Controlling Home and Office Appliances with Smart Phones

The Personal Universal Controller system lets users interact with their appliances through automatically generated remote control interfaces on their smart phone devices.

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 come with remote controls. However, as appliances become more computerized with more features, their user interfaces tend to become harder to use. The Wall Street Journal reports that “the result is a new epidemic of man-machine alienation.”

At the same time, people are carrying smart phones with better I/O capabilities than the average home appliance, such as high-resolution screens, text-entry technologies, and speech capabilities. Phones will likely 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. Phones can also communicate over cellular networks, and most have built-in short-range communication capabilities (such as Bluetooth) that could let them communicate with and control appliances in their environment. Additionally, most phones have only one user, making it easy for the phone to provide personalized interfaces. For example, a phone can provide interfaces that are consistent with appliance interfaces that the user is familiar with, or it might combine multiple appliance interfaces to create a single interface organized around tasks rather than appliances. Combin-
Related Work on Building User Interfaces

Researchers have conducted work on improving consumer electronics’ interfaces and on automatically generating interfaces for phones. To our knowledge, these areas have never before been combined.

Oluﬁ sayo Omojokun and his colleagues collected usage data for consumer electronics in real home settings and applied a machine learning approach to discover a particular user’s core set of functionality and to cluster these functions into task groups.1 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 exploring a mixed approach that combines automatic and user-oriented approaches to design user interfaces. Our approach differs because our interfaces include full functionality for each appliance rather than a subset containing the most commonly used functions. In the future, we’re interested in applying Omojokun and his colleagues’ work to optimize our user interfaces’ organization to favor commonly used functions while still including the remaining functions.

DiamondHelp combines a task-based dialog interface with a direct manipulation interface to increase consumer electronics interfaces’ usability and consistency.2 The interface is designed for display on a large screen in the home, such as a television or PC, and uses a two-part design that would be difﬁcult to adapt to today’s mobile phones. The user interface’s task-based portions are automatically generated from task models, but the direct manipulation portions are hand designed. DiamondHelp’s unique aspect is its combination of two interface styles, which lets users 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.3 Recently, researchers have explored how to apply this knowledge to mobile platforms such as phones and PDAs. SUPPLE uses optimization techniques to automatically generate Wireless Application Protocol-based interfaces for phones with built-in browsers although phone interfaces aren’t their system’s focus.4 These WAP-based interfaces would have the same problems for appliances as Web-based interfaces. Jacob Eisenstein and his colleagues used their eXtensible Interface Markup Language (XIML) to transform desktop interfaces into interfaces for mobile devices, including a phone.5 Their approach uses a set of transformation rules, many of which can be applied automatically, although a designer must usually tune the final interfaces.

Our motivation

We see future phones being the preferred mode of interaction with many appliances because the phone is always available and its improved hardware can provide a better user interface. There’s precedent for using phones to remotely control the environment. The Salling Clicker (www.salling.com/Clicker/windows) is popular for controlling Macintosh applications from a Bluetooth-enabled smart phone, and RuttenSoft’s Media Remote (www.ruttensoft.com/smartphone/mediaremote.htm) lets a phone remotely control Windows Media Player on a desktop. In late 2002, NEC introduced a mobile phone in Japan containing an infrared transceiver that lets the phone act as a universal remote control.5

If phones are to act as remote controls, they’ll need user interfaces—but where will these user interfaces come from? The many different kinds of appliances prohibit each phone manufacturer from providing hand-designed interfaces for each appliance on every phone. Appliance manufacturers could store predesigned 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 universal plug and play (www.upnp.org) and others—but Web-style interfaces don’t support the interactivity that people want with their appliances. For example, a Web interface can’t support an interactive slider that adjusts a value.

References

1. O. Omojokun et al., “Comparing End-User and Intelligent Remote Control Interface Generation,” Personal and Ubiquitous Computing, vol. 10, no. 2, 2006, pp. 136–143.
2. C. Rich et al., “DiamondHelp: A Collaborative Interface Framework for Networked Home Appliances,” Proc. 5th Int’l Workshop on Smart Appliances and Wearable Computing, IEEE C5 Press, 2005, pp. 514–519.
3. P. Szekely, “Retrospective and Challenges for Model-Based Interface Development,” Proc. 2nd Int’l Workshop Computer-Aided Design of User Interfaces, Namur Univ. Press, 1996, pp. 1–27.
4. K. Gajos and D.S. Weld, “SUPPLE: Automatically Generating User Interfaces,” Proc. 9th Int’l Conf. Intelligent User Interfaces (UI’04), ACM Press, 2004, pp. 93–100.
5. J. Eisenstein, J. Vanderdonckt, and A.R. Puerta, “Applying Model-Based Techniques to the Development of UIs for Mobile Computers,” Proc. 6th Int’l Conf. Intelligent User Interfaces (UI’01), ACM Press, 2001, pp. 69–76.
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.

