Automatically Generating High-Quality User Interfaces for Appliances

Jeffrey Nichols

December 2006
CMU-HCII-06-109

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

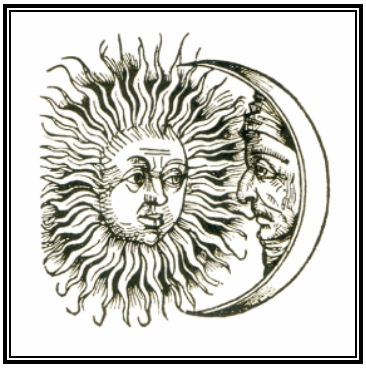
Thesis Committee:
Brad A. Myers, Chair
Scott E. Hudson
John Zimmerman
Dan R. Olsen, Jr., Brigham Young University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was supported in part by the National Science Foundation (through the author's Graduate Research Fellowship and grants IIS-0117658 and IIS-0534349), Microsoft, General Motors, the Pittsburgh Digital Greenhouse, and Intel. Equipment supporting this research was generously donated by Mitsubishi Electric Research Laboratories, VividLogic, Lantronix, Lutron, IBM Canada, Symbol Technologies, Hewlett-Packard, and Lucent. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any sponsoring party or the U.S. Government.

Keywords: Automatic interface generation, aggregate user interfaces, handheld computers, personal digital assistants, mobile phones, home theater, appliances, personal universal controller (PUC), user interface description languages (UIDLs), remote controls, multi-modal interfaces, speech recognition, Smart Templates, user interface consistency, personal consistency, familiarity

Automatically Generating High-Quality User Interfaces for Appliances



Jeffrey Nichols

Abstract

In this dissertation, I show that many appliance usability problems can be addressed by moving the user interface from the appliance to a handheld device that the user is already carrying, such as a Personal Digital Assistant (PDA) or mobile phone. This approach, called the Personal Universal Controller (PUC), takes advantage of the increasing pervasiveness of wireless communication technologies that will allow handheld devices to communicate directly with appliances. Automatic generation of the appliance user interface allows the PUC to create interfaces that are customized to the platform of the controller device, the user's previous experience, and all the appliances that are present in the user's current environment.

This dissertation makes several contributions to the state of the art in automatic interface generation:

- Automatic generation that makes use of dependency information to determine a better structure for the generated user interfaces
- The general Smart Templates technique for incorporating domain-specific design conventions into an appliance specification and automatically rendering the conventions appropriately on different platforms and in different interface modalities
- Algorithms that apply knowledge of similarity between specifications and interfaces the user has previously seen to generate new interfaces that are *personally consistent*
- Algorithms that use a model of the content flow between appliances to generate task-based interfaces that combine functionality from multiple appliances

An evaluation of the PUC system compared the automatically generated interfaces for two all-in-one printers with the manufacturer's interfaces for the same two appliances and found that users of the automatically generated interfaces were twice as fast and four times as successful for both common and complex tasks. The evaluation also shows that the PUC's consistency features allow users to be twice as fast when using a new appliance that is similar to an appliance they have previously encountered. This evaluation is the first known user study of automatically generated interfaces compared to human designs, and it shows that automatic generation of user interfaces for end users is now viable for interactive systems.

For my parents

Table of Contents

List of Figures

List of Tables

Acknowledgements

Many people have touched my life, helped me find direction, and kept me on the path to finishing this thesis. I have tried to acknowledge as many of these people below as I could, and my sincerest apologies to anyone I have accidentally forgotten.

I would first like to thank Brad Myers for his guidance and advice throughout my time at Carnegie Mellon. Brad taught me what research is, how to do it, and how to present the results in a clear and understandable fashion. I am still amazed that, even at his busiest, Brad was always able to find time to read one of my papers and give me an enormous amount of feedback on its content. I only hope that I can be half as successful as a researcher and a mentor in my career as Brad has been in his.

I would like to thank the other members of my thesis committee, Scott Hudson, John Zimmerman, and Dan Olsen, for providing invaluable comments along the way that shaped the direction of this work.

I would also like to thank Scott Hudson for all of his efforts in organizing the HCII's Ph.D. program, and for putting up with all of the student feedback that I brought to him during my time as the student ombudsperson. Scott was always willing to help address our problems, and I do not think the program would be half as strong without his presence.

I was lucky to have the assistance of many great individuals over the course of PUC project. In particular, I would like to thank Brandon Rothrock for assisting with several aspects of the system implementation, Duen Horng "Polo" Chau for helping with the design of the flow-based interface and the evaluation of the PUC interfaces discussed in Chapter 10, Kevin Litwack for building adaptors for many real world appliances and helping find problems with an early version of the specification language, and Michael Higgins and Joseph Hughes from Maya Design for helping with the initial design of the specification language and the initial implemenation of the PUC server-side infrastructure. I would also like to thank Thomas K. Harris, Stefanie Tomko, and Roni Rosenfeld for adopting the PUC technologies as the basis for their speech interface generator. Many thanks also to Htet Htet Aung, Mathilde Pignol, Rajesh Seenichamy, and Pegeen Shen for their contributions to developing the language documentation, writing specifications, and building appliance adaptors.

I am equally indebted to the corporate and government sponsors who generously funded my research or provided equipment: the National Science Foundation, the Pittsburgh Digital Greenhouse, Microsoft, General Motors, Intel, Mitsubishi Electric Research Labs, Lantronix, Lutron, and VividLogic.

Many others have influenced this work, including other students of Brad's, other students in the HCII Ph.D. program, and students and faculty doing related work at other universities. They are: Daniel Avrahami, Laura Dabbish, James Fogarty, Darren Gergle, Andy Ko, Desney Tan, Luis von Ahn, Jake Wobbrock, Gregory Abowd, Sonya Allin, Anupriya Ankolekar, Ryan Baker, Gaetano Borriello, Anind Dey, Jodi Forlizzi, Krzysztof Gajos, Gary Hsieh, Pedram Keyani, Queenie Kravitz, Johnny Lee, Joonhwan Lee, Ian Li, Sue O'Connor, Jerry Packard, Trevor Pering, Kai Richter, Fleming Seay, Irina Shklovski, Rachel Steigerwalt, Joe Tullio, and Roy Want.

I would like to thank my parents, Mick and Sally, and my sister, Amy, for all their support, love, and guidance. I have dedicated this thesis to my parents for kindling my interest in computers at a young age, first buying a Commodore 64 on which I learned BASIC, signing me up to take classes on programming, and later paying for Internet service in the early 90's when very few people knew what that was. Without their encouragement of my interests, and my education in general, I would not be where I am today.

Finally, I must thank my wonderful girlfriend Naomi. I am very happy that we have been able to remain close, even though our degree programs have forced us to live apart on opposite coasts for many years, and I am thankful that our time apart is almost over! There is no one who has supported me more and I could not have finished this without her.

CHAPTER 1

Introduction

Every day users interact with many computerized devices at both their home and office. On an average day I use my microwave oven, television, DVD player, alarm clock, and stereo at home, a computer for tracking speed and mileage on my bike, and a copier, fax machine, telephone/answering machine, vending machine, and CD player at school. That does not count the mobile phone and wristwatch that I usually carry around with me, nor does it count the three "normal" computers that I use daily or many of the computerized components that my car would have if it had been built in the last ten years. All of these devices have different interfaces, even for functions that are common across most of them such as setting the internal clock. Even devices that are functionally similar, like my car stereo, home stereo, and the media player on my computer at school, have vastly different interfaces. Problems like these are encountered by most users of today's computerized devices. Users must spend time learning how to use every device in their environment, even when they are similar to other devices that they already know how to use.

Part of the usability problem with today's computerized appliances is created by the many trade-offs that manufacturers must balance. They would like to produce highly usable appliances, but those appliances must also reach market in a timely fashion, be cheap to manufacture, and have the new features that users want. The declining cost of microproces-

sors allows manufacturers to cheaply and quickly add more computation and new features to their appliances, but unfortunately building high-quality user interfaces for these new features is still very expensive and time-consuming. 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].

One solution to these problems is to move the user interfaces from the appliances to some other intermediary "UI device" that is independent of the appliance manufacturer and focuses solely on the user interface. A UI device could be a handheld computer, like Microsoft's PocketPC, a mobile phone or even a speech system built into the walls of a building. One advantage of this approach is that people are increasingly carrying a mobile device that could be used as such a UI device. Many of these devices already have the ability to communicate with other devices in the environment using wireless networking protocols like 802.11 (Wi-Fi) or Bluetooth. Furthermore, these mobile devices are built with specialized interface hardware, like color and/or touch-sensitive LCD screens, which make the creation of high-quality user interfaces easier. A UI device could leverage its specialized hardware to provide a superior user interface as compared to what can be cost-effectively built into an appliance.

There are several potential approaches for moving the user interface onto the UI device:

- The controller device is *pre-programmed with interfaces* at the factory or by users in their homes. The advantage to this approach is that all of the interfaces are hand-designed specifically for the controller device and the appliances that it can control. Most of today's universal remote controls are pre-programmed with the codes for controlling a variety of home entertainment appliances, though some can be "taught" additional codes for other appliances and others have programming environments that allow users to create their own interfaces. If the manufacturer of the controller device does not provide an interface however, it is often tedious and time-consuming for users to program their own interfaces.
- The controller device *downloads complete user interfaces* from appliances or the internet. This approach, which is used by JINI [Sun 2003] and Speakeasy [Newman 2002], has the advantage that the controller device does not need to know in advance

about every appliance it might control. The appliance must be able to provide interfaces for each of the different controller devices it might encounter however, such as a standard graphical interface for use with touch screens and normal desktops, a list-based interface for use with phones, and a speech interface for devices that have a built-in speech recognizer. Controller devices with unique designs, such as a future watch with a circular screen and dials for interaction, would be very difficult to support with this approach as it would not be practical for appliance manufacturers to produce new interfaces for all of their appliances every time a new handheld device is released. Most of today's web technologies can be seen as using this approach, as many pages are carefully designed to support high-resolution screens and do not render appropriately on small devices such as mobile phones.

- The controller device *downloads abstract specifications* from appliances and uses those specifications to automatically generate an interface that is customized to the controller's particular design, properties of the user, and the user's environment. It is important that the specification be abstract enough to support the creation of interfaces on different platforms and in different modalities, but also contain enough information about the appliance in order to create a high-quality user interface. One advantage of this approach is that neither the controller devices nor the appliances need to know anything about the other in advance to ensure that the interfaces are usable. Another advantage is that external factors can be taken into account in the design of the interface, such as the user's previous experience or the functionality of other connected appliances. Although the automatic generation of user interfaces is difficult, I believe this approach is the most promising because it separates the creation of the user interface from the manufacturers of the appliances and the controller devices. This gives UI devices a greater opportunity to improve upon the usability of current appliance interfaces than the other approaches.

The technical focus of this dissertation is on the automatic generation of user interfaces for end users. Researchers have been exploring the automatic generation of user interfaces for nearly two decades, but most work relies on an interaction designer to guide the generation process and/or to edit the resulting user interfaces to fix any design flaws [Szekely 1996]. I do not believe that end-users of any UI device will be willing to spend the time and effort to modify their automatically generated interfaces in this way, and thus a UI device will need to generate high quality user interfaces on the first attempt.

Previous work in automatic interface generation has also focused in large part on building user interfaces for desktop applications, a task at which trained human designers can produce higher quality artifacts than any current automated system. While I would like interfaces generated by my system to approach the quality of human designers, my focus is instead on applying automatic generation techniques to create interfaces that would not be practical for a human designer to produce. For example, an automated system can produce a custom interface for every individual user whereas human designers are limited to designing interfaces for large user groups. Although previous work has examined how interfaces might be generated for multiple platforms from the same model [Szekely 1995, Eisenstein 2001], until now there had been no exploration of how automatic generation might be used to customize interfaces to each individual user and their particular environment and situation.

An important question to ask then is in what specific ways can UI devices with automatic interface generation be used to improve the usability of appliances? To answer this question, I will discuss several usability problems and how my approach can address them.

One of the biggest problems for appliance user interfaces is the large number of functions present in most appliances coupled with the limited number of interactive physical elements, such as buttons and screens, which can be built into a computerized appliance. To address this problem, most appliances overlap two or more functions on each button. For example, on the AT&T 1825 telephone/answering machine, pressing and releasing the play button will play all messages but pressing the play button and holding it down will play only the new messages. The limited size of the display screens also means that most appliances rely on cryptic messages or audio cues to give feedback to the user. For example, the same AT&T answering machine beeps once to notify the user that a speed-dial number has been successfully programmed and beeps twice if programming failed.

UI devices are able to address these problems for several reasons. UI devices can afford to have screens that are larger and higher resolution than the screens on most appliances. These screens can render at least a few lines of text at a time, which allows for better explanations of error conditions and feedback on the appliance's state. A UI device also has built-in infrastructure for supporting the development and rendering of complex user interfaces. This typically includes a toolkit that supports the creation of on-screen virtual controls, such as buttons and sliders, and a set of UI guidelines that govern how the interface should look and feel. Through use of the built-in infrastructure, a UI device can ensure that every appliance function is represented by a separate control and prevent the functional overlap that makes

so many appliances hard to use. An automatic interface generator can also make its interfaces easier to use by following the UI guidelines of the device and allowing users to leverage their knowledge of conventions used by the UI device's other applications.

Another usability problem for appliances arises when multiple appliances are used together in a system, such as in a home theater or presentation room. In order to accomplish their tasks, users must understand both how to use each appliance individually and how to manipulate the appliances so that they work properly with each other. Troubleshooting problems when they occur can be difficult because the user must determine which appliances are configured correctly and which are not. It is common for users to mistakenly believe that the problem lies in an appliance that is working correctly, which lengthens troubleshooting time and frustrates users. These interface problems occur for two reasons: 1) the user interfaces for each appliance are designed separately with few, if any, cues about how one appliance's interface will affect the operation of other appliances, and 2) for each task that might be performed with a system, each appliance has many functions that are not relevant for performing that task. These functions, though important for other tasks, may interfere with users' progress on their current task.

A UI device that automatically generates interfaces can address this problem by taking into account how appliances are connected. A UI device can also combine the user interfaces of multiple appliances into a single interface that is customized to the user's task, given that the UI device knows what task the user is trying to perform and how the appliances work together to perform that task. To obtain this information, I have a developed a novel solution based on the observation that most tasks in a system of appliances involve the flow of content from a "source" appliance to a "sink" appliance, possibly passing through other appliances along the way. For example, the task of playing a DVD movie in a home theater can be viewed as configuring the appliances to allow content from the DVD player to flow to the television and speakers. I have also developed a scaleable modeling technique to make content flow information available to the UI device, and the flow-based interface concept to allow users to directly specify the high-level task(s) they would like to perform. An AI planner is used to automatically configure the system to perform the tasks that users specify, and task-based interfaces are generated based on the currently-active content flows. This work also advances interface generation, as it is the first work to automatically generate interfaces that combine functionality from different sources.

