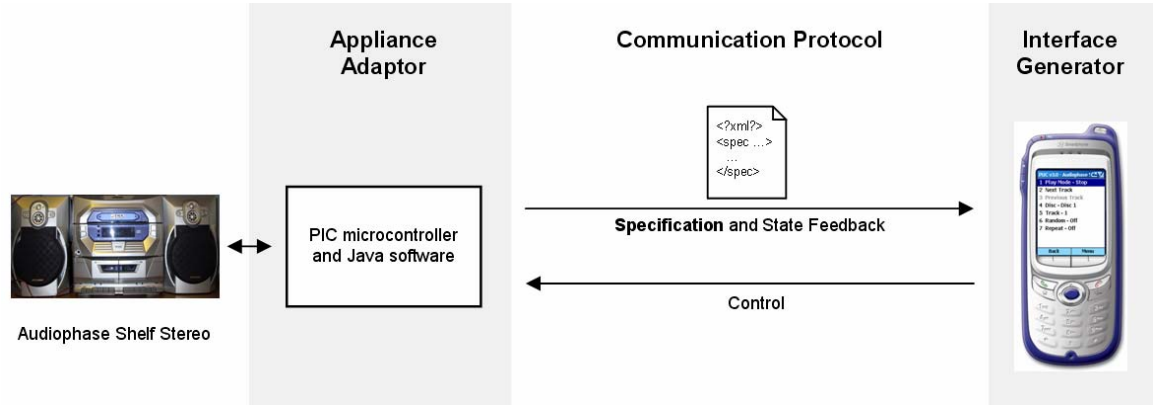


Figure 1. The PUC architecture, showing an appliance and the four parts of the PUC system: appliance adaptors, a communication protocol, specification language, and interface generators. Note that the PUC system uses a peer-to-peer connection model, allowing multiple interface generators to connect to one appliance and an interface generator to connect to multiple appliances.

interface. When a user decides to control an appliance, the controller device downloads an abstract functional description from that appliance and uses that description to automatically generate an interface for controlling that appliance. A two-way communication channel between the controller and the appliance allows the user's commands to be sent to the appliance and feedback to be provided to the user. We have explored the use of graphical interfaces on PDAs and speech interfaces in the past [Nichols 2002], but this paper is the first time that we have described generating graphical interfaces for a smart phone.

The PUC system has four parts: a specification language, a communication protocol, appliance adaptors, and interface generators (see Figure 1). Automatic generation of user interfaces is enabled by the specification language, which requires each appliance to describe its functions in an abstract way. The goal in designing this language was to include enough information to generate a good user interface, but not include any specific information about look or feel. Decisions about look and feel are left up to each interface generator. Included in the language are state variables and commands to represent the functions of the appliance, a hierarchical "group tree" to specify organization, dependency information that defines when states and commands are available to the user based on the values of other states, and multiple human-readable strings for each label in a specification.

One goal of the system is to control real appliances. Since there are no appliances available that natively implement the PUC protocol, we have built appliance adaptors. Adaptors are translation layers that are built between the PUC protocol and the appliance's proprietary protocol. A number of appliance adaptors have already been built, including a software adaptor for the AV/C

protocol that can control most camcorders that support IEEE 1394 and another adaptor that controls Lutron lighting systems. Hardware adaptors have also been built for appliances, such as the Audiophase shelf stereo shown in Figure 1, that do not natively support any communication protocol. We have also created simulators for appliances that we did not have easy access to, including an elevator and the Driver Information Console (DIC) in a GMC Yukon Denali SUV (see Figure 2). We have also experimented with building general purpose adaptors to industry standards, such as UPnP and HAVi.

The last, but most important, piece of the PUC architecture is the interface generator. Interface generators have been built on several different platforms, including graphical interface generators on PocketPC, Microsoft's Smartphone (described here), and desktop computers, as well as a speech interface generator that uses the Universal Speech Interfaces framework [Rosenfeld 2001].