**Personal Universal Controller**

The work we present in this article is built within our Personal Universal Controller (PUC) system, which we designed to let users control appliances in their environment through a remote user interface. When a user decides to control an appliance, the controller device downloads an abstract functional description from the appliance and uses that description to automatically generate an interface. A two-way communication channel between the controller and the appliance sends the user’s commands to the appliance and provides feedback. We’ve explored using graphical interfaces on PDAs and speech interfaces in the past, but this is the first time that we’ve described generating graphical interfaces for a smart phone.

The PUC system has four parts: a specification language, a communication protocol, appliance adapters, and interface generators (see figure 1). The specification language enables automatic generation of user interfaces and requires each appliance to describe its functions abstractly. Our 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. We leave decisions about look and feel to each interface generator. Included in the language are state variables and commands to represent the appliance’s functions, a hierarchical “group tree” to specify organization, dependency information that defines when states and commands are available to the user on the basis of the other states’ values, and multiple human-readable strings for each label in a specification.

One goal of the PUC is to control real appliances. Because no appliances natively implement the PUC protocol, we build appliance adapters. Adapters are translation layers that bridge the gap between the PUC protocol and the appliances’ proprietary protocols. We have built several appliance adapters, including a software adapter for the AV/C protocol that can control most camcorders that support IEEE 1394 and another adapter that controls Lutron lighting systems. We have also built hardware adapters for appliances that don’t natively support any communication protocol, such as the Audiophase shelf stereo in figure 1. We also created simulators for appliances that we didn’t have easy access to, including an elevator and the driver information console in a GMC Yukon Denali sport utility vehicle (see figure 2). We’ve also experimented with building general-purpose adapters to industry standards, such as universal plug and play and home audio video interoperability (HAVi).

The 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, and desktop computers, as well as a speech interface generator that uses the Universal Speech Interfaces framework.

**Smart phone interface generation**

We created a smart phone interface generator using Microsoft’s Windows CE-based Smartphone platform. The platform is a set of hardware requirements for original equipment manufacturers and a Windows CE-based operating system. The hardware platform requires a 220 × 176-pixel screen without touch sensitivity. Interaction takes place through a four-way directional pad, a keypad, home and back buttons, and two soft buttons with
labels that are shown on the screen. We implemented our generator software in C# using the .NET Compact Framework, which lets us reuse some of our PocketPC interface generator’s parsing and infrastructure code. We could use only a few interface generation rules from the PocketPC, however, because the Smartphone has a dramatically different user interface style.

Our 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 smart phone applications, letting users leverage their knowledge of their smart phone to control appliances. The guidelines stipulate that most interfaces should use a list-based hierarchy that leads to summary panes for viewing data or to editing panes for modifying data. Our generator follows these guidelines and focuses on optimizing the lists’ structure so the hierarchy is shallow and each list requires only one screen. Figure 2 shows our generated interface for a GMC Yukon Denali driver information console with each hierarchy level. The interface generator keeps as much of the interface in the list format as possible, but sometimes a variable can’t be manipulated within the list’s constraints. In this case, the generator creates an editing pane containing the appropriate controls. Our interface generator might also create summary panes when a number of read-only state variables are grouped together. The list hierarchy the interface generator creates is static, so 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 invoke 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 Smartphone interface’s list-based structure presents several unique design challenges for our smart phone interface generator. The most important challenge is to make the hierarchical list structure intuitive so that the user can find functions quickly while minimizing the number of different screens that make up each generated interface. We must also minimize the number of editing panes, especially to prevent situations where only one control is on an editing pane. 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.

Creating an intuitive list hierarchy. This is important because users won’t be able to interact with an appliance if they can’t 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. The generator starts by analyzing the dependency information to find sets of functions that aren’t available at the same time. It then modifies the group tree, which is already a hierarchical structure of the appliance’s functions, on the basis of the sets that are found to ensure that mutually exclusive sets are separated into different groups. The generator marks the changes to the group tree so that the list-building process can take appropriate action.

The generator then builds the list hierarchy from the modified group tree. Starting with the topmost group in the tree, it constructs a list by making each labeled child group into a child list. The generator adds every state variable and command that it encounters to the list as an item. Groups don’t have to have labels, so not all groups in the specification will have corresponding child lists in the user interface. This might mean that some lists will be larger than the screen can show at once, but our rules will attempt to address this problem later in the generation process. If the interface generator encounters a mutually exclusive set of functions, usually no additional action is necessary 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 3a). If the user can’t choose which set of functions is enabled through the interface—such as when the appliance has a read-only mode—the interface generator can create overlapping lists that switch on the basis of the appliance’s state (see figure 3b and 3c).