Another problem for users is inconsistency between similar appliances from different manufacturers. For example, Figure 1.1 shows interfaces for three appliances that share similar photocopier capabilities. Notice that while the appliances share a few of the same physical elements, such as a number pad and screen, the user interfaces for these appliances are quite different. The methods of interaction are different, as the office copier (see Figure 1.1a) uses a large touch screen (one of the few appliances expensive enough to offer such a luxury) while the other two use physical buttons for navigating menus displayed on smaller screens. Labels also differ, such as for the brightness function which is labeled "darkness" on the HP all-in-one printer, "exposure" on the Canon all-in-one printer, and "light/dark" on the generic office copier. The organization of functions is also different through the appliances. As one example, the date and time are set on the HP printer through the fax configuration screen, whereas they are accessed through the general setup screen on both of the other appliances. The lack of consistency between interfaces prevents a user who is familiar with one of these appliances from leveraging that knowledge to use either of the other appliances.

a.

b.

c.

Figure 1.1. Physical interfaces for three different appliances with copier functionality: a) a Canon NP6035 office copy machine, b) a Canon PIXMA MP780 All-In-One Photo Printer, and c) a Hewlett-Packard Photosmart 2610 All-In-One printer. The latter two appliances also have fax and special photo printing capabilities.

UI devices are in unique position to ensure a consistent experience for their users, because the devices will be used for controlling most appliances and can track all of interfaces that users see. This knowledge, combined with information showing how the new interface is similar to previous interfaces, can be used to automatically generate new interfaces that use controls, labels, and organization that are already familiar to the user. These new interfaces are *personally consistent*, because they are based on their users' previous experiences. Different users may receive different generated interfaces for the same new appliance because their previous experiences are different. My work is the first work to demonstrate that an automatic generation system can ensure consistency for its generated interfaces and that this consistency is valuable in practice.

1.1 The Personal Universal Controller

I have explored all of these ideas in a system that I call the Personal Universal Controller (PUC), which enables UI devices to be constructed and supports the automatic generation of user interfaces.

The generation of user interfaces is enabled by the PUC's specification language, which allows each appliance to describe all of its functions in an abstract way. My goal is designing this language was to include enough information to generate a high-quality user interface, but not include any specific information about look or feel. Decisions about look and feel are left up to each interface generator. More than 30 specifications have been written for real appliances using the PUC language (see the full list in Table 5.1). These specifications describe the complete functionality of their appliances, unlike today's "universal" remote controls that typically support only the most common subset of an appliance's functionality. An authoring study of the specification language had subjects with no previous knowledge learn the language and then produce a specification of moderate size for a low-cost VCR. Subjects were able to learn the language in an average of 1.5 hours and produce the VCR specification in around 6 hours.

Four interface generators have been implemented that produce user interfaces from specifications written in the PUC's language, including graphical interface generators on PocketPC, Microsoft's Smartphone, and desktop computers, and a speech interface generator that uses the Universal Speech Interfaces (USI) framework [Rosenfeld 2001]. The graphical interface generators for the PocketPC and Smartphone cover two very different interface styles. The

a. b. c. d.

Figure 1.2. Interfaces generated by the PUC on the PocketPC for several different appliances: a) Windows Media Player 9, b) the navigation system from a 2003 GMC Yukon Denali, c) the Canon PIXMA MP780 all-in-one printer, and d) the HP 2610 all-in-one printer.

PocketPC has a medium-sized touchscreen that allows interactions similar to a desktop computer, though with a smaller screen area (see Figure 1.2). The Smartphone has a small screen with no touch sensitivity and a four-way joystick for navigating around the interface. As a result, Smartphone interfaces are list-based and typically have a much deeper hierarchy than PocketPC interfaces (see Figure 1.3).

The interface generators have been tested with a variety of complex appliance specifications, including those for several VCRs, the HP and Canon all-in-one printers mentioned above, and even the navigation system for a GMC Yukon Denali SUV. This latter specification for the navigation system is especially complex, but is easily handled by the PocketPC interface generator (see Figure 1.4). I have also evaluated the usability of the generated interfaces by comparing the PUC-produced interfaces for the HP and Canon all-in-one printers (see Figure 1.2c-d) with the manufacturers' interfaces on the actual appliances (see Figure 1.1b-c). The comparison showed that subjects on average were *twice as fast* and *four times as successful* when using the PUC interfaces to complete a set of representative tasks.

The interface generators can also produce interfaces that are *personally consistent* based on a user's previous experiences. When a new interface is generated for an appliance with functionality that is similar to a previous appliance that the user has seen, the interface generator uses special algorithms to ensure that the similar functions are represented the same way and placed in the same location. The difficulty with generating consistent interfaces is that while

a. b. c. d.

Figure 1.3. Interfaces generated by the PUC on the Smartphone for several different appliances: a) Windows Media Player, b) a simulated elevator, c-d) a shelf stereo with CD, radio, and tape functionality.

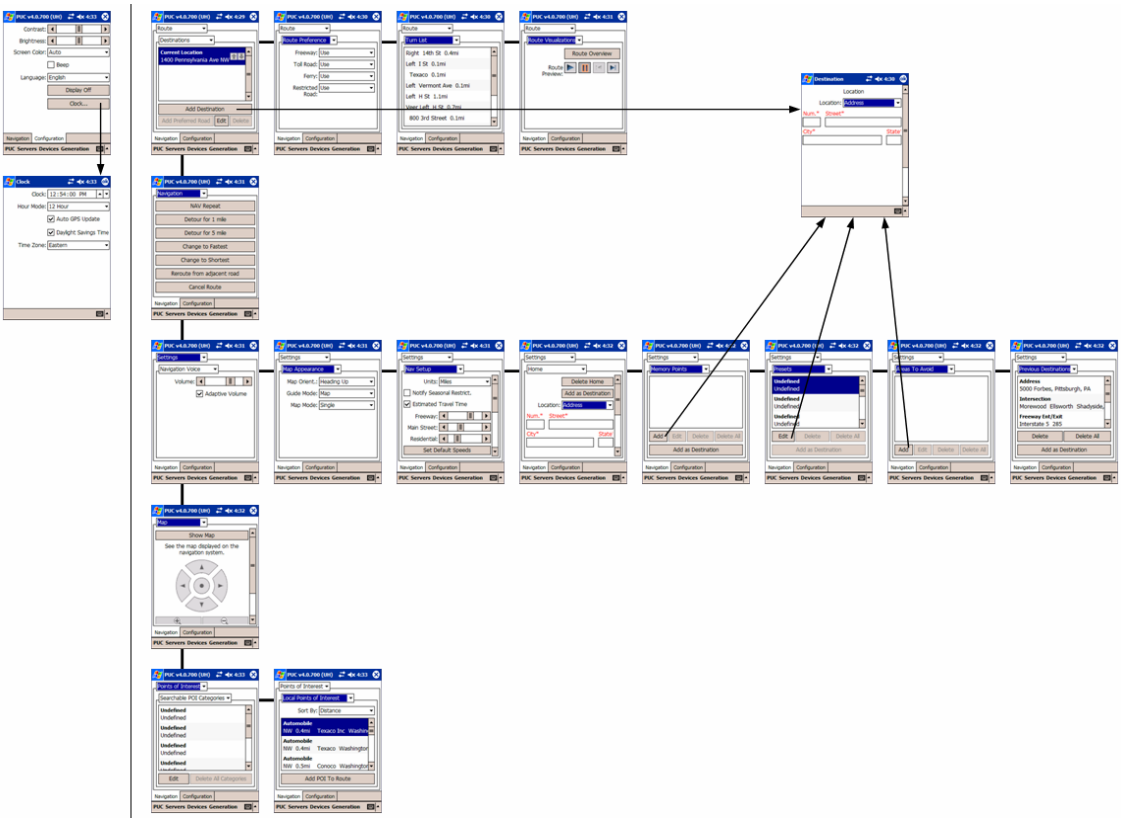


Figure 1.4. The full PocketPC interface generated for the navigation system from a 2003 GMC Yukon Denali. The arrows show buttons that cause a dialog box to open.

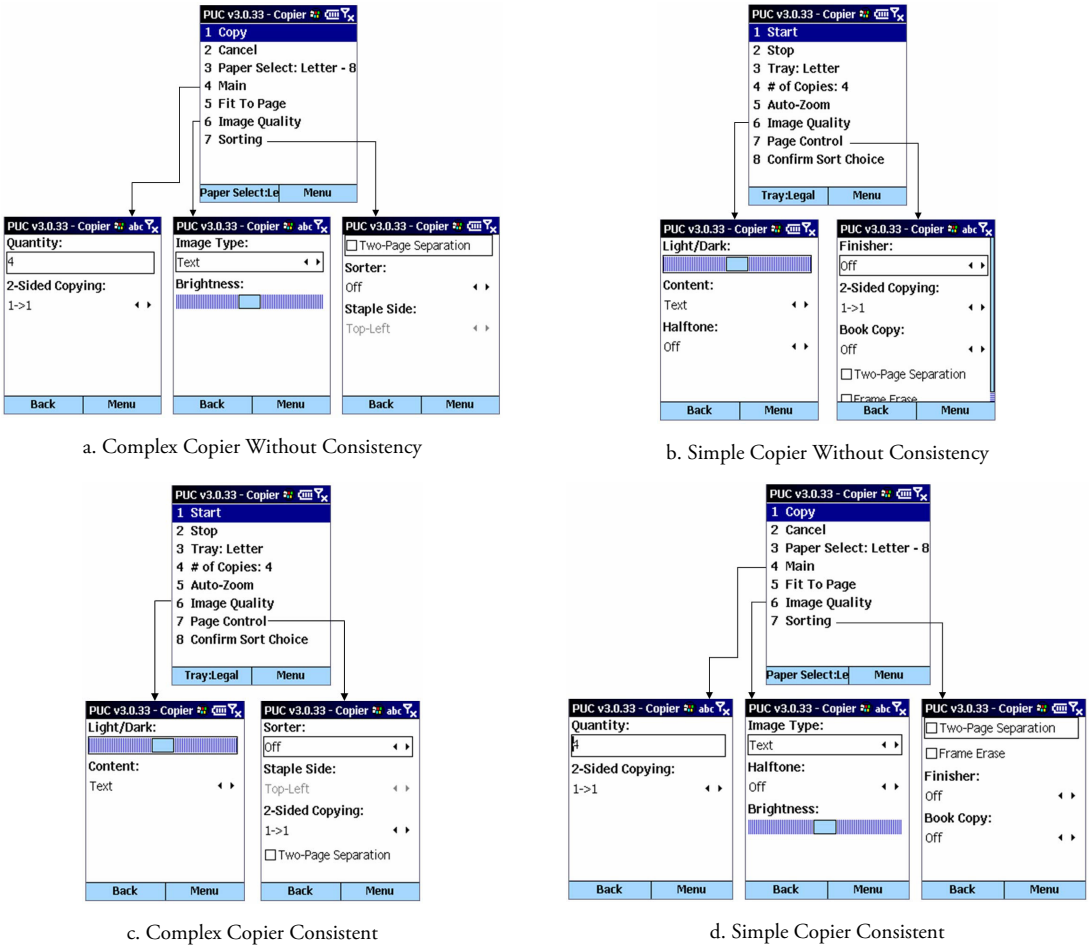
a. Complex Copier Without Consistency

b. Simple Copier Without Consistency

c. Complex Copier Consistent with Simple Copier

d. Simple Copier Consistent with Complex Copier

Figure 1.5. Copier interfaces generated with and without consistency on the PocketPC platform.



a. Complex Copier Without Consistency

b. Simple Copier Without Consistency

c. Complex Copier Consistent with Simple Copier

d. Simple Copier Consistent with Complex Copier

Figure 1.6. Copier interfaces generated with and without consistency on the Smartphone platform.

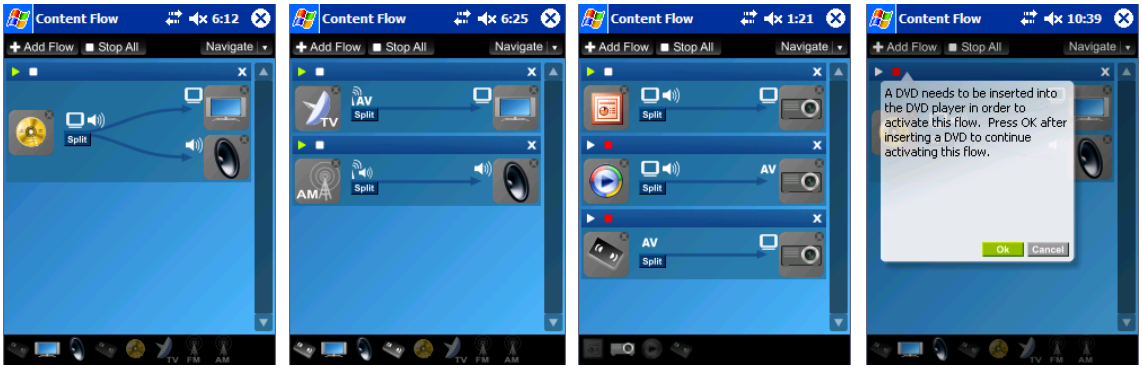
appliances may share some similar functions, there are also many unique functions which must be included in the user interfaces. I have developed techniques for preserving the usability of unique functions as changes are made for consistency.

Examples of interfaces generated with consistency are shown in Figure 1.5 for the PocketPC and in Figure 1.6 for the Smartphone. Notice that there are several differences between the original interfaces for the complex and simple copiers. The visual organization of the two interfaces is different on the PocketPC and the structural organization differs on the Smartphone. Labels are quite different, such as "Start" and "Stop" being used on the complex copier and "Copy" and "Cancel" used on simple copier. Some functions are located differently between the interfaces, such as the two-sided copying function which is located with the quantity function on the complex copier but with the sorting functions on the simple copier. There are also unique functions, such as the "Halftone" function on the complex copier and the "Book Copy" function on the simple copier. Note that the consistent interfaces for these copiers address the differences without disrupting the usability of the unique functions.

I have also conducted an evaluation of the rules for generating consistent interfaces, again using the HP and Canon all-in-one printer appliances. In this study, users with no knowledge of either appliance were asked to perform a set of tasks on a PUC interface for one of the appliances. Following the tasks, users were instructed on the correct method for performing each task on the first interface and then asked to complete the same tasks on a PUC interface for the other printer. This second interface was either generated normally (without consistency) or generated to be consistent with the first interface. I found that users were *twice as fast* on average at performing tasks on the second interface when that interface was generated to be consistent as compared to the users of the normally generated interface.