Smartphone Interface Generation

We have created a smartphone interface generator using Microsoft's Windows CE-based Smartphone platform. The platform is a set of hardware requirements for OEMs and a Windows CE-based operating system that runs on top of compliant hardware. The hardware platform requires a 220x176 screen without touch-sensitivity. Interaction takes place through a 4-way directional pad, a normal phone keypad, home and back buttons, and two soft buttons with labels that are shown on the Smartphone's screen. We have implemented our generator software in C# using the .NET Compact Framework, which allows us to reuse some of the parsing and infrastructure code from our PocketPC interface generator. Only a few of the interface generation rules could be shared from the PocketPC however, because the Smartphone has a dramatically different user interface style.

Our Smartphone interface generator creates interfaces that follow Microsoft's Smartphone user interface guidelines. We chose this approach so that our interfaces would be consistent with other Smartphone applications, allowing users to leverage their knowledge of their Smartphone to control appliances. The guidelines stipulate that most interfaces should use a list-based hierarchy that leads to summary panes for viewing data or editing panes for modifying data. Our generator follows these guidelines and focuses on optimizing the structure of the lists so the hierarchy is shallow and each list requires only one screen. Figure 2 shows our generated interface for the DIC in a GMC Yukon Denali DIC with each level of hierarchy. The interface generator tries to keep as much of the interface in the list format as possible, but sometimes a variable cannot be

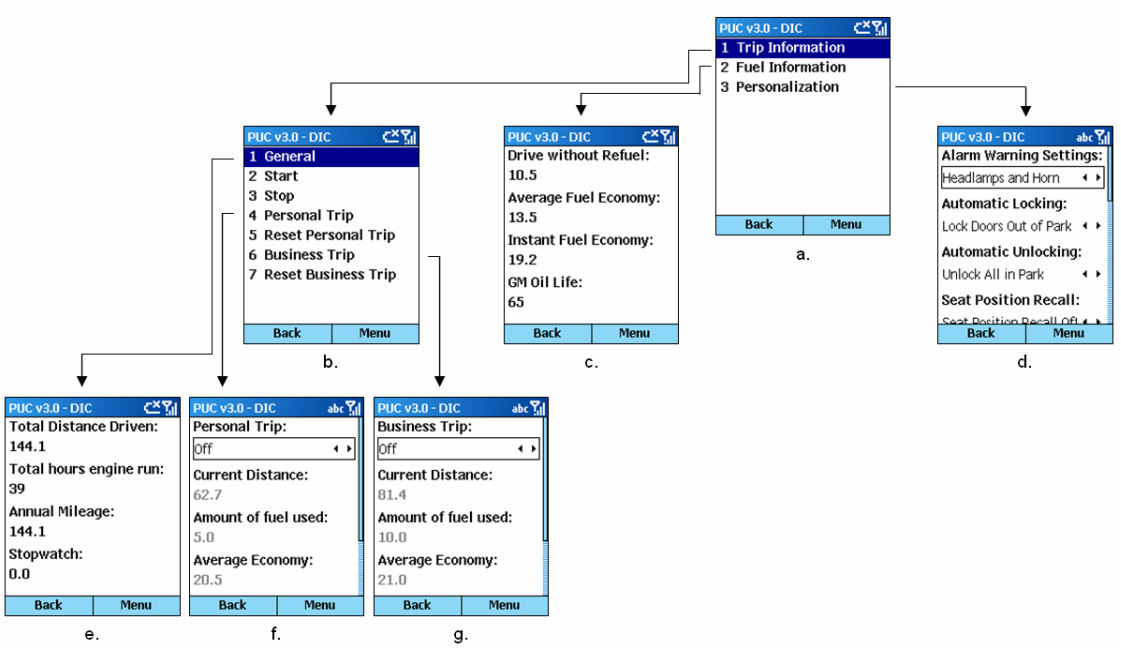


Figure 2. The automatically generated interface for the Driver Information Center in a 2003 GMC Yukon Denali SUV. The user navigates through list panes (a,b) to get to summary (c,e) and editing panes (d,f,g).

manipulated within the constraints of the list. In this case, an editing pane is created that contains the appropriate controls. Our interface generator may also create summary panes when a number of read-only state variables are grouped together. Note that the list hierarchy created by the interface generator is static, so that users can learn the hierarchy over time and remember where commonly used functions are.