Optimizing list structure for navigation. This ensures that users spend less time finding features in the interface and more time using those features. The challenge is balancing the existing structure within the constraints of the Smartphone’s user interface guidelines. The interface generator must address two constraints of the Smartphone interface:

- Navigation is particularly important in Smartphone interfaces because each list screen can show only nine items and users constantly navigate up and down the list hierarchy. So, the interface generator should make the list hierarchy as shallow as possible and should place the maximum number of functions onto each screen. The list structure must still reflect the appliance’s properties, however, and shouldn’t deviate significantly from the initial structure.
- Editing panes are necessary because many functions can’t 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 the Smartphone list interface supports. Other functions can only be instantiated as list items because the Smartphone doesn’t have a corresponding control that can be placed on an editing pane. Commands such as “seek” are a good example of this, because the Smartphone doesn’t allow onscreen buttons such as those used to invoke commands in our PocketPC interfaces.

The smartphone interface generator optimizes navigation using a rule-based approach. The generator applies the rules iteratively during a depth-first traversal of the list hierarchy. It also applies rules bottom-up, so it applies the rules to all of a list’s children before applying them to that list. Each list’s children 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 an interface’s most important functions.

We have five rules for optimizing navigation, which the generator applies in the order discussed here. Each rule
looks for a particular set of features in the list hierarchy and makes some change to the list if it finds that set of features. Some rules make decisions about whether a particular function will be displayed as a list item or as a control on an editing pane. Throughout this article, we call functions that can only be displayed in a list list-only items. Functions that must be displayed on an editing pane are panel-only items, and all other functions are list-or-panel items.

The first two rules minimize the number of editing panes that the user can access from the current list. Neither rule is applied if the current list contains only one panel-only item. The first rule searches for situations where the parent list has more empty slots than list-only items. If this occurs, all the list-only items are promoted into the parent list. An editing pane replaces the current list, and the remaining items are placed on that pane (see figure 4a). This causes any list-or-panel items to be displayed on the editing pane. This also has the side effect of occasionally creating summary panes when all the list-or-panel items are labels (see figures 2c, 2e, and 4a).

The second rule searches for situations with more than one panel-only item. If this occurs, the generator looks for sets of panel-only items that have labels with a common prefix or suffix. For each set the generator finds, it moves the items in the set onto a new editing pane. The list item that opens the new 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 parent group’s label concatenated with the term controls. We decided against this, however, because the user would have trouble guessing what functions were on the panel given this label. Also, the navigation cost in terms of the number of key presses for giving each panel-only item its own editing pane isn’t much different than having a panel of unrelated controls.

The third rule looks for any remaining panel-only and list-or-panel items that haven’t been assigned to an editing pane. The rule moves every list-or-panel item to a list and every remaining panel-only item to its own editing pane, as we discussed earlier.

Now that all editing panes have been created and every item has been assigned to an editing pane or a list, we can optimize the number of items in a list. The fourth and fifth rules are similar to the first and second rules, except that they manipulate only list items. The fourth rule eliminates unneeded child lists by moving all their items into the parent list if there’s enough room. The fifth rule tries to break up lists that have more than nine items because the screen can’t display them all at once. The method for doing so uses common label prefixes and suffixes, just like the second rule. It creates child lists in reverse priority order until the current list contains nine items or fewer.
Assigning a function to the “most common” soft button. We’ve experimented with several different methods for doing this. Initially, we used this button to move up in the list hierarchy, duplicating the physical back button’s functionality. This helped novice phone users navigate our interface, but we thought 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 function on that screen according to the specification language’s priority information. If there’s a tie, we choose the function that occurs first in the appliance specification. One function is chosen for each screen, and these functions don’t change once the interface is built.

The second method is adaptive, meaning that the soft button’s function 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 used. If there’s no usage information, we use the algorithm from the first method to select the function. We change the soft button every time the screen changes or the user invokes a function, but we plan to experiment with other techniques. Unlike the first approach, the adaptive approach may select the “back” function if the usage information suggests that the user is likely to move up in the hierarchy next.

We haven’t formally evaluated either method. The nonadaptive approach’s advantage is that users can memorize its function as they use the interface, but our specification’s priority information isn’t always reliable and doesn’t always pick the right function. For example, the nonadaptive approach picks the shelf stereo’s power button although this isn’t a commonly used function. 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 usually seems faster to remember each function’s keypad shortcut rather than to read the soft button’s label. Perhaps the adaptive approach becomes beneficial after using the interface for a significant period of time, but we don’t have any regular users who can verify this yet.