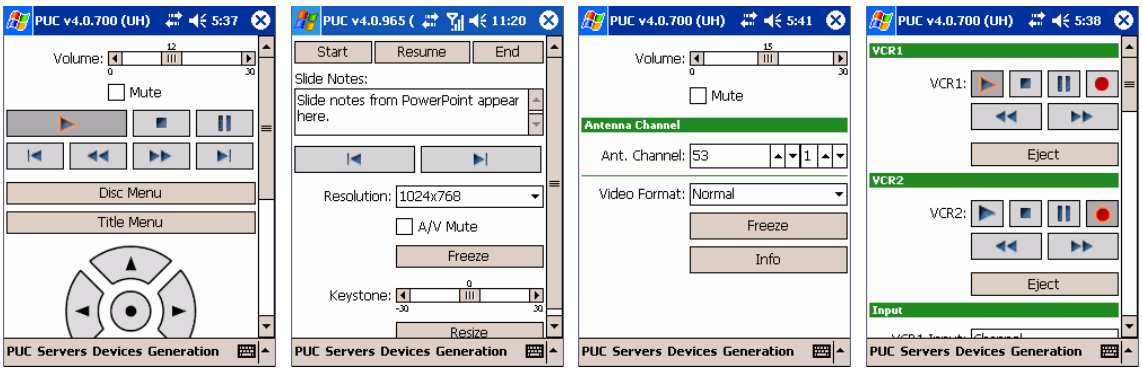
The interface generator can also produce user interfaces for systems of multiple connected appliances, such as home theaters or presentation rooms. These interfaces are generated based on a full content flow model of the appliance system that is assembled from each appliance's specification and a wiring diagram for the system supplied by a third party, such as a future wiring technology or the user. The PUC provides two types of interfaces for interacting with a system of appliances: the flow-based interface that allows users to specify their high-level tasks (see Figure 1.7) and aggregate interfaces that combine functions from multiple appliances to allow users to perform low-level actions during a task (see Figure 1.8).

The PUC system produces four different kinds of aggregate user interfaces. Active Flow Control interfaces provide access to the most common control functions of the currently active flows, such as volume and playbacks controls. Figure 1.8 shows four example of this type of aggregate user interface. Active Flow Setup interface provide access to less common configuration features for the currently active flows, such as brightness and contrast for a television or speaker balance for a receiver. The General Setup aggregate interface contains setup and configuration options that do not relate to any flow, such as the parental control settings present on some appliances. Finally, similar functionality from multiple appliances



a. b. c. d.

Figure 1.7. Several examples of the Flow-Based Interface being used for various tasks: a) playing a DVD movie with the video shown on the television and audio coming through the stereo's speakers, b) watching a sporting event on television but listening to the play-by-play over the radio, c) selecting different sources of content during a presentation, with a PowerPoint slideshow as the current source, and d) resolving a problem with the DVD player through the question/answer interface.



a. b. c. d.

Figure 1.8. Several examples of aggregate user interfaces generated based on the user's current task: a) playing a DVD movie with the video shown on the television and audio coming through the stereo's speakers, b) presenting Powerpoint slides through a projector, c) watching broadcast television with audio playing through the television's speakers, and d) recording a tape in one VCR from a tape playing in another VCR.

are merged into a single interface control for certain functions, such as the clock or language settings. The merged aggregate interface allows the user to set the current time, for example, and have this value migrated automatically to every appliance in the system.

In order to demonstrate the feasibility of the PUC system and to ensure that the PUC supports all of the complexities of today's computerized appliances, it is important for the PUC to control real appliances. To accomplish this, the PUC system has a communication protocol to allow handheld devices to interact with appliances and a set of appliances adaptors that translate between the PUC protocol and the proprietary protocols of existing appliances. The PUC is currently able to control nine existing appliances (see Table 4.1 for a complete list) and it would be easy to increase this number as more appliances are built with support for external control.

1.2 Topics Outside the Scope of this Work

My work on the PUC system could have been taken in many directions. This section describes issues related to the PUC system that I do not explore in this dissertation:

Help systems: When users encounter problems with an automatically generated interface, they should be able to access help information that is generated based upon the properties of the interface. Such automated help systems have been created in the past, such as the Cartoonist system for the UIDE environment [Sukaviriya 1990].

Automated trouble-shooting for complex systems: The functionality of a complex system, like the home theater system described in the previous section, often depends on how its component pieces are connected together. For example, video performance will be bad if a DVD player is connected through a VCR, or it will not be possible to record video from one VCR onto another if the two are not connected properly. The PUC system has sufficient information to reason about such problems and conceivably could help users find solutions for their particular systems, but this is not something that I have looked at as part of my thesis work.

Service Discovery: A PUC controller device must be able to "discover" appliances in the environment that the user may wish to control. Efficiently performing this task without centralized servers has been a focus of several research projects, and those techniques have become common enough to be included in commercial systems such as UPnP [UPnP

2005] and JINI [Sun 2003]. The PUC system relies on existing techniques and does not further this research.

End-User Programming and Macros: Facilitating end-user programming tasks, such as the creation of user-specified macros, would be an interesting direction for the PUC research. This area is not unique to the PUC system however, and many other researchers are exploring end-user programming in other contexts. I am confident that their advances could be applicable to the PUC system in the future.

Security: Security is an important issue for systems like the PUC. How do users keep people who are driving by on the street from maliciously controlling their stereos or kitchen appliances? Again, a lot of interesting work could be done in this area, but I have chosen not to address this in my thesis.

1.3 Contributions

My thesis is:

> **A system can automatically generate user interfaces on multiple platforms for remotely controlling appliances where the user's performance is better than with the manufacturer's interfaces for the appliances.**

I have conducted evaluations of my interface generators that demonstrate that this thesis holds, as discussed in detail in Chapter 10. This dissertation also makes a number of other contributions:

- An abstract appliance specification language for describing the complete functionality of a wide-range of appliances,
- The use of dependency formulas in appliance specifications to help determine the structure of generated user interfaces,
- The general Smart Templates technique for incorporating domain-specific design conventions into an appliance specification and rendering the conventions appropriately on different platforms and in different interface modalities,
- A language for describing semantic similarities between appliance specifications,
- Algorithms that apply knowledge of appliance similarities and user history to generate new interfaces that are consistent with previous interfaces the user has seen,

- The Flow-Based Interface concept, which allows users to quickly and easily specify high-level goals for a multi-appliance system,
- Algorithms use a model of content flow in a multi-appliance system to generate task-based interfaces that combine functionality from multiple appliances,
- Interface generation software on multiple platforms: PocketPC, Microsoft's Smartphone, and desktop computers, which make use of the above contributions to produce appliance interfaces that have been demonstrated via user testing to be more usable than manufacturers' interfaces for the same appliances.

1.4 Dissertation Overview

The next chapter, Chapter 2, surveys related work. The remainder of the dissertation describes the different components of the PUC system building towards the description of the interface generation process in Chapters 8 and 9 and the usability evaluation of the generated interfaces in Chapter 10.

Chapter 3 describes preliminary studies of appliance user interfaces that set the groundwork for the design of the specification language and the interface generator. Of particular importance in this chapter is the list of requirements for any system that intends to control appliances.

Chapter 4 discusses the general architecture of the PUC system and the infrastructure needed for communicating with and controlling real appliances.

Chapter 5 describes the specification language and discusses its completeness and usability.

Chapter 6 discusses consistency, including previous work in the area and what consistency means for the PUC system. This chapter also covers the infrastructure needed for generating consistent interfaces, including the knowledge base that stores similarity mappings between specifications and maintains a history of interfaces that the user has seen.

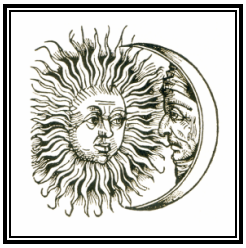
Chapter 7 describes the Smart Templates technique that allows domain-specific design conventions to be included in automatically generated interfaces.

The PUC interface generation process is split into two chapters. Chapter 8 describes the process for single appliance user interfaces on the PocketPC, desktop, and Smartphone plat-

forms. Chapter 9 covers the generation of aggregate interfaces for multi-appliance systems and the PUC's novel use of content flow information.

Chapter 10 describes a usability evaluation of the generated interfaces compared to the manufacturers' interfaces for two all-in-one printers and an evaluation of the consistency algorithms compared to generation without consistency.

Finally, Chapter 11 discusses the overall PUC system, reviews the contributions of the dissertation, and outlines directions for future work.



CHAPTER 2

Related Work

This chapter surveys previous work in three areas of research: control of appliances, user interface generation, and aggregate user interfaces.

Work in the control of appliances includes both commercial products, such as universal remote controls, and appliance communication infrastructures being developed in both industry and academic research. A few projects have examined how the user interfaces for consumer electronics might be improved.

User interface generation has been the subject of research for many years, sometimes under the name of model-based user interfaces because the interfaces are generated from models of the application domain, tasks the user might perform, and the target platform. The original goal of this work was to allow programmers, who were not typically trained to design interfaces, to produce good user interfaces for their applications. More recent work falls into two categories: fully automatic generation for producing interfaces customized for each user and designer-guided generation for producing and maintaining very large scale user interfaces.

Aggregate user interfaces are those that combine functionality from multiple sources to produce one user interface. Also known as "mash-ups," this idea has recently become a popular piece of the new "Web 2.0" technologies, where developers are combining data from multi-

ple web sites on a single site to produce compelling visualizations and applications. Web mash-ups are one type of aggregate interface, though most examples to date were built by hand. Some work has been done in the human-computer interaction and ubiquitous computing communities on automatically generating aggregate interfaces. The Web Services and Semantic Web communities have also been investigating web service composition, though most of this work focuses on infrastructure issues and does not address the user interface.

2.1 Control of Appliances

A number of systems have been created for controlling appliances. Commercial products have been available for years that allow limited control for certain electronic appliances, and recently companies have begun forming standards groups to agree on new solutions for controlling appliances, such as HAVi [HAVi 2003], JINI [Sun 2003], and UPnP [UPnP 2005]. Another standards group, INCITS/V2 [INCITS/V2 2003], was formed to examine standardizing appliance control in order to benefit people with handicaps. There have also been several research projects that have explored this area such as Xweb [Olsen Jr. 2000], ICrafter [Ponnekanti 2001], and Roadie [Lieberman 2006].

2.1.1 Commercial Products

For years many companies have been selling so-called "universal remote controls," which replace the standard remote controls for televisions, VCRs, and stereos with a single remote control unit. A one-way infrared protocol is used to issue commands to the appliances. Typical universal remote control products have physical buttons that correspond to the most common subset of features found on the appliances that the universal remote can control. For televisions this is usually limited to channel up and down, volume up and down, a number pad for manually entering channels, and a power button. For example, my mother has a universal remote for her television and VCR, but must use the original remote for the TV in order to access the television's configuration menus. Some universal remotes avoid this problem with a teaching feature, which allows the user to assign buttons on the universal remote to a particular code that is recorded from the original remote control.

The Philips Pronto (see Figure 2.1) was one of the first LCD-based universal remote control products. In addition to being able to program new infrared codes for new appliances, users can also design the panels of the controls that they use. Using the Pronto, it is easy, for example, to create a specialized screen for watching movies that has the DVD player and stereo

volume controls, and another for watching television that only has controls for the cable box channels. Users can even associate multiple codes with a single button, allowing them, for example, to create a macro for playing a DVD that turns on the DVD player and television, switches to the appropriate channel, and plays the DVD. The problem with the Pronto, as with the other universal remotes, is that all of the programming must be done manually, which can be a tedious and time-consuming task, especially for a large number of appliances.

The Logitech Harmony remote (see Figure 2.2) is unique among universal remotes because it internally tries to maintain a record of the current state for all of the appliances that it can control. This has the limitation that the remote must know the state of the system when it is first used and that *all* control must be done via the Harmony remote afterwards, but it has the advantage that the remote can hide functionality that is not available in the current state. The user interface is further simplified using a task-based interface shown on the small LCD screen which displays a list of tasks, such as "play movie in VCR" or "play DVD." The list is based upon the appliances the user has and the current state of the system. When one of these options is selected, the remote sends the appropriate codes to all appliances and may also instruct the user to do certain tasks, such as insert a DVD into the player.

Both of these remote control devices also synchronize with a desktop computer to make the task of programming easier. This also allows users to download their remote control layouts from the device and share them with other users on the Internet. Several communities have been created to share panels for the Pronto, such as `http://www.remotecentral.com` and `http://www.prontoedit.com`. Synchronization is also the basis for programming the Harmony remote, which is done via a web site that gives the user access to Harmony's extensive proprietary database of appliance state information. Synchronization helps decrease the

Figure 2.1. The Philips Pronto TSU-9600 remote control device.

Figure 2.2. Two Harmony remote control devices (the 890 and 1000 models).

tediousness and time-consuming nature of programming the remote controls, but only for appliances where some user has already uploaded the codes. For other appliances, the programming process is just as time-consuming when using these advanced universal remotes.

2.1.2 Commercial Standards

A number of industry groups have formed to create standards for remotely controlling devices. Four of the most prominent are the Microsoft-led Universal Plug and Play (UPnP) [UPnP 2005] initiative, the UPnP-affiliated Digital Living Network Alliance (DLNA) [DLNA 2006], the Home Audio Video Interoperability (HAVi) initiative which is led by "eight of the world's leading manufacturers of audio-visual electronics" [HAVi 2003] and the INCITS/V2 effort [INCITS/V2 2003] which is a collaboration between the National Institute for Standards and Technology (NIST) and a consortium of researchers from industry and academia. The goal of all of these standards initiatives is to create a flexible communication infrastructure that makes it easier for people to control the appliances in their environment and for appliances to interoperate with each other.

Industry standards are also being developed to enable the construction of distributed network services. Unlike the above standards, this work is focused on generic "services" rather than specifically on electronic appliances. The proposed technologies in these standards, including service discovery and description, could be used to find and control appliances. These standards include Sun Microsystem's JINI system [Sun 2003] and the OSGi Alliance [OSGi 2006].

2.1.2.1 INCITS/V2 Standard

Recent government legislation requires that appliances purchased by the government or government entities be usable by people with a wide variety of disabilities. Unfortunately, most appliances built today have no accessibility features. The InterNational Committee for Information Technology Standards (INCITS) has begun the V2 standardization effort [INCITS/V2 2003], which is currently developing standards for

Figure 2.3. An interface generated for WinAmp on a PocketPC using the INCITS/V2 framework [IMTC 2006]. Reproduced with permission.

a Universal Remote Console (URC) that would enable many appliances to be accessible through the Alternative Interface Access Protocol (AIAP). A URC controls an appliance by using AIAP to download a specification written in three parts: a user interface "socket" that describes only the primitive elements of the appliance, a "presentation template" that describes either an abstract or concrete user interface, and a set of resource descriptions that give human-readable labels and help information for the user interface. The URC will either then automatically generate an interface from an abstract presentation template, or display one of the interfaces specified in a concrete presentation template. An example WinAmp interface generated from by a URC using the V2 framework is shown in Figure 2.3. I have provided feedback to the V2 group in the past that led to the current design of their specification. A detailed report is available analyzing the similarities and differences between the V2 and PUC systems [Nichols 2004a].

2.1.2.2 Universal Plug and Play

Universal Plug and Play (UPnP) is designed both to allow user control and appliance interoperation. The two important units within UPnP are the "service" and the "control point," which are similar to the appliance and controller respectively in the PUC system. Each UPnP service has a downloadable description formatted in XML that lists the functions and state variables of that service. Control points connect to services, download their descriptions, and then call functions and receive event notifications from the services. One important difference between the PUC and UPnP is the automatic generation of user interfaces. UPnP chose to avoid automatic generation, and instead relies on standardized appliance descriptions. A standardized description allows a control point to know in advance what functions and state variables a service will have, which allows a hand-designed user interface to be created in advance on a control point. Similar to HAVi, UPnP does allow services to specify additional functions and state variables beyond those in the standardized set, but it is not clear how a control point would accommodate these additional functions or variables in its user interface. UPnP provides a way around this, by allowing a control point to also download a web page and control the specialized functions of the service using standard web protocols, but the solution results in two different user interfaces being displayed on the same controller device.