The user interface guidelines also state that the left soft button should always be used for invoking the most commonly used function for a given interface. We explored one static method and one adaptive method for choosing this function and found trade-offs between the approaches.

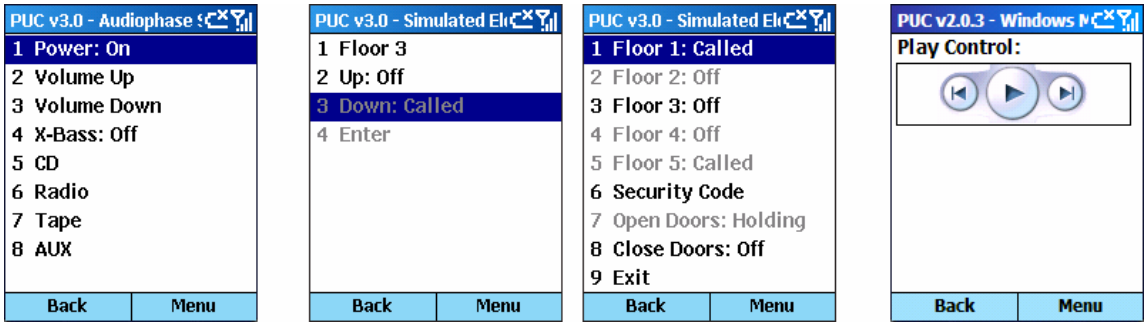
List-Based Interface Generation Rules

The list-based structure of the Smartphone interfaces leads to several unique design challenges for our Smartphone interface generator. The most important challenge is to make the hierarchical list structure intuitive to the user so that functions can be found quickly, while at the same time minimizing the number of different screens that make up each generated interface. The number of editing panes also must be minimized, especially to prevent situations in which only one control is on an editing pane. A part of minimizing editing panes is deciding whether a particular variable should be manipulated through a list item or a control on an editing pane. A challenge to all of this is to make navigation quicker without significantly violating the structure described in the

appliance specification. A final challenge is deciding which function to assign to the left soft button, which is supposed to invoke the most commonly used function on the current screen.

Creating an intuitive list hierarchy is one of the most important challenges for our Smartphone interface generator, because users will be unable to interact with an appliance if they cannot find the functions they want to use. The Smartphone generator uses a combination of information from a specification's group tree, dependencies, and labels to create an intuitive list hierarchy for users. We start by analyzing the dependency information in order to find sets of functions that are not available at the same time. The group tree, which is already a hierarchical structure of the appliance's functions, is modified based on the sets that are found to ensure that mutually-exclusive sets are separated into different groups. The changes to the group tree are marked so that the list building process can take appropriate action.

The list hierarchy is then built from the modified group tree. Starting with the top-most group in the tree, a list is constructed by making each child group that is labeled into a child list. Every state variable and command that is encountered is added to the list as an item. Groups are not required to have labels, so not all groups in the specification will have corresponding child lists in the user interface. This may mean that lists are created that are larger than can be shown on the screen at once, but we have rules that will attempt to address this problem later in the generation process. If a mutually-exclusive set of functions is encountered, then usually no additional action is required because of the changes already made to the group tree. This results in each set of functions being available from a separate list that is accessed from the same parent list (see Figure



a. b. c. d.

Figure 3. Example screens from automatically generated Smartphone interfaces. (a) The opening screen for controlling a shelf stereo. Our dependency information rule created the separate lists for CD, Radio, etc. (b)-(c) Two screens from a simulated elevator interface. The particular screen shown to the user depends on whether the user is (b) outside or (c) inside the elevator car. (d) A Smartphone rendering of the media-controls Smart Template from an interface for controlling the Windows Media Player application on a desktop computer. The template's design is based on the Smartphone Windows Media Player application, and is operated using the right, left, and select buttons of the phone's thumb stick.

3a). In the case where the user cannot choose which set of functions is enabled through the interface, such as when the appliance has a read-only mode, the interface generator may create overlapping lists that are switched based on the state of the appliance (see Figure 3b,c).

Optimizing the list structure for navigation ensures that users spend less time finding features in the interface and more time using those features. The challenge of optimizing is balancing the structure that has already been built with the constraints of Smartphone user interfaces.