Both methods also suffer from the small area available for the soft button’s label. In many cases, it’s not possible to display a sufficient label on the interface, particularly when both a function’s name and value must be shown. One solution might be to use icons, but our system doesn’t yet have any way for a specification author to include icons as a label for functions.

Our work demonstrates the feasibility of automatically generating high-quality interfaces for a wide range of appliances on a smart phone device.

Shared generation techniques
We also applied two techniques that we use for generating interfaces on the PocketPC to the Smartphone. The first uses the dependency information in our specification language;4 dependency information defines when each state variable and command is available in terms of other state variables. Many appliances have modes that prevent users from accessing some functions at the same time as other functions. For example, on many shelf stereos, when you select an audio source (for example, tape), you can’t manipulate any features of the other sources (for example, radio or CD). As mentioned earlier, the Smartphone generator considers these modes when building its initial list hierarchy (see figures 3a–3c). This structure wouldn’t have been found if the interface generator had relied only on the grouping information in the appliance specification. Furthermore, the generator wouldn’t know to change the interface when the appliance state changes without using dependency information.

The second shared technique is Smart Templates,8 which addresses the problem of automatically generating interfaces that conform to domain-specific design patterns. For example, automated tools wouldn’t 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 let preprogrammed design knowledge coexist with automatic interface generators. Interface generator builders and specification authors decide in advance on the meaning of high-level tags that can 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 might 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 predefined restrictions on the group’s contents or the variable’s parameters. If an interface generator understands a tag in a specification, it can produce a device-specific rendering for that Smart Template based on the tagged group or variable’s contents. If it doesn’t recognize the tag, the interface generator can still produce an interface for the Smart Template because the contents are specified using our specification language’s primitive elements. We’ve specified several Smart Templates, including date, time-absolute, time-duration, address, media-controls, and many others.

Our work demonstrates the feasibility of automatically generating high-quality interfaces for a wide range of appliances on a smartphone device. Our approach does have some limitations, however. For example, it’s difficult to automatically generate interfaces for appliances that have data that users expect to be displayed in a particular way, such as calendar data, because user expectations can’t be easily described in an appliance specification or rendered by an interface generator. Fortunately, we can generate interfaces for the appliances we’ve encountered because most appliances don’t have data that is 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’ve recently enhanced our system to ensure that newly generated interfaces consider previous interfaces the user has interacted with and to combine functionality from multiple appliances into a single task-based user interface. We also plan to conduct user studies to see how our interfaces compare with existing appliance interfaces.

ACKNOWLEDGMENTS

This work was funded in part by grants from the NSF, Microsoft, General Motors, and the Pittsburgh Digital Greenhouse and by equipment grants from Mitsubishi Electric Research Laboratories, VividLogic, Lutron, and Lantronix. The NSF funded this work through a Graduate Research Fellowship for Jeffrey Nichols and under Grant No. IIIS-0117658.

REFERENCES

1. D. Maddy, “Interfaces for Consumer Products: How to Camouflage the Computer?” Proc. SIGCHI Conf. Human Factors in Computing Systems (CHI 92), ACM Press, 1992, pp. 287–290.

2. L. Gomes, “Appliances Have Become Like PCs: Too Complex for Their Own Good,” The WallStreetJournal Online, 2003; www.pebbles.hcii.cmu.edu/puc/localmedia/wsj-20030512.pdf.

3. J. Nichols, B.A. Myers, and B. Rothrock, “UNIFORM: Automatically Generating Consistent Remote Control User Interfaces,” Proc. SIGCHI Conf. Human Factors in Computing Systems (CHI 06), ACM Press, 2006, pp. 611–620.

4. J. Nichols et al., “Huddle: Automatically Generating Interfaces for Systems of Multiple Connected Appliances,” to be published in Proc. 19th Ann. ACM Symp. User Interface Software and Technology (UIST 06), ACM Press, 2006.

5. “NEC Selects Agilent Technologies’ IR Transceiver with Remote Control Capability for Full-Color Display Mobile Phone,” Agilent, 2003; www.agilent.com/about/newsroom/presrel/2003/07may2003a.html.

6. J. Nichols et al., “Generating Remote Control Interfaces for Complex Appliances,” Proc. 15th Ann. ACM Symp. User Interface Software and Technology (UIST 02), ACM Press, 2002, pp. 161–170.

7. R. Rosenfeld Jr., D. Olsen, and A. Rudnick, “Universal Speech Interfaces,” New Visions of Human–Computer Interaction, vol. 8, no. 6, 2001, pp. 34–44.

8. J. Nichols, B.A. Myers, and K. Litwack, “Improving Automatic Interface Generation with Smart Templates,” Proc. 9th Int’l Conf. Intelligent User Interfaces (IUI 04), ACM Press, 2004, pp. 286–288.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.