Several UPnP products are available today, including gateway/router products, streaming AV products, and a Pan/Tilt video camera from Axis Communications. UPnP currently has

standardized twelve different service descriptions, and more devices are likely to appear on the market as the number of standardized service specifications grows.

2.1.2.3 Digital Living Network Alliance

The Digital Living Network Alliance (DLNA) is developing a series of standards, based on the UPnP standards, to improve the usability of network-connected appliances throughout the home. A typical DLNA usage scenario would be viewing some content from a networked PC, such as music, video, or pictures, on a stereo or television located elsewhere in the house. Unlike for UPnP, where the focus is on technology, DLNA's focus is strictly on improving the user experience. DLNA guidelines cover not only the technical layer, to allow easy integration between appliances, but also the design of the user interfaces. Products can only receive DLNA certification if they comply with the technical and UI guidelines. The PUC philosophy differs from DLNA in that the PUC allows users to decide how to integrate functionality between their appliances and supports the inclusion of new appliances that cannot be envisioned today. DLNA is based around a set of general use cases for appliances that can be envisioned today, which may require revision in the future if new classes of appliances become commonplace.

2.1.2.4 Home Audio-Video Interoperability

Home Audio-Video Interoperability (HAVi) is the only platform designed specifically for consumer electronics devices like televisions and VCRs, and only works over an IEEE 1394 (Firewire) network. Televisions that feature HAVi are available today from RCA and Mitsubishi Electric, and Mitsubishi at one time produced a VCR that supported HAVi. HAVi's user interface concept is that the television, because it is only appliance with a large screen, should control all the other appliances in a home network. There are three ways that a HAVi controller might control another appliance: (1) every type of appliance that might be controlled has a standardized interface specified by the HAVi working committee and a HAVi controller could have a hand-designed interface built-in for each standardized type, (2) every appliance can export a "level 1" or data-driven interface, which is basically a description of a hand-designed interface that includes buttons, labels, and even multiple panels, and (3) every appliance can export a "level 2" user interface, which is a piece of mobile code written in the Java language which displays a remote control user interface when executed on a HAVi controller. None of the interface descriptions are abstract, as the PUC appliance specification language is, and only the second and third interface description options may allow the HAVi

controller to access special features of the appliance. The main advantage of HAVi over other proposed industry standards is its ability to control older "legacy" appliances using older protocols such as AV/C [Association 1996]. The main disadvantage of HAVi is the size of its API, which includes three levels of interface specification, standardized templates for many types of appliances which must be built into any controller implementation, a Java virtual machine, and support for a number of legacy protocols.

2.1.2.5 JINI

Sun's JINI system was designed as a network infrastructure to make it easier for programmers to create distributed systems. Like the PUC, INCITS/V2, HAVi, and UPnP, it could allow a controller device to manipulate an appliance, but the infrastructure is much more general. The system is a set of APIs for discovering services, downloading an object that represents the service, making calls on the object using a remote procedure call protocol, and finally releasing the object when it is no longer needed. Like HAVi, JINI also relies on the Java platform to send mobile code from the service to the computer that wishes to use the service. This mechanism could be used, for example, to display a user interface that allows a human to control a service. It would be possible to implement a system like the PUC on top of the JINI protocol, but JINI by itself does not provide the user interface description features that the PUC does.

2.1.2.6 OSGi

OSGi [OSGi 2006] is a dynamic module system for the Java programming language, which provides several standard primitives that allow developers to produce applications from a set of small, reusable components. The OSGi Service Platform allows modules to be composed across devices and networks and for the composition to be changed dynamically to support new tasks. This platform also allows for modules to be automatically discovered and integrated into a currently running system. These features allow OSGi to be used as the infrastructure in a variety of smart home projects, as listed on the web:

`http://www.osgi.org/markets/smarthome.asp`

2.1.3 Research Systems

2.1.3.1 Universal Interactor

Hodes, et al. [Hodes 1997] propose a similar idea to our PUC, which they call a "universal interactor" that can adapt itself to control many devices. Their approach uses two user inter-

face solutions: hand-designed interfaces implemented in Tcl/Tk, and interfaces generated from a language they developed called the "Interface Definition Language" (IDL). IDL features a hierarchy of interface elements, each with basic data types, and supports a layer of indirection that might allow, for example, a light control panel to remap its switch to different physical lights as the user moves between different rooms. Unlike the PUC work, this work seems to focus more on the system and infrastructure issues than the user interface. It is not clear whether IDL could be used to describe a complex appliance, and it seems that manually designed interfaces were typically used rather than those generated from an IDL description.

2.1.3.2 IBM Universal Information Appliance

An IBM project [Eustice 1999] describes a "Universal Information Appliance" (UIA) that might be implemented on a PDA. The UIA uses an XML-based Mobile Document Appliance Langauge (MoDAL) from which it creates a user interface panel for accessing information. A MoDAL description is not abstract however, as it specifies the type of widget, the location, and the size for each user interface element.

2.1.3.3 ICrafter

The Stanford ICrafter [Ponnekanti 2001] is a framework for distributing and composing appliance interfaces for many different controlling devices. It relies upon a centralized interface manager to distribute interfaces to handheld devices, sometimes automatically generating the interface and other times distributing a hand-designed interface that is already available. ICrafter can even distribute speech interfaces described by the VoiceXML language to those controllers that support speech. Support for the automatic generation of user interfaces is limited however, and they also mention the difficulty of generating speech interfaces.

Perhaps the most interesting feature of ICrafter is its ability to aggregate appliances together and provide a user interface. To support composition, ICrafter relies on a set of "service interfaces" (that abstract the functionality of services) and a set of interface aggregators that are each hand-coded to build an interface for a particular pattern of service interfaces. When a user requests an interface for multiple services, ICrafter looks for an aggregator that matches the pattern and, if an aggregator is found, returns a single interface generated by that aggregator. For example, a camera might implement the DataProducer interface and a printer might implement the DataConsumer interface. The generic aggregator for the DataPro-

ducer/DataConsumer combination could then generate a combined interface for the camera and printer.

ICrafter's approach has several limitations however, which the PUC overcomes. First, a generic aggregator in ICrafter is only able to generate an interface for the common properties and functions shared by its service interfaces and none of the unique functions that may be implemented by a specific service. Second, ICrafter's generic aggregators are not able to include any design conventions that might be specific to the services. For example, a play button would be appropriate if a DataProducer was a DVD player but not if the producer was a camera. For ICrafter to produce interfaces with unique functions and appropriate design conventions, a special purpose interface aggregator would need to be built for the specific appliances involved. In contrast, The PUC's interface aggregation is faithful to the specific appliance interfaces that are being aggregated and includes all functionality of the connected appliances.

2.1.3.4 Xweb

The Xweb [Olsen Jr. 2000] 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 the Universal Speech Interface approach, as it is derived from a joint philosophy [Rosenfeld 2001]. 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 [Olsen Jr. 2000]. The interfaces designed for the PUC using the Universal Speech Interface design differ by trying to stay closer to the way people might talk about the task itself, and is somewhat closer to naturally generated speech.

2.1.3.5 Ubiquitous Interactor

The Ubiquitous Interactor (UBI) system [Nylander 2004] is also working to separate presentation from functionality for services. Services in UBI are described using interaction acts, somewhat like the abstract interaction objects used by the PUC interface generator, which

describe the interaction the user should have without providing any information about how the interaction should be presented. The general description of the service can be augmented with service- and device-specific hints that are provided in a customization form. The UBI interface generator combines the information from the interaction act and the customization forms to produce a final user interface.

The unique feature of UBI comes from its customization forms that allow the service provider to supply hints about how the generated interface should appear. This gives the service providers control over the generated interfaces and allows them to include brand marks and interactions.

2.1.3.6 Analyses of Actual Remote Control Usage

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. My approach differs because the PUC interfaces include the full functionality for each appliance rather than a subset containing the most commonly used functions. In the future, I am interested in applying Omojokun's work to optimize the organization of the PUC's user interfaces to favor commonly used functions while still including the remaining functions.

2.1.3.7 DiamondHelp

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.

2.1.3.8 Roadie

The Roadie system [Lieberman 2006] provides a goal-oriented user interface for consumer electronics that may combine features of multiple appliances. Like the PUC's flow-based interface (see Chapter 9), Roadie uses a planning algorithm to automatically configure appliances to match user goals. Unlike the PUC, Roadie uses a database of commonsense knowledge to find and understand possible user goals within the system. The user can specify the action they wish to perform using natural language, and then Roadie will attempt to interpret this action using its database and create a plan. Because the possible actions are restricted to the contents of the commonsense database, Roadie may not be able to support uncommon actions, such as those related to an uncommon configuration of appliances or to a new class of appliance that has just been added to the system. The PUC, in contrast, is able to acquire a model of the system from the appliances themselves, and thus is not subject to these limitations.

2.2 Automatic & Guided User Interface Design

Research in interface generation has a long history dating back to some of the earliest User Interface Management Systems (UIMSs) developed in the mid-80's, such as COUSIN [Hayes 1985]. The original goal of these systems was to automate the design of the user interface so that programmers, who were typically not trained in interface design, could produce applications with high quality user interfaces. This work led to creation of systems in the late 80's and early 90's, such as UIDE [Sukaviriya 1993], ITS [Wiecha 1990], Jade [Vander Zanden 1990], and Humanoid [Szekely 1992], which required designers to specify models of their applications that could then be used to automatically generate a user interface. The generated interfaces could generally be modified by a trained interface designer to produce a final user interface. These interfaces were sometimes called *model-based user interfaces* because of the models underlying their creation.

These early model-based systems had several drawbacks. Most notably, creating the models needed for generating an interface was a very abstract and time-consuming process. The modeling languages had a steep learning curve and often the time needed to create the models exceeded the time needed to manually program a user interface by hand. Finally, automatic generation of the user interface was a very difficult task and often resulted in low quality interfaces [Myers 2000]. Most systems moved to designer-guided processes rather than use a fully automatic approach.

Two motivations suggested that continued research into model-based approaches might be beneficial:

- *Very large scale user interfaces* assembled with existing techniques are difficult to implement and later modify, and detailed models of the user interface can help organize and partially automate the implementation process. The models can then be used to help designers re-visit the interface and make modifications for future versions.
- A recent need for *device-independent interfaces* has also motivated new research in model-based user interfaces and specifically on fully automated generation. Work in this area has also begun to explore applications of automatic generation to create interfaces that would not be practical through other approaches. For example, the PUC's consistency feature (see Chapters 6 & 9) generates interfaces that are personally consistent with each user's previous experience.

While all of these systems discussed in this section generate interfaces, to our knowledge no user studies have been conducted to evaluate the resulting interfaces. The closest reported study is of SUPPLE [Gajos 2004], discussed below, which asked subjects without any interface design training to produce interfaces for a presentation room control panel. The developers then showed that SUPPLE could generate similar versions of each of these interfaces by varying the task information provided to the interface generator. The interface used in this study had only a few simple functions however, and users' performance on the SUPPLE interfaces was not measured or compared with any other interfaces.

The following sections highlight many of the model-based systems that have been produced over the years. The discussion is broken down into a discussion of the early systems (ending in the early-to-mid 90's) and more recent systems.

2.2.1 Early Model-Based Systems

The initial research in model-based systems was conducted from approximately the mid-80's to the early 90's. An excellent review of early model-based user interface research can be found in [Szekely 1996].

2.2.1.1 Mickey

One of these early systems was Mickey [Olsen Jr. 1989], which automatically generated menus and dialog boxes from function signatures and strategically placed comments in the

code implementing the application logic. This simplified the construction of user interfaces for programmers, who could now implement the logic, add a few special comments, and immediately have a limited user interface for their application. While the generated user interface was rarely sufficient for the entire application, the techniques demonstrated by Mickey showed promise for simplifying the user interface implementation process.

2.2.1.2 Jade

Jade [Vander Zanden 1990] is another example of an early model-based system for automatically generating dialog box layouts based on a textual specification of the content. Like the PUC's specification language, Jade's textual specification contains no graphical information, which keeps each specification small and allows the look-and-feel of the generated interfaces to be independent of their content. Unlike the PUC system, Jade allows interface designers to manually edit its results to fix any problems in the automatically generated interfaces. Most of the model-based systems discussed in this section have similar features for allowing the interface designer to guide the generation process and/or edit the final user interfaces. While the PUC system could allow manually editing, it is important to remember that users of the PUC system are not trained designers and will rarely have the time or desire to modify a generated interface.

2.2.1.3 UIDE

Systems of the late 80's and early 90's, such as UIDE [Sukaviriya 1993], HUMANOID [Szekely 1992] and ITS [Wiecha 1990] expanded on these ideas with more complicated models that could generate more sophisticated user interfaces. UIDE, which stands for User Interface Design Environment, is the earliest of these systems. The knowledge base contains information about objects, the actions that users can use to manipulate those objects, and pre-conditions and post-conditions for each action that describe what must be true for the action to be executed and conditions that are true once the action has been executed. Pre-conditions and post-conditions are similar to the dependency information used in the PUC specification language. The development of UIDE led to several advances in the automatic design and layout of dialog boxes. It was shown that a decision tree could be constructed that performed well for choosing the interface element to use for a particular variable or action [de Baar 1992], and the DON system [Kim 1993] used metrics and heuristics to create pleasing layouts of interface elements. The PUC interface generators use and extend these techniques. Another interesting tool developed as a part of UIDE is Cartoonist [Sukaviriya

1990], a system for automatically generating animated help from pre- and post-condition information. It may be possible to create a similar system using the PUC specification's dependency information, but that is a subject for future work.

2.2.1.4 Humanoid

HUMANOID [Szekely 1992] is a tool for supporting the creation of the entire application, going beyond the creation of menus and dialog boxes and focusing on the construction of interfaces with visualizations for complex data. An important feature of HUMANOID is the designer's interface, which integrates all design aspects of the system into a single environment and focuses the designer on a tight design/evaluate/redesign cycle. To support this cycle, the system is explicitly designed such that the application can be run even if it is not fully specified. The benefit of this is that designers can get immediate feedback and explore many alternatives in a short amount of time.

2.2.1.5 Mastermind

The MASTERMIND project [Szekely 1995] started as collaboration to combine the best features of UIDE and HUMANOID. In addition to modeling capabilities of those systems, MASTERMIND also uses task models to inform its automatic interface designs. Task models have since been used in nearly every new model-based system. MASTERMIND was also one of the first systems to explore the idea of generating different interfaces for desktop computers, handheld computers, and pagers [Szekely 1996] by using the model to decide which information or interface elements could be removed from the interface as the size decreased. The interfaces generated for each different device used the same interaction techniques, which is not true of the dramatically different PUC interfaces generated for the PocketPC as compared to the Smartphone.