There are two constraints of the Smartphone interface that need to be addressed:

- Navigation is particularly important in Smartphone interfaces because only nine items can be shown on each list screen and users constantly navigate up and down the list hierarchy. This means that the Smartphone interface generator should try to make the depth of the list hierarchy as shallow as possible and place the maximum number of functions onto each screen. The list structure must still reflect the properties of the appliance however, and should not deviate significantly from the initial structure.

- Editing panes are necessary in the Smartphone interface because many functions cannot be manipulated in the list. For example, a state variable with an enumerated type might be edited with a combo box or slider, neither of which is supported in a Smartphone list interface. Other functions can only be instantiated as list items because there is not a corresponding control that can be used on an editing pane. Commands, such as “Seek,” are good example of this, because the Smartphone does not allow on-screen buttons such as those used to invoke commands in our PocketPC interfaces.

The Smartphone interface generator optimizes navigation using a rule-based approach. The rules are applied iteratively during a depth-first traversal of the list hierarchy. Rules are also applied bottom-up, so the rules are applied to all of the children of a list before being applied to that list. The children of each list are traversed in “priority”-order, which is a measure of importance that the specification author defines for each function and group in the appliance specification. Traversing in this way ensures that the rules have more flexibility for optimizing the most important functions of an interface.

We currently have five rules for optimizing navigation, which are applied in the order discussed here. Each looks for a particular set of features in the list hierarchy and makes some change to the list if that set of features is found. Some of these rules make decisions about whether a particular

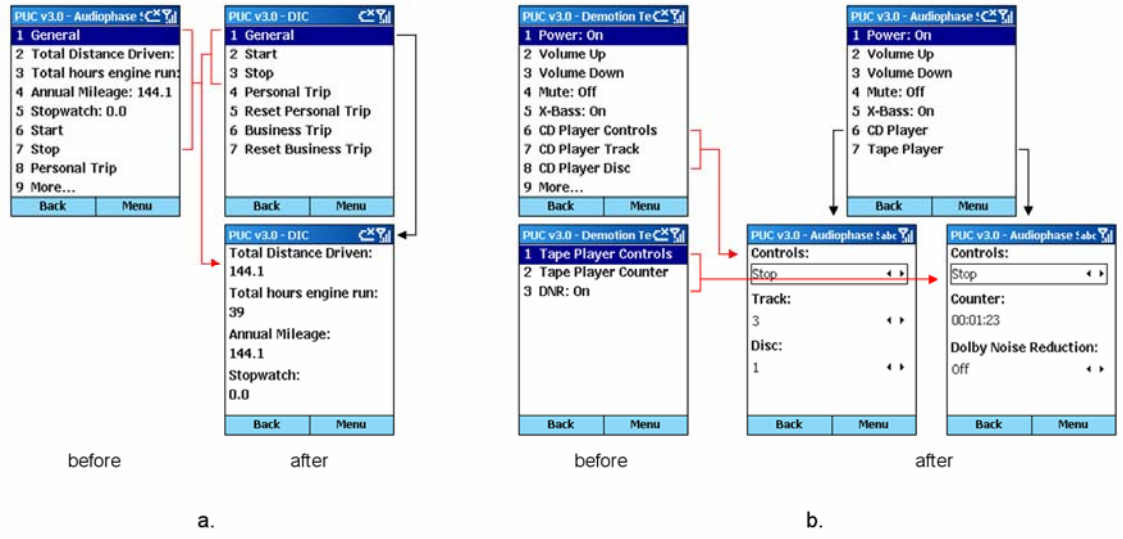


Figure 4. Diagrams showing how the first (a) and second (b) rules for optimizing the list structure behave. Black arrows indicate how the screens are connected and red lines indicate changes made by the rules. Note that in (a) some items were list-only and thus were promoted to the top-level list while the others were placed on a panel. The items all happen to be labels, so this panel is a summary pane.

function will be displayed as a list item or as a control on an editing pane. During this discussion, functions that can only be displayed in a list will be called “list-only items.” Functions that must be displayed on an editing pane will be called “panel-only items,” and all other functions will be called “list-or-panel items.”