2.2.1.6 ITS

ITS [Wiecha 1990] is another model-based interface system, and was developed by researchers at IBM. The ITS system differs from other model-based systems in its explicit separation of concerns within its four-layer specification. ITS's layers consist of actions for modifying data stores, dialog for specifying control flow, style rules for defining the interface elements, layout, and language of the user interfaces, and style programs that instantiate the style rules at run-time. The layers are designed to make it easier for experts in different areas to collaborate on the interface design. For example, programmers would implement the actions and

style programs, while interface designers would write the style rules and application experts would specify the dialog. An important focus of ITS is making the dialog and style rules layers highly usable so that non-technical experts could be "first-class participants" [Wiecha 1990] in the design process. The design process was also very iterative; rules were expected to be continually refined until an acceptable user interface was created. Unlike many of the other model-based interface systems, ITS was used to create several commercial applications, including all of the kiosks at the EXPO '92 worlds fair in Seville, Spain.

2.2.1.7 TRIDENT

TRIDENT [Vanderdonckt 1995], a model-based system built around the same time as MASTERMIND, combines the ideas of an automatic interface generator with an automated design assistant. Like other systems, TRIDENT uses a task model, an application model, and a presentation model as the basis for creating interfaces. The TRIDENT system established a set of steps for its interface generation process: determine the organization of application windows, determine navigation between windows, determine abstractly the behavior of each presentation unit, map abstract presentation unit behaviors into the target toolkit, and determine the window layout. At each step, the interface designer could ask the system to perform the step using one of several techniques or do the work themselves. For example, TRIDENT determined layout using a bottom-right method that for each element would ask, "should this element be placed to the right or below the previous element?" A set of heuristics were used to automate the decision, or the interface designer could explicitly decide, often resulting in interfaces with a pleasing appearance. TRIDENT also used its task models, specified in a format called an Activity Chaining Graph (ACG), to automatically determine the number of windows needed for an application.

2.2.2 Model-Based Systems for Very Large Interfaces and Platform Independence

Later model-based systems concentrated on two features of earlier systems that were particularly successful: the use of task models to describe users' goals with an interface and the combination of multiple interface models to produce a final interface. With the advent of XML, another recent trend has been the development of user interface description languages (UIDLs) in an attempt to standardize the model formats used by different model-based systems.

2.2.2.1 Mobi-D

Mobi-D [Puerta 1997] is model-based user interface development environment capable of producing very large-scale user interfaces. The Mobi-D development process differs from previous systems in that a series of declarative models are created iteratively, starting with models of the users and their tasks and ending with a presentation model that represents the final interface. All of these models are stored together and many relations are described between the different models to assist the system and designer with their interface building and maintainenance tasks. Mobi-D also has many component tools for helping designers at various phases of the interface design process, from describing the users' tasks to the final guided assembly of the user interface. Assembly of interfaces in Mobi-D is a highly structured process, where the system steps the user through each of the sub-tasks associated with the interface and provides suggestions of appropriate controls for a particular task. Mobi-D, and the rest of the model-based systems for building large-scale interfaces, serve a different purpose than the PUC system and can be seen as complementary. It is conceivable that features from the PUC system, such as automatic modifications for consistency or aggregation of user interfaces, could be beneficial in larger scale interfaces. Models of the interface will be necessary to implement these features however, and these models might already be available for interfaces built with a model-based development environment, such as Mobi-D.

2.2.2.2 ConcurTaskTrees

Early options for specifying task models were the formal specification language LOTOS [ISO 1988] or GOMS [Card 1983], and many of the first model-based systems to use task models created their own languages for specifying the models. Recently, ConcurTaskTrees [Paterno 1997] has become popular as a language for representing tasks in several model-based systems (including TIDE [Ali 2002] and TERESA [Mori 2004]). ConcurTaskTrees is a graphical language for modeling tasks that was designed based on an analysis of LOTOS and GOMS for task modeling. ConcurTaskTrees extends the operators used by LOTOS, and allows the specification of concurrent tasks which is not possible in GOMS. ConcurTaskTrees also allows the specification of who or what is performing the task, whether it be the user, the system, or an interaction between the two. A special development environment was built for creating task models using ConcurTaskTrees called the ConcurTaskTrees Environment (CTTE) [Mori 2002].

2.2.2.3 XIML

The eXtensible Interface Markup Language (XIML) [Puerta 2002] is a general purpose language for storing and manipulating interaction data based on Mobi-D. XIML is XML-based and capable of storing most kinds of interaction data, including the types of data stored in the application, task, and presentation models of other model-based systems. XIML was developed by RedWhale Software and is being used to support that company's user interface consulting work. They have shown that the language is useful for porting applications across different platforms and storing information from all aspects of a user interface design project. It may be possible to express the information in the PUC specification language within an XIML document, but the language also supports many other types of information that will not be needed, such as concrete descriptions of user interfaces.

2.2.2.4 IBM PIMA and MDAT

The IBM PIMA project creates specialized interfaces for different platforms, including PDAs and phones, from a generic model of the application [Banavar 2004b]. This work incorporates automatic generation techniques, but differs from my work in two ways: 1) layout information can be included in PIMA's generic application model to help specialize the interface to different platforms, and 2) designers typically must "tweak" the specialized interfaces after generation. The PUC system uses Smart Templates and more sophisticated interface generation rules to address these issues. The work on PIMA has been integrated into the Multi-Device Authoring Technology (MDAT) project [Banavar 2004a], which focuses on the particular issues of authoring web pages that may be rendered on multiple platforms.

2.2.2.5 UIML and TIDE

The User Interface Markup Langauge (UIML) [Abrams 1999] claims to provide a highly-device independent method for user interface design, but it 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 TIDE interface design program [Ali 2002] has been implemented to address some of these issues with UIML. TIDE requires the designer to specify the interface more generically first, using a task model. Then the task model is mapped, with the designer's assistance, into a generic UIML model, which

is then further refined into a specific user interface. This process is very similar to that of PIMA, mentioned above.

2.2.2.6 TERESA

Transformation Environment for interactive Systems representations (TERESA) [Mori 2004] is a semi-automatic system for transforming user interfaces between different platforms. It allows designers to build their systems at an abstract level using the ConcurTaskTrees modeling language [Paterno 1997] and then transform that model into other models at different abstraction levels, including a concrete user interface for several different platforms. TERESA's current focus is on web applications, though in principle it could be extended to other environments. The current system is able to automatically generate interfaces, but in practice it seems to require designer involvement at each level of abstraction to create a usable interface.

2.2.2.7 USIXML

The USer Interface eXtensible Markup Language (USIXML) [Limbourg 2004] allows the specification of many different types of user interface models, including task, domain, presentation, and context-of-use models, with a substantial support for describing relationships between all of the supported models. The explicit goal of this language is to support all features and goals of previously developed UIDL's and as such it has many many features. USIXML appears to be complete enough to specify all of the features in the PUC specification language, but it is not clear how easy the language is to author or whether it is concise enough to produce specifications that can be easily handled by a resource-constrained device.

2.2.2.8 XAML and XUL

The eXtensible Application Markup Language (XAML) [Microsoft 2006] and the XML User interface Language (XUL) [Bojanic 2006] are two different languages for specifying a user interface developed by Microsoft and Mozilla respectively. XAML will be used in the next major revision of the .NET Framework, and eventually in the Vista operating system, to describe most graphics content that is rendered to the screen. XUL is currently used to define the user interfaces of all Mozilla browsers. Documents written in either language are similar to the presentation models used by many model-based systems, which abstract some platform-specific elements but are typically fixed to one interface modality with constraints on the form factor and input techniques. In this case, both languages are designed for large-

screen graphical interfaces. XUL has been shown to be beneficial for porting applications across various platforms of this type, including Windows, Linux, and Macintosh. The PUC specificiation language differs from these languages in that it describes appliance functionality without any specific details of the user interface, allowing the specification to apply for interfaces in different modalities and substantially different format factors with different input techniques.

2.2.2.9 SUPPLE

Most automatic interface generation systems, including the PUC, use a rule-based approach to create user interfaces. SUPPLE [Gajos 2004] instead uses a numeric optimization algorithm to find the optimal choice and arrangement of controls based on a cost function. The developers of SUPPLE have experimented with including a number of different factors in this cost function. Common factors to all of their functions are the cost of navigation between any two controls and the cost of using a particular control for a function. Additional costs have been included based on the common tasks that a user performs [Gajos 2004], consistency between interfaces for the same application generated on different platforms [Gajos 2005b], and the physical abilities of the user (for assistive technology) [Gajos 2006]. Figure 2.4 shows some example interfaces generated by SUPPLE.

SUPPLE's approach allows it to manage the trade-offs in an interface design by exploring the entire design space. This is somewhat more flexible than the PUC's rule-based approach, but also requires exponentially more processing as more variables and interface elements are considered. This means that SUPPLE's performance will degrade as the complexity of the user

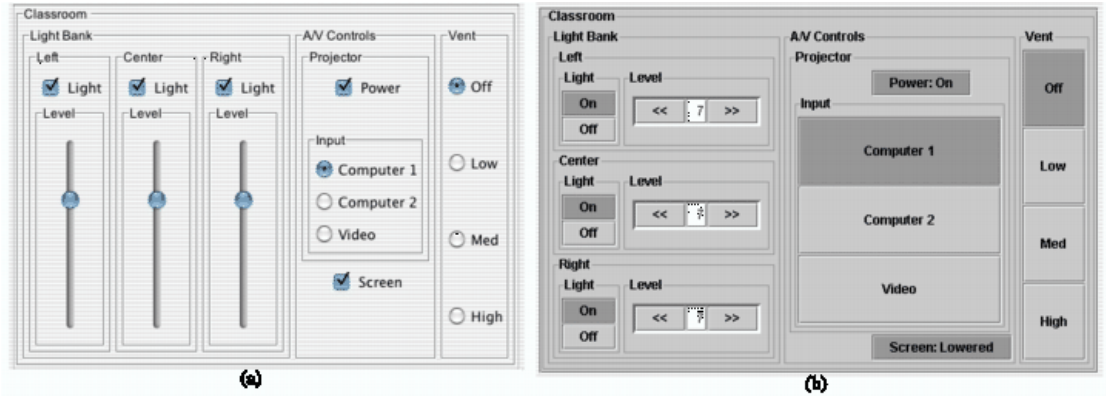


Figure 2.4. Examples interfaces for a classroom controller generated by Supple for different devices: a) a standard desktop computer with a mouse, and b) a touchscreen. The classroom has three sets of lights (with variable brightness), an A/C system, and an LCD projector with a corresponding motorized screen [Gajos 2004]. Reproduced with permission.

interface increases. Another difference is SUPPLE's interface description, which contains some of the same information as the PUC specification language but does not currently have a written syntax. Instead the description is defined by run-time objects created by a programmer, much like the second-generation UIDE system.

2.3 Aggregate User Interfaces

While there has been a great deal of research on automatically combining services, especially in the Web Services and Semantic Web communities, there has been very little work on combining the user interfaces. Work in Web Services typically takes the infrastructure as a given, and focuses on how existing standards such as WSDL, UDDI, and DAML-S [Sycara 2003] can be used to automatically discover and compose services. Unlike in my work, Web Service composition research seems to rarely consider creating user interfaces for combined services [Srivastava 2003]. One exception is an article [Staab 2003] which briefly mentions control issues. There has been work on user interfaces for specifying how services should be connected (e.g. [Kim 2004]) that could be leveraged in the future to build an editor that specifies how appliances are connected.

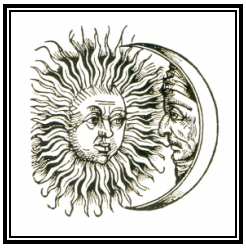
Several systems have explored the infrastructure issues that are involved in connecting and configuring systems of multiple appliances. One such system is Speakeasy [Newman 2002], which uses mobile code to allow arbitrary devices and services to interact, and also to distribute user interfaces to the handheld devices from which users interact. While Speakeasy might be able to automatically provide a wiring diagram for the PUC's interface aggregation feature, it does not provide support for automatically generating user interfaces or for combining user interfaces for multiple appliances into a single aggregate user interface.

I am aware of only two systems that have provided automatic combination of user interfaces. One is the ICrafter system [Ponnekanti 2001], discussed earlier, which can aggregate user interfaces for services that implement specific programmatic "service interfaces." Interface aggregators were implemented for specific combinations of service interfaces, which prevent ICrafter users from aggregating services in ways that the developers did not anticipate in advance.

The second system is the Gravity project [Hall 2003], which provides a mechanism for automatically constructing an application user interface based on a dynamic set of building blocks. These blocks might change based on the user's context, such as the location, envi-

ronment, task, etc. It seems that the focus with Gravity is on the framework issues, such as discovering available components and repairing the interface when a currently visible building block is no longer accessible. Currently, there seems to be no user interface integration between the building blocks, as each block is displayed as a separate panel within the Gravity application. An important focus of the PUC work is integrating functions from each appliance into a single interface.

Recently, aggregate user interfaces, known better as "mashups," have become popular among the Web 2.0 community [Merrill 2006]. The PUC's combination of functionality from multiple appliances can be seen as a type of mashup. The PUC's functionality differs from most mashups however. In part, this is because the PUC's mashups are produced automatically whereas current web mashups are produced manually by skilled web programmers. Another difference is that the PUC's aggregate interfaces are generated to perform a particular task and combine operations rather than data. Most web mashups combine one or more data collections with a set of interaction techniques for visualizing that data, often a map. While these visualizations can be useful, their value is in exploring the data in new ways rather than allowing the user to manipulating multiple web applications simultaneously to accomplish a specific task.



CHAPTER 3

Preliminary User Studies[1]

The previous chapter showed that automatically generating high quality user interfaces is a difficult problem. As a first step toward automatically generating remote control interfaces, we hand-designed control panels for two appliances, evaluated them for quality, conducted two user studies, and then attempted to extract the features of these control panels that contributed most to their usability. This approach helped us understand the features that a high quality remote control interface will have [Nichols 2002a, Nichols 2003], and then apply them in the interface generator software [Nichols 2002b].