The first two rules minimize the number of editing panes that may be accessed from the current list. Neither of these rules is applied if the current list contains only one panel-only item. The first of these rules searches for situations where the number of empty slots in the parent list is greater than the number of list-only items. If this is found, then all of the list-only items are promoted into the parent list. The current list is then replaced with an editing pane and the remaining items are placed on that pane (see Figure 4a). Note that this causes any panel-or-list items to be displayed on the editing pane. This has the side-effect of occasionally creating summary panes when all of the list-or-panel items are labels (see Figure 2c,e and Figure 4a).

The second rule searches for situations where there is more than one panel-only item. If this is found, then the generator looks for sets of panel-only items that have labels with a common prefix or suffix. For each set that is found, an editing pane is created and the items in the set are placed on it. The list item that opens the editing pane is labeled with the common portion of the label associated with that set (see Figure 4b). We originally considered having an additional rule that moved all panel-only items onto a single editing pane if no sets were found and labeling the item

that opened the pane with the label of the parent group concatenated with the term “Controls.” We decided against this rule however because we believed the user would have a hard time guessing what functions were on the panel given this label, and because the navigation cost in terms of the number of key presses for giving each panel-only item its own editing pane is not much different than having a panel of unrelated controls.

The third rule looks for any remaining panel-only and list-or-panel items that have not been assigned to an editing pane. Every list-or-panel item is assigned to a list, and each remaining panel-only item is given its own editing pane, as discussed above.

Now that all of the editing panes have been created and every item has been assigned to an editing pane or a list, we can now optimize the number of items in a list. The fourth and fifth rules are very similar to the first and second rules, except that they manipulate only list items. The fourth rule eliminates unneeded child lists by moving all of their items into the parent list if there is enough room. This rule always promotes the most important items first because the list hierarchy is being traversed in priority order. The fifth rule tries to break up lists that have more than the nine items that can be shown at once on the screen. The method for doing this uses common label prefixes and suffixes, just like the second rule. Child lists are created in reverse priority order until the current list contains nine items or less.

We have experimented with several different methods for **assigning a function to the “most common” soft button.** Initially we used this button to move up in the list hierarchy, which duplicated the functionality of the physical “back” button. This helped novice phone users navigate our interface, but we felt that it might be more useful to assign common functions from the appliance to the button instead. We investigated two approaches: a static approach using priority information from the appliance specification and an adaptive approach based on recorded usage information.

Our first method chooses a function for each screen by ranking each of the functions on that screen according to the priority information in the specification language. If there is a tie, we choose the function that occurs first in the appliance specification. One function is chosen for each screen, and these functions do not change once the interface is built.

The second method is adaptive, which means that the function assigned to the soft button changes as the user interacts with the interface. We select the function by searching the recorded usage information for the most likely next function from the last function that was used. If there is no

usage information, we use the algorithm from the first method to select the function. We currently change the soft button every time the screen changes or the user invokes a function, but we plan to experiment with other times. Unlike with the first approach, “back” may be assigned to the soft button if the usage information suggests that the next thing the user is likely to do is move up in the hierarchy.

We have not conducted any formal evaluation of either of these methods. The non-adaptive approach has the advantage that users can memorize the function that is assigned to it as they use the interface, but the priority information in our specification is not always reliable and does not always pick the right function. For example, the power button is picked on the main screen of our shelf stereo though in fact this is not a function that seems to be used very often. The adaptive approach would seem to fix this problem because it relies on actual usage data, but the cognitive load of keeping track of which function is currently assigned to the button seems too high. It seems to usually be faster to remember the keypad shortcut for each function rather than to read the label on the soft button. It may be that the adaptive approach becomes beneficial after using the interface for a significant period of time, but we do not currently have any regular users who can verify this.

Both methods also suffer from the small area available for the label on the soft button. In many cases it is not possible to display a sufficient label in the space provided on the interface, particularly when both the name and value of a function need to be shown. One solution might be to use icons, but our system currently does have any way for a specification author to include icons as a label for functions.