3.1 Hand-Designed User Interfaces

Two common appliances were chosen as the focus of our hand-designed interfaces: the Aiwa CX-NMT70 shelf stereo with its remote control (see Figure 3.1a) and the AT&T 1825 tele-

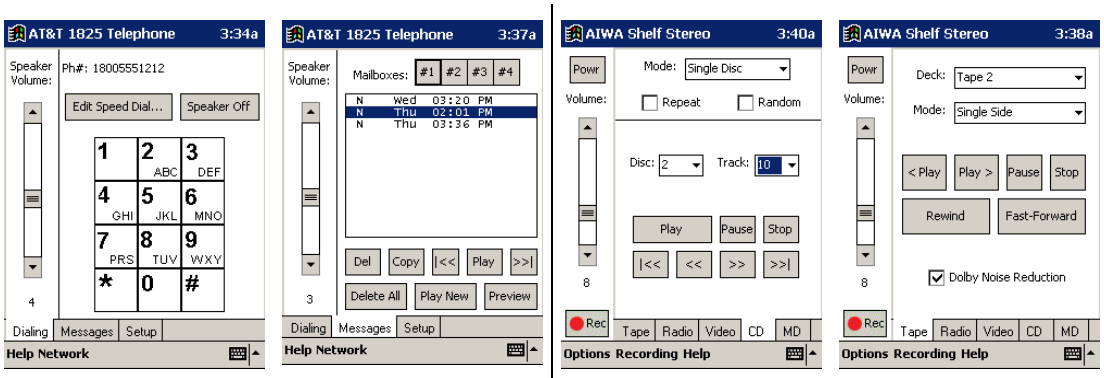
[1] The work in this chapter was originally described in Jeffrey Nichols and Brad A. Myers. "Studying the Use of Handhelds to Control Smart Appliances," in *Proceedings of the International Workshop on Smart Appliances and Wearable Computing* (IWSAWC). Providence, RI. May 19-22, 2003. pp. 274-279. and Jeffrey Nichols, Brad A. Myers, Thomas K. Harris, Roni Rosenfeld, Michael Higgins, and Joseph Hughes. "Requirements for Automatically Generating Multi-Modal Interfaces for Complex Appliances," in *Proceedings of the IEEE Fourth International Conference on Multimodal Interfaces* (ICMI). Pittsburgh, PA. October 14-16, 2002. pp. 377-382

a. b.

Figure 3.1. a) The Aiwa CX-NMT70 shelf stereo with its remote control and b) the AT&T 1825 office telephone/digital answering machine used in our studies.

a. b. c. d.

Figure 3.2. Paper prototypes of the phone (a-b) and stereo (c-d) interfaces for the Palm.



a. b. c. d.

Figure 3.3. Screenshots of the implemented phone (a-b) and stereo (c-d) interfaces for the PocketPC.

phone/digital answering machine (see Figure 3.1b). These two appliances were chosen because both are common, readily available, and combine several functions into a single unit. I owned the Aiwa shelf stereo that was used, and the AT&T telephone was the standard unit installed in many offices at Carnegie Mellon. Aiwa-brand stereos seemed to be particularly common, at least among the subject population, because ten of the twenty-five subjects in the user studies owned Aiwa systems.

Two sets of interfaces were designed for these appliances: low-fidelity paper-prototype designs for the PalmOS platform (see Figure 3.2) and functionally equivalent high-fidelity designs that were implemented in Visual Basic for Microsoft's PocketPC platform (see Figure 3.3). A different platform was used for the high-fidelity designs because of the availability of Microsoft's eMbedded Visual Basic tool, which made the implementation relatively painless.

Because of the complexity of both appliances, the Palm prototypes required approximately 20 hours to create and were improved with heuristic analysis prior to the user study. The PocketPC interfaces required more than fifty hours of design and implementation effort to create. The PocketPC interfaces were improved through a combination of heuristic analysis techniques and think-aloud studies with pilot users.

There were two goals with the design of these interfaces:

Functional completeness with the appliances: A goal of the automatically generated interfaces is to deal with the full complexity of the appliances, and it was important that the hand-designed interfaces would help in understanding how to address this goal.

Consistency with the conventions of the platform: An anticipated advantage of a UI device is that interfaces would be easier to use because the user would already have some familiarity with the platform and could leverage this knowledge to better use the appliance interfaces. Although there was no screening to ensure that our user study subjects were experts with the Palm or PocketPC platforms, an effort was made to ensure platform consistency in order to understand how difficult this goal would be for the automatic interface generators.

The study of the low-fidelity interfaces showed that it was important for subjects to feel in control of the appliances. Unfortunately, it was not possible to use the PocketPC to actually control either appliance, so instead software was created to simulate the appearance of controlling the appliances. A laptop with external speakers was connected to the PocketPC via a wireless network, allowing the user's actions to be transmitted to the laptop. The laptop then

simulated control by generating auditory feedback that was consistent with what would be expected if the PocketPC were actually controlling either of the appliances.

3.2 User Studies

Both studies were between-subjects comparisons of the hand-designed PDA interfaces and the interfaces on the actual appliances. The performance of the subjects was measured using several metrics, including the time to complete a task, the number of errors made while attempting to complete a task, and how often external help was required to complete a task. The purpose of these studies was to discover how users performed using the hand-designed interfaces versus the interfaces of the actual appliances and discover what aspects of the hand-designed interfaces were difficult to use. The studies of the low- and high-fidelity interfaces were very similar, so I will discuss the common procedure and evaluation methods before discussing the studies in detail.

3.2.1 Procedure

When each subject arrived, they were asked to fill out a consent form and a two-page questionnaire about their computer background and remote control use. Then each subject worked with two interfaces in one of four possible combinations to control for order effects. Each subject saw one actual interface and one handheld interface, and one stereo interface and one phone interface, neither necessarily in that order. For each interface, the user was asked to work through a set of tasks. When finished, a final questionnaire was given that asked whether the actual appliance or PDA interface was preferred and for any general comments about the study and the interfaces.

3.2.2 Evaluation

In order to compare the interfaces for both appliances, task lists were created for the stereo and phone. Each list was designed to take about twenty minutes to complete on the actual appliance, and the same tasks were used for both the handheld and actual interfaces. About two-thirds of the tasks on both lists were chosen to be easy, usually requiring one or two button presses on the actual appliance. Some examples of easy tasks are playing a tape on the stereo, or listening to a particular message on the phone. The remaining tasks required five or more button presses, but were chosen to be tasks that a user was likely to perform in real life.

These included programming a list of tracks for the CD player on the stereo, or setting the time on the phone.

It was anticipated that some subjects would not be able to complete some of the more difficult tasks. If a subject gave up while working with the actual phone or stereo, they were given the user manual and asked to complete the task. Subjects working on the prototype interfaces were allowed to press the "Help" button, available in some form on every screen. On the paper interfaces for the first study, a verbal hint was given, whereas the Visual Basic interfaces for the second study presented a scrollable screen of text, indexed by topic.

The performance of each subject on both lists of tasks was recorded using three metrics: time to complete the tasks, number of missteps made while completing the tasks, and the number of times external help was needed to complete a task. The time to complete the tasks was measured from the press of the first button to the press of the button that completed the last task. External help is any use of the manual for an actual appliance or the help screen on the PDA, or any verbal hint from the experimenter.

For the purposes of this study, a misstep is defined as the pressing of a button that does not advance progress on the current task. Repeated pressings of the same button were not counted as additional missteps. Sometimes a subject would try something that did not work, try something else, and then repeat the first thing again. If the interface had given no feedback, either visibly or audibly, the repeated incorrect steps are not counted as additional missteps. No missteps are counted for a task after the user has requested external help.

3.3 Study #1

The first study compared the actual appliance interfaces to our hand-designed paper prototypes (see Figure 3.2).

For the handheld portion of our experimental procedure, subjects were given a stylus and a piece of paper that showed a picture of a Palm V handheld device displaying the remote control interface. Subjects were instructed to imagine that the picture was an actual handheld, and to interact with it accordingly. Whenever the subject tapped on an interface element on the screen, a new picture was placed over the old one to show the result of the action. If auditory feedback was required, such as when the subject pressed play on the CD panel of the stereo (see Figure 3.2c), the test administrator would verbally tell the subject what happened.

3.3.1 *Participants*

Thirteen Carnegie Mellon graduate students volunteered to participate in this study, five female and eight male. All subjects were enrolled in the School of Computer Science, and all had significant computer experience. Seven owned Palm devices at the time of the study. Only one subject had no Palm experience and the remaining five had exposure to Palm devices in class or through friends. Everyone in the group had some experience with stereo systems. Only two did not have a stereo. Four subjects happened to own a stereo of the same brand used in this study.

3.3.2 *Results*

The results of the study indicate (all $p < 0.001$) that subjects made fewer missteps and asked for help less using the prototype handheld interfaces than using the actual appliances (see Figure 3.4). This indicates that the prototype handheld interfaces were more intuitive to use than the actual interfaces.

Time was not recorded for this study because we believed that delays created by the paper prototypes would dominate the time required to complete all the tasks. Even so, informal measurements suggested that subjects needed about one-half the time to complete all of the tasks using the prototypes as compared to the actual appliances.

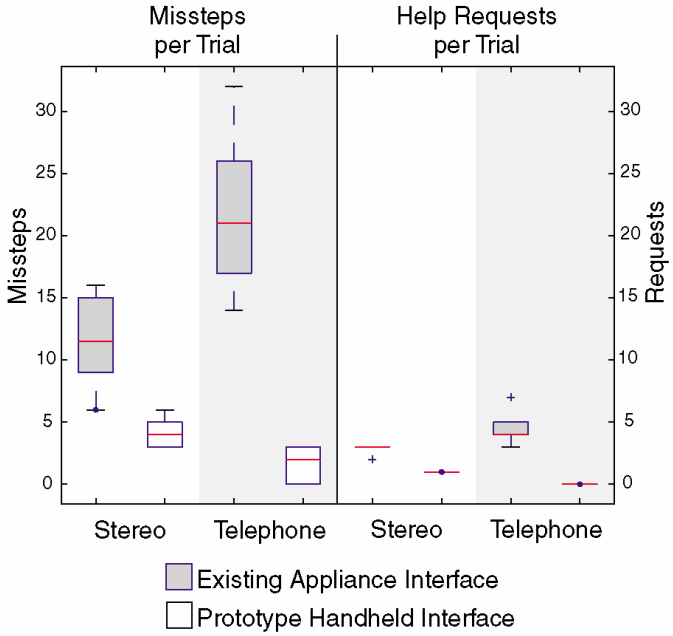


Figure 3.4. Box-plots showing the range of missteps and help requests (uses of external help) for each appliance and interface type.

3.3.3 Discussion

Users had great difficulty using the actual appliances, but were able to understand and operate the paper prototype interfaces with reasonable ease. One exception was found in the prototype stereo interface, which made use of the Palm's built-in menu system. None of our subjects navigated to screens that were only accessible through the menus without help, because they did not think to press the button that makes the menus visible. This was in spite of the fact that more than half used Palm devices regularly and were aware of the menu system. Although the study was successful, there was concern that the prototype interfaces benefited from the close interaction of the subject and the experimenter. In the paper prototype portion of the study, the experimenter provided all feedback to the user, including verbal hints when the user requested them. Because of these issues, a new study was conducted with full implementations of the interfaces so that the experimenter would be a passive observer instead of an active participant.

3.4 Study #2

The second study improved upon on the first study by replacing the paper prototypes with the full-fidelity PocketPC prototypes.

One very important issue with these interfaces was their use of several conventions that are specific to the PocketPC operating system. In particular, there is a standard OK button for exiting dialog boxes that is displayed in the top right corner of the screen. Users in pilot tests did not discover this feature, and thus were unable to exit from certain screens in the interface. The interfaces were not changed because of our goal to use the conventions of the controlling device. Instead, a tutorial program was created and presented to subjects before they began the study. The tutorial covers the OK button, text-entry, and the location of the menu bar, which is at the bottom of the screen instead of the top as on desktop computers.

3.4.1 Participants

Twelve students from Carnegie Mellon volunteered to participate in the study, in response to an advertisement posted on a high-traffic campus newsgroup. The advertisement specifically requested people with little or no knowledge of handheld computers. Subjects were paid US$7 for their participation in the study, which took between thirty and forty-five minutes to complete. Eight men and four women participated, with a median age of 22 and

an average of five years experience using computers. Subjects self-rated their skill at using computers for everyday tasks and their knowledge of handheld computers on a seven-point Likart scale. On average, subjects rated their knowledge of handhelds three points less than their skill with everyday computers (an average of 5.5 for everyday skill and 2.5 for handheld knowledge). Half the group owned Aiwa-brand stereos and two had AT&T digital answering machines.

3.4.2 Results

The results of the study indicate that subjects performed significantly better ($p < 0.05$ for all) using the PDA interfaces in all three metrics: time to complete the tasks, number of help requests, and number of missteps. Note that time could be measured in this study because there was no longer overhead from shuffling papers. Figure 3.5 shows box-plots comparing the handheld and actual interfaces for each metric on the stereo and phone respectively.

For both appliances, users of the actual interfaces took about twice as long, needed external help five times more often, and made at least twice as many mistakes as users of the PDA interfaces (note that this improvement is similar to that found when using the automatically generated interfaces compared with the actual interfaces, as described in Chapter 10).

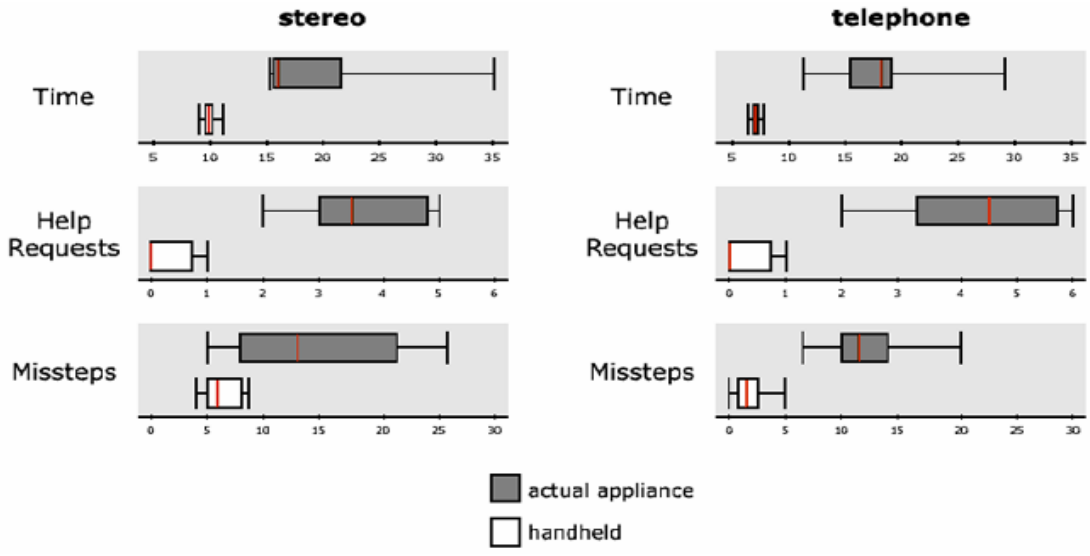


Figure 3.5. Box-plots of results from the second user study.

3.4.3 Discussion

The results of the second study are very similar to those of the first. Most of our subjects did not need to use external help to complete tasks using the handheld, and those that did use help only used it once. This compares to each subject's average of 3.6 uses of help for the actual stereo and 4.3 uses for the actual phone. Poor labeling, insufficient feedback, and the overloading of some buttons with multiple functions can account for this large difference on the actual appliances.

The worst examples of poorly labeled buttons and overloaded functions were found on the AT&T phone. This phone has several buttons that can be tapped quickly to activate one function and be pressed and held to activate another function. There is no text on the telephone to indicate this.

A similar problem is also encountered on the stereo. Setting the timer requires the user to press a combination of buttons, each button press within four seconds of the last. The stereo does not display an indicator to warn of this restriction, and often users were confused when a prompt would disappear when they had not acted quickly enough.

The phone also suffered from an underlying technical separation between the telephone and the answering machine functions. None of the buttons on the phone can be used with the answering machine. Even the numeric codes must be set using arrow buttons rather than the phone keypad. All but one subject tried to use the keypad buttons to set the code. The exception had used a similar AT&T phone in the past.

All of these problems were avoided in the PDA interfaces, because there was room for labels that were more descriptive and certain multi-step functions could be put on a separate screen or in a wizard. Using different screens to separate infrequently used or complex functions can also be problematic, however. Other buttons or menu items must be provided so that the user can navigate between screens, and the labels for these navigation elements must describe the general contents of the screen that they lead to. This was particularly a problem for the handheld stereo interface, which has more than ten screens. Many of the screens are accessible through the menu bar at the bottom of the screen. Subjects in the study and think-aloud participants before the study were very tentative about navigating the menus to find a particular function. In tasks that required the subject to navigate to a screen from the menu bar, the subject commonly opened the correct menu, closed the menu, did something wrong on the current screen, and then opened the menu again before finally picking the correct item.

The PDA stereo interface had other problems as well. In particular, the record function was difficult to represent in the interface because it was associated with tapes but needed to be available in all of the stereo's five playback modes: tape, radio, CD, etc. Although a record button was available on every screen (see Figure 3.3c-d), many subjects would get confused and incorrectly switch to the tape mode instead of pressing the record button. The red circle next to the text label on the "Rec" button was added after pilot testing to make the button more visible, because we thought that people tried the tape mode because they did not see the record button. This change seemed to have little effect, however.

3.5 Analysis of Interfaces

Once we were confident that our interfaces were usable, the interfaces were analyzed 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.

The prototype interfaces showed that finding groups of similar functions is important for constructing a good interface. These groups define how elements are placed relative to each other, and which elements can be separated across multiple screens. The different screens of the tab components are the best examples of grouping in our prototype interfaces (see Figure 3.2 and Figure 3.3). Grouping is also used to separate the mode, random, and repeat elements from the rest of the elements of the stereo CD player interface (see Figure 3.3c). These elements are used in all of the CD player's modes, while the other components are only used in half the modes.

Unfortunately, the visual groups cannot be specified explicitly because their members may vary between target platforms. For example, on a device with a small screen it might be necessary to separate the display of the current disc and track from the controls for playing a CD. It would not be appropriate if the PUC separated the play and stop buttons however. We noted that grouping information could generally be specified as a tree, and that the same tree could be used for interfaces of many different physical sizes provided the tree had sufficient depth. User interfaces designed for small screens would need every branch in the tree, whereas large screen interfaces might ignore some deeper branches.

It was also found that grouping is influenced by modes. For example, the Aiwa shelf stereo has a mode that determines which of its components is playing audio. Only one component can play at a time. In the stereo interfaces shown in Figure 3.3c-d you will note that a tabbed interface is used to overlap the controls for the CD player, tape player, etc. Other controls that are independent of mode, such as volume, are available in the sidebar. Unlike regular grouping information, information about modes gives explicit ideas about how the user interface should be structured. If two sets of controls cannot be available at the same time because of a mode, they should probably be placed on overlapping panels. We designed dependency equations to describe appliance mode information in our language.

The prototype interfaces also showed that feedback is important. One important way the interfaces provided feedback was by disabling, or graying out, a control so the user could tell when that function was not available. Many of the errors that users made with the actual appliance interfaces occurred when they pressed buttons for functions that were not currently available. This is another area where dependency information is helpful, because it defines exactly when the control(s) for a function should be disabled.

It was also noticed that most of the functions of an appliance were manipulating some data in a definable way, but some were not. For example, the tuning function of a radio is manipulating the current value of the radio station by a pre-defined increment. The seek function also manipulates the radio station value, by changing it to the value of the next radio station with clear reception. This manipulation is not something that can be defined based on the value of a variable, and thus it would need to be represented differently in the specification language.

Each of the interfaces used different labels for some functions. For example, the Palm stereo interface (see Figure 3.2c-d) used the label "Vol" to refer to volume, whereas the PocketPC stereo interface (see Figure 3.3c-d) used "Volume." This problem seems likely to be even worse for much smaller devices, such as mobile phones or wrist-watches. Thus it seems important for the specification language to include multiple labels that an interface generator could choose between when designing its layouts.

Finally, it was found that all of the interfaces used some "conventional" designs that would be difficult to specify in any language. At least one example of a conventional design can be found in each of the panes in Figure 1: (a) shows a telephone keypad layout, (b) uses standard icons for previous track and next track, (c) shows the standard layouts and icons for

play buttons on a CD player, and (d) uses the standard red circle icon for record. These conventions should not be specified as a part of the functional description of the appliance however, because they are not always applicable. The dialing pad convention does not make any sense for a speech interface, for example. I have developed a solution for addressing this problem called Smart Templates [Nichols 2004b], which is discussed in Chapter 7.

3.6 Requirements

The hand-designed interface work led to the development of a list of requirements that the PUC system must fulfill in order to generate high-quality interfaces. This section describes those requirements and briefly discusses how they are fulfilled (or not) by the PUC and other systems.

3.6.1 Two-Way Communication

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

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

3.6.2 Simultaneous Multiple Controllers

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

to create a multi-modal graphical and speech interface. Most current systems seem to fulfill this requirement.

3.6.3 No Specific Layout Information

The appliance specification should include information about the functions of that appliance, but it should not include specific information about how controls should be positioned on the screen. We share this philosophy with the V2 [INCITS/V2 2003] and Xweb [Olsen Jr. 2000] projects but not with UIML [Abrams 1999], which can include concrete information in its description about layout. This requirement enforces modality independence by limiting how detailed a designer can specify the functions of an appliance. If it were possible to describe concrete interfaces within an appliance specification language, designers would be tempted to include too many details about how each interface should be implemented. This has several disadvantages:

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

3.6.4 Hierarchical Grouping

A fundamental requirement of any user interface is good organization, because users must be able to intuitively find a particular function. An appliance specification can easily define organization using a tree to group similar functions. This makes the interface generation process easier, because most concrete interfaces can also be represented as a tree. The utility of trees for grouping seems to be universally accepted; most current systems use some kind of tree for grouping functions, although USIXML [Limbourg 2004] uses a graph.

3.6.5 Actions as State Variables and Commands

Each action that the user can take must be represented in the appliance specification. We found, as have many others, that state variables and commands are a succinct way to represent the manipulable elements of an appliance. Some systems, such as V2 and Microsoft's UPnP [UPnP 2005], separate the state variables from the commands that act upon them. This means that a specification for a radio might include a station variable, and also tune up, tune down, seek up, and seek down commands associated with the variable. The PUC system infers as many functions from the state variable as possible, but still uses commands for those functions that cannot be inferred, such as seek up and seek down.

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

3.6.6 Dependency Information

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

Dependency information may also be useful for generating help information, as in the UIDE system [Sukaviriya 1990]. UIDE used built-in pre- and post-condition information to determine why a particular function is not available and to generate instructions for making the function available. In our comparison study with the graphical interfaces we observed that users most often sought help when they wanted to use a function that was currently inactive.

Dependency information is similar to pre- and post-condition information and could be used to generate the same kind of help as UIDE.

As mentioned above, it is important that dependency information be in a form that can be analyzed by the interface generators. The original version of the V2 standard included dependency information, but the dependencies were defined as arbitrary ECMAScript expressions which were difficult, if not impossible, to analyze. This precluded the dependency information from being used for graphical layout, speech generation or command help. The PUC avoids this problem by specifying dependency information as a concise set of relations joined by logical operations. The PUC is the first system we are aware of that uses dependency information as an input to its automatic interface generator, and the current version of the V2 standard has adopted a similar approach.

3.6.7 Sufficient Labels

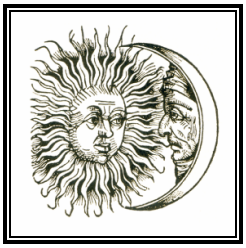
Our comparison study of the hand-designed interfaces with the actual appliance interfaces showed that good labels are an important part of creating a high quality user interface. Labels are an even more important part of speech interfaces, because there are no graphical hints to assist the user's understanding of the interface. To give flexibility to the interface generator, a label in an appliance specification should not be a single text string but instead a collection of text strings, pronunciation keys, and text-to-speech recordings. Pronunciation keys and text-to-speech recordings help improve the quality of the speech interface. Multiple text strings give a graphical interface generator the flexibility to select the label with the most information that can be fit in the allotted space. The PUC was the first system to provide more than just single string text labels, and this approach has since been adopted by the V2 standard.

3.6.8 Shared High-Level Semantic Knowledge

Despite all of the previous requirements, we must recognize that it is impossible to encode all the information into an appliance specification that a human would use to design an interface. In addition to functional information about the appliance, a human designer will also use his knowledge of conventions when creating an interface. There are many such conventions, such as the arrangement of buttons on a telephone number pad or the country-specific format for specifying dates. Conventions tend to be used across different types of appliances, but their usage will differ depending on the functionality of each appliance. For example, media controls like play, stop, and pause can be found on many appliances, but while most

devices will have play and stop, not all will fast-forward, rewind, next track, or previous track. Some appliances will add their own less common functionality, such as the play new button on some answering machines.

Describing every convention and how it should be applied for an appliance would require a lot of detailed specification and might violate our third requirement of not including any layout information in our specifications. Instead, I have developed an innovative flexible standardization technique call Smart Templates, which allows conventions to be standardized in advance, for appliance specifications to easily describe how conventions might be applied in the appliance user interface, and interface generators to appropriately render a convention based on the appliance, the controller device, and various properties of the user (such as locale). Smart Templates are discussed in more detail in Chapter 7.



CHAPTER 4

System Implementation

I have fully implemented the PUC system, allowing users to control real appliances through automatically generated interfaces on real handheld devices. The chapter overviews the PUC architecture, and discusses aspects of the implementation that are not discussed elsewhere.

4.1 Architecture

The overall architecture of the PUC system is shown in Figure 4.1. The PUC system has two main entities: the appliances that provide some service to the user, and controller devices that automatically generate and present remote control interfaces for the appliances. Appliances and controller devices communicate using a peer-to-peer approach, which allows any controller device to control multiple appliances simultaneously and any appliance to be controlled by multiple controller devices simultaneously. Two-way communication is required, because controllers must be able to send commands and receive both specifications and appliance state information from the appliances.

Appliances and controller devices communicate via a custom communication protocol that was designed to be operable across many different network layers, such as Wi-Fi, Bluetooth, or Zigbee, although our current implementations only support TCP/IP. Our protocol is

based on XML and its messages are designed around the functional elements that are supported by the PUC specification language. Unlike many other appliance control systems, such as UPnP, the PUC does not support automatic discovery of appliances. Currently, the user of a controller device must specify a server to connect to, either via an IP address or a name, and then the protocol can relay the name of any appliances that are controllable through that server. I considered including discovery features in the protocol, but decided this feature was outside the scope of my research. In the future, some discovery mechanism could be integrated with the PUC. In particular, it would seem best to use a location-based discovery system to ensure that users are controlling the appliances they intend to control and not, for example, their neighbors appliances.

Appliances are increasingly being built with communication protocols that allow two-way communication, but unfortunately these protocols are often proprietary and not compatible with the PUC protocol. To enable control of real appliances, we build *adaptors*: hardware and/or software that translate the appliance's proprietary protocol to the PUC protocol. In some cases, it is not possible to control the actual appliance. In these cases, we are forced to write adaptors that simulate the behavior of appliances.

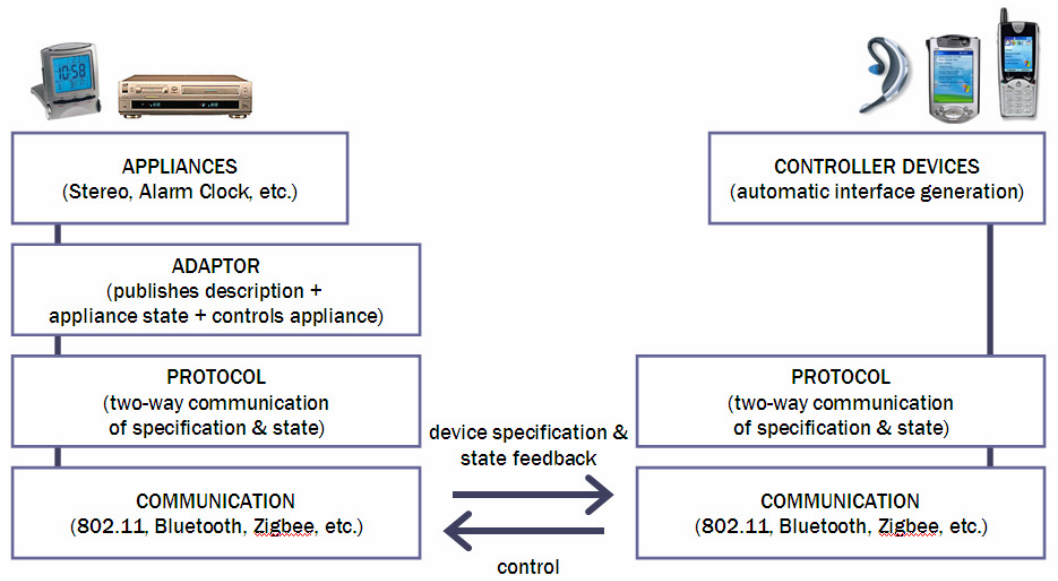


Figure 4.1. Diagram of the PUC system architecture, showing the communication between each of the different components.

4.2 Controlling Appliances

We have built adaptors for nine different existing appliances (see Table 4.1) and simulators for seven others (see Table 4.2). There is also a generic "Debug Server" that can read any PUC specification, create a simulator based that specification, and provide a user interface that allows the developer to adjust the state of the appliance as though that appliance was actually functioning. The user interface generated by the Debug Server uses a subset of the same algorithms as our other graphical interface generators, though it allows read-only state variables to be modified and does not show commands (see Figure 4.2).

Most of the adaptors we have built were created entirely in software and require the appliance to be attached to a PC in order to function. For example, the Sony Camcorder is controlled through the AV/C protocol running over an IEEE 1394 (Firewire) cable. Our adaptors for desktop applications typically use the Add-In capability provided by the developer, while our other adaptors communicate through the serial port or with general protocols such as UPnP. The one adaptor not written entirely in software is for the Audiophase Shelf Stereo, which did not originally have any mechanism for communicating its internal state to a third party. This appliance was modified by our collaborators at MAYA Design, a local Pittsburgh design firm, with custom hardware that electronically "watched" the LCD screen of the stereo and determined its state from the set of lights that were currently displayed. Control of the shelf stereo was enabled through the standard IR protocol used by the stereo's own remote control.