Shared Generation Techniques

We were also able to apply two of the techniques that we use for generating interfaces on the PocketPC to the Smartphone. The first technique uses the dependency information in our specification language, which defines when each state variable and command is available in terms of other state variables. We have found that many appliances have modes that prevent some functions from being accessed at the same time as other functions. For example, on many shelf stereos, when one of the audio sources (tape, radio, CD, etc.) is selected, none of the features of the other sources can be manipulated. As discussed in the previous section, the Smartphone generator takes these modes into account when building its initial list hierarchy (see Figure 3a-c). This structure would not have been found if the interface generator had relied only on the grouping information in the appliance specification. Furthermore, the generator would not know

to change the interface when the appliance state changes without the use of dependency information.

The second shared technique is called Smart Templates [Nichols 2004], which addresses the problem of automatically generating interfaces that conform to domain-specific design patterns. For example, automated tools would not otherwise produce the standard layout for entering a street address on a navigation system or use standard icons for play, stop, and pause on a media player. Smart Templates allow pre-programmed design knowledge to co-exist with automatic interface generators. Interface generator builders and specification authors decide in advance on the meaning of high-level tags that may be added to groups and variables in a specification. For example, a group tagged with our media-controls Smart Template must contain either several commands for “Play,” “Stop,” and “Pause,” or a “Mode” state with an enumerated type containing each of those labels. Optionally, the group may also contain commands for “previous track” and “next track” for media players and “play new” for answering machines. Figure 3d shows a Smartphone rendering of this template, which mimics the interface for the Smartphone version of Windows Media Player.

Using Smart Template tags in a specification requires the specification author to adhere to the pre-defined restrictions on the contents of the group or parameters of the variable. If an interface generator understands a tag in a specification, then it is able to produce a device-specific rendering for that Smart Template based on the contents of the tagged group or variable. If the tag is not recognized, the interface generator can still produce an interface for the Smart Template because the contents are specified using the primitive elements of our specification language. We have specified a number of Smart Templates, including date, time-absolute, time-duration, address, media-controls, and many others.

Conclusions

As computing becomes more pervasive, there will be more and more computerized appliances in our environment that we will want to control. A platform is needed that allows users to use the devices they have at hand, such as their mobile phones, in order to control their appliances. In this article we have shown that it is possible to automatically generate interfaces for controlling appliances on a smart phone. Using smart phones as remote controls for appliances makes sense because these phones have better user interface hardware than most appliances, the ability to communicate, and a high likelihood of being available when a user needs to control an appliance.

Our approach does have some limitations. It is difficult to automatically generate interfaces for appliances that have a lot of data, such as a calendaring appliance, because there is significant user expectation about how the data will be displayed that cannot be easily described in an appliance specification or rendered by an interface generator. Fortunately, we are able to generate interfaces for most of the appliances that we have encountered because most appliances do not have so much data that they are subject to this limitation. We also believe that any limitations of this system are offset by the advantages of being able to generate interfaces that are customized to users and the devices they prefer to use. We are now working on enhancements to our system that will ensure that newly generated interfaces take into account previous interfaces with which the user has interacted. We are also planning to conduct user studies of our generated interfaces to see how they compare with manufacturers' interfaces on existing appliances.

Author Bios

Jeffrey Nichols is a doctoral student in the Human-Computer Interaction Institute in Carnegie Mellon University's School of Computer Science. He is the lead researcher on the Personal Universal Controller project, exploring how handheld computers can improve the usability of household and office appliances. He received a BS degree in computer engineering from the University of Washington in 2000.

Address: Human Computer Interaction Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3891. jeffreyn@cs.cmu.edu, http://www.cs.cmu.edu/~jeffreyn/

Brad A. Myers is a Professor in the Human-Computer Interaction Institute in the School of Computer Science at Carnegie Mellon University. He is the author or editor of over 270 publications, and he is on the editorial board of five journals. He has been a consultant on user interface design and implementation to over 50 companies, and regularly teaches courses on user interface design and software. In 2004, he was elected to the CHI Academy, an honor bestowed on the principal leaders of the field, whose efforts have shaped the discipline and led the research in human-computer interaction. Myers received a PhD in computer science at the University of Toronto. He received the MS and BSc degrees from the Massachusetts Institute of Technology during which time he was a research intern at Xerox PARC. From 1980 until 1983, he worked at PERQ Systems Corporation. His research interests include user interface development systems, user interfaces, hand-held computers, programming by example, programming languages for kids, visual programming, interaction techniques, window management, and programming

environments. He belongs to SIGCHI, ACM, IEEE Computer Society, IEEE, and Computer Professionals for Social Responsibility.

Address: Human Computer Interaction Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3891. bam@cs.cmu.edu, http://www.cs.cmu.edu/~bam

Acknowledgements

This work was funded in part by grants from NSF, Microsoft, General Motors, and the Pittsburgh Digital Greenhouse, and equipment grants from Mitsubishi Electric Research Laboratories, VividLogic, Lutron, and Lantronix. The National Science Foundation has 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 no necessarily reflect those of the National Science Foundation.

References

[Agilent 2003] Inc. Agilent. *NEC selects Agilent Technologies' IR transceiver with remote control capability for full-color display mobile phone*. Palo Alto, 2003. http://www.agilent.com/about/newsroom/presrel/2003/07may2003a.html.

[Brouwer-Janse 1992] Maddy D. Brouwer-Janse, Raymond W. Bennett, Takaya Endo, Floris L. van Nes, Hugo J. Strubbe and Donald R. Gentner. "Interfaces for consumer products: "how to camouflage the computer?"" *CHI'1992: Human factors in computing systems,* Monterey, CA, May 3 - 7, 1992. pp. 287-290.

[Eisenstein 2001] Jacob Eisenstein, Jean Vanderdonckt and Angel R. Puerta. "Applying model-based techniques to the development of UIs for mobile computers," *Intelligent User Interfaces,* Santa Fe, 2001. pp. 69-76.

[Gajos 2004] Krzysztof Gajos and Daniel S. Weld. "SUPPLE: Automatically Generating User Interfaces," *Intelligent User Interfaces,* Funchal, Portugal, 2004. pp. 93-100.

[Gomes 2003] Lee Gomes. "Appliances Have Become Like PCs: Too Complex for Their Own Good," *The Wall Street Journal OnLine.* May 12, 2003. http://www.pebbles.hcii.cmu.edu/puc/localmedia/wsj-20030512.pdf.

[Nichols 2002] J. Nichols, Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Pignol, M. "Generating Remote Control Interfaces for Complex Appliances," *UIST 2002,* Paris, France, 2002. pp. 161-170. http://www.cs.cmu.edu/~pebbles/papers/PebblesPUCuist.pdf.

[Nichols 2004] J. Nichols, Myers, B.A., Litwack, K. "Improving Automatic Interface Generation with Smart Templates," *Intelligent User Interfaces,* Funchal, Portugal, 2004. pp. 286-288.

[Omojokun 2005] Olufisayo Omojokun, Jeffrey S. Pierce, Charles L. Isbell Jr. and Prasun Dewan. "Comparing End-User and Intelligent Remote Control Interface Generation," *To appear in The Third Conference on Appliance Design (3AD),* 2005.

[Rich 2005] Charles Rich, Candy Sidner, Neal Lesh, Andrew Garland, Shane Booth and Markus Chimani. "DiamondHelp: A Graphical User Interface Framework for Human-Computer Collaboration," *5th International Workshop on Smart Appliances and Wearable Computing (IWSAWC),* 2005. p. To Appear.

[Rosenfeld 2001] R. Rosenfeld, Olsen, D., Rudnicky, A. "Universal Speech Interfaces," *interactions: New Visions of Human-Computer Interaction.* 2001. **VIII**(6). pp. 34-44.

[RuttenSoft 2005] Inc. RuttenSoft. *Media Remote 2004 for Smartphone 2003*. 2005. http://www.ruttensoft.com/smartphone/mediaremote.htm.

[Salling 2005] Jonas Salling. *Salling Clicker*. Salling Software. 2005. http://homepage.mac.com/jonassalling/Shareware/Clicker/.

[Szekely 1996] P. Szekely. "Retrospective and Challenges for Model-Based Interface Development," *2nd International Workshop on Computer-Aided Design of User Interfaces,* Namur, Namur University Press. June 5-7, 1996. pp. 1-27.

[UPnP 2005] UPnP. *Universal Plug and Play Forum*. **2005.** 2005. http://www.upnp.org.