We also investigated building general adaptors for existing appliance control protocols, such as UPnP and HAVi. UPnP and HAVi both have their own appliance description languages, and the idea was to build a gateway for each that would translate that protocol's appliance description into a PUC specification. With the translated specification and mapping from one description language to another, the gateway could translate between the protocols on the fly. Unfortunately, we were not able to build these general adaptors for two reasons, one practical and one more fundamental. The practical reason we chose not to build a general purpose adaptor is that few appliances are designed to use these protocols, and those that are export only a few functions through the protocol. This makes the value in creating a general adaptor fairly low. Fundamentally, the problems with building a general adaptor were even worse. The description languages for UPnP and especially for HAVi were not nearly detailed enough to produce a reasonable PUC specification. For HAVi, the problem is that descrip-

tions are limited to a low-level description of the desired user interface including pixel locations and sizes for every control. Without links between the labels and their associated controls, it is very difficult to create a reasonable PUC specification. For UPnP, the descriptions provided functional information at the right level of abstraction, but there was no grouping information and few human-readable labels. PUC specifications could be created from this information, but the resulting interfaces would be poor at best.

Table 4.1. Appliance adaptors built by the PUC research team

Home Entertainment Appliances
Audiophase Shelf Stereo
Sony Camcorder

Lighting Controls
Intel UPnP Light
Lutron RadioRA Lighting
X10 Lighting

Desktop Applications
Microsoft PowerPoint
Microsoft Windows Media Player 9
PUC Photo Browser

Other
Axis UPnP Pan-Tilt-Zoom Surveilance Camera

Table 4.2. Appliance simulators built by the PUC research team

Home Entertainment Appliances
Panasonic PV-V4525S VCR

GMC 2003 Yukon Denali Systems
Driver Information Console
Climate Control System
Navigation System

Office Appliances
Canon PIXMA 750 All-In-One Printer
HP Photosmart 2610 All-In-One Printer

Other
Simulated Elevator

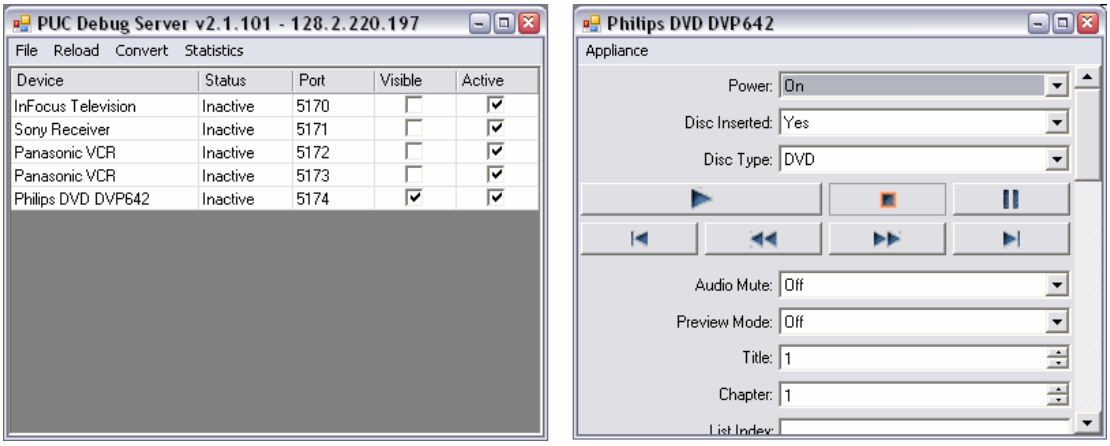


Figure 4.2. The interface for the PUC Debug Server. a) The main window showing the appliances currently being simulated by this server. b) The interface for simulating a Philips DVD player that was automatically generated by the Debug Server.

4.3 Generating Interfaces on Controller Devices

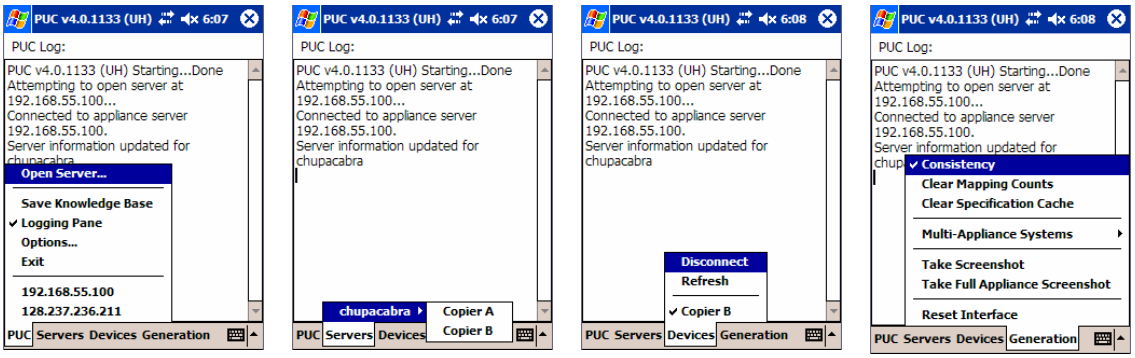
I have built graphical interface generators for three different platforms, Microsoft's PocketPC, Smartphone, and TabletPC, and collaborated on the creation of a speech interface generator with researchers from Carnegie Mellon's Language Technologies Institute (LTI). All interface generators speak the PUC communication protocol and generate interfaces from the specifications written in the PUC's specification language.

The speech interface generator creates interfaces using Universal Speech Interface techniques [Rosenfeld 2001] developed by my collaborators in LTI. This generator was implemented using a variety of speech technologies developed at Carnegie Mellon, including the Phoenix parser [Ward 1990] and the Sphinx recognizer [CMU 2006]. Unlike the other interface generators, the speech generator attempts to determine up-front which appliances the user will want to control and then it automatically generates a grammar that supports control for all of those appliances. A language model and a pronunciation dictionary are also automatically generated to assist the speech recognizer.

The graphical interface generators are all implemented in C# using the .NET Compact Framework 2.0. Even though the generators run on different platforms, using .NET has allowed me to share a substantial amount of code between each of the interface generators. This includes the code the implements the communication protocol, parses the specification language and performs other common tasks. Some interface generation rules can be shared among the platforms as well, but each platform also has its own unique generation rules. These rules will be discussed in detail in Chapter 8. Each of the graphical generators has its own custom interface for allowing the user to connect to appliances. Some custom controls were also implemented for the various generators to support the generation of list interfaces and add common controls that were not supported in the Compact Framework.

4.3.1 PocketPC and Desktop Implementation

The PocketPC and desktop generators have a similar menu-based interface that allows users to connect to appliances and generate user interfaces. The interface generator menus are used exclusively for generator functions and no appliance functionality is ever added to them. This decision was based in part of the preliminary user studies (see Chapter 3) that showed that subjects rarely looked in menus for appliance functions. There are four menus. The PUC menu (shown in Figure 4.3a) allows users to connect to a new appliance server or

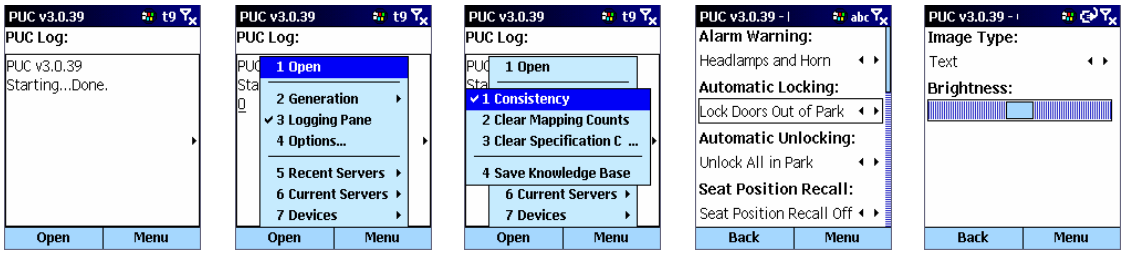


a. b. c. d.

Figure 4.3. Screenshots of the menu interface for the PocketPC PUC interface generator. The desktop interface generator has a similar menu structure. The backgrounds of all these screenshots show the logging panel where messages from the interface generator are displayed (primarily for debugging purposes).

choose a recently-used server. Appliance servers are machines through which multiple appliances may be attached. These servers maintain a list of all the appliances that are connected to them and send this list to controller devices when they connect. The Servers menu (shown in Figure 4.3b) shows the servers that the controller device is connected to, with a hierarchical menu for each server showing the appliances that are connected to that server. If the user selects an appliance from this hierarchical menu, then the controller device connects to the appliance (most likely through an appliance adaptor), downloads the appliance specification, and generates the user interface. Appliances connected to the controller device are displayed in the Devices menu (shown in Figure 4.3c). The Generation menu (shown in Figure 4.3d) gives access to various options that control the generation of user interfaces, including resetting the consistency system and opening the various screens of the multi-appliance system (see Chapter 9).

To support the generation of interfaces, it was also necessary to implement a number of custom controls to cover functionality that was not present in the .NET Compact Framework. For example, the Compact Framework does not have a panel that automatically adds scroll bars when controls are placed off the panel's viewable area. It was also necessary to implement a date-time picker, a slider, the combo box organizing panel (used for navigation between the three panels on the left half of Figure 8.1b), and a few different list widgets (see Figure 1.2b for a shot of a list widget displaying multiple steps in a route).



a. b. c. d. e.

Figure 4.4. Screenshots of interfaces for the Smartphone interface generator showing the menu-based interface (a-c) and custom controls built for the Smartphone (d-e).

The generator interface also has a logging panel that is primarily used for debugging, as shown in Figure 4.3. This view allows a user to track the progress of the PUC when generating a new interface and also reports when messages are received from appliance servers.

4.3.2 Smartphone Implementation

The Smartphone interface generation also has a menu-based interface, however the structure of the menus are slightly different because of differences in the Smartphone's menu design. Smartphone devices have two soft buttons underneath the screen, with the leftmost button invoking a common command and the rightmost button opening a single menu. This menu has hierarchical items that include the same functionalaity as the PocketPC and desktop generators. The initial view of the Smartphone interface (see Figure 4.4a) shows a logging panel with the two initial soft button labels at the bottom. From this screen the user can press the left soft button to open a connection to an appliance server or the right soft button to open the menu (see Figure 4.4b). The menu has hierarchical items corresponding to the server, device, and generation menus of the PocketPC and desktop generators. Recently connected servers are also placed in separate sub-menu to limit the number of items in the menu. The multi-appliance interfaces have not yet been implemented for the Smartphone, so only consistency options are available in the generation menu (see Figure 4.4c).

It was also necessary to implement custom controls for the Smartphone interface. Although the interface guidelines for the Smartphone called for three types of interface views (see section 8.1.2 for a more detailed description), no controls were provided in the Compact Framework for implementing these views. Thus, it was necessary to implement a list widget for the typical list-based interaction (see Figure 1.3a-c for several examples) and a widget for panel-based interactions that automatically scrolls as the user's input focus changes (see

Figure 4.4d). It was also necessary to implement a custom slider widget because the Smartphone version of the Compact Framework (v1.1) did not provide an interactive scrollbar. An example use of the custom slider control is shown in Figure 4.4e for controlling the brightness of a copier).

4.4 Communication

The communication protocol defines 11 different messages that may be sent between an appliance and a controller device. 6 of these messages may only originate from the controller devices (see Table 4.3), whereas the other 5 may only originate from an appliance (see Table 4.4). The protocol defines two additional messages that allow appliances to identify themselves to other appliances. In our current design we assume a lossless underlying networking technology, such as TCP/IP, and so our protocol does not include any messages whose sole purpose is to acknowledge the receipt of another message. The protocol also does not define a strict ordering among any of the messages; both controller devices and appliances must be prepared to receive any message at any time.

All messages in the PUC protocol are composed of two chunks of a data: an XML chunk followed by an optional binary chunk. The binary chunk is used to transmit data, typically images, which cannot be easily converted to text and transferred within the XML chunk. Both chunks of data are preceded in the message by two 4 byte fields: the first containing the length of both data chunks and the second containing the length of only the XML chunk (see Figure 4.5).

The data in the XML chunk must conform to our communication protocol schema, which defines each message type and each type's parameters. This schema can be found in Appendix C.1 and detailed documentation on the language is available online at:

`http://www.pebbles.hcii.cmu.edu/puc/protocol_spec.html`

Messages are designed around state variables and commands, which are the basic functional elements of the PUC specification (for more detail see section 5.2.1.1). Controller devices may send messages requesting changes to a state variable or the invocation of a command. In response, appliances will often send one or more state change notifications. These three messages are most common sent in the PUC protocol. State change notifications may also be sent when the controller device requests an update of an appliance's entire state. There is also

Table 4.3. Messages that may be sent by the controller device.

Name	Description
`state-change-request`	This message requests the appliance to change the designated state to the value contained in the message.
`command-invocation-request`	This message requests the appliance to invoke a command. This may cause state changes as side effects.
`spec-request`	This message requests the appliance to send a copy of its specification. It will send this via the `device-spec` message.
`full-state-request`	This message requests the appliance to send `state-change-notification` messages for every state that it has.
`state-value-request`	This message requests the appliance to send a `binary-state-change-notification` message containing binary data for a particular state.
`server-information-request`	This message is sent by controllers to get the list of appliances connected to a particular server.

Table 4.4. Messages that may be sent by the appliance to a controller device.

Name	Description
`state-change-notification`	This message is sent whenever the appliance changes state. The state name and its value are sent.
`binary-state-change-notification`	This message may be sent in two different contexts. It is sent when binary data has changed on the appliance and in response to a `state-value-request` message from the controller.
`device-spec`	This message contains the appliance specification. It is sent after receiving a spec-request message.
`alert-information`	This message contains a string message that must be delivered to the user.
`server-information`	This message is sent by the service discovery manager to let a controller know which appliances are connected to this server and whether the set of appliances has changed.

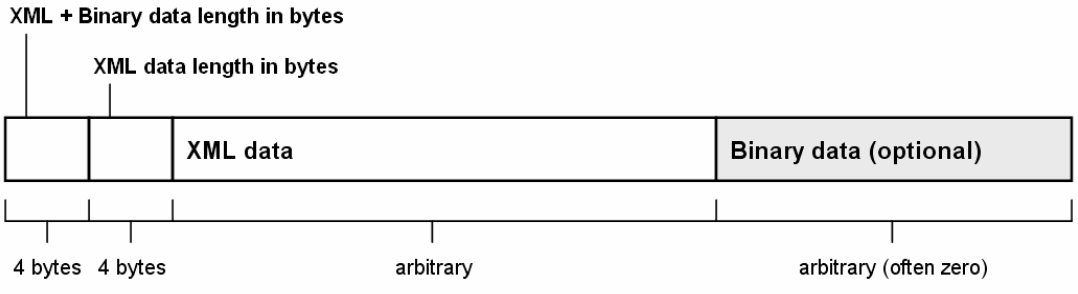
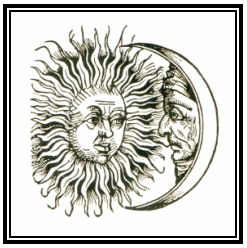


Figure 4.5. Message format for the PUC communication protocol.

a message for a controller device to request an appliance's specification and another message used by the appliance to send the specification document.

The PUC's support for binary data and for complex data formats, such as lists, also introduces the possibility that large amounts of data will need to be sent to the controller device whenever the appliance needs to update its state. To address this problem for lists, we have included four list operations that allow the appliance to only update the portion of a list that has changed rather than resending the entire list. We also considered adding a versioning system for lists to address problems that might arise if the user decided to interact with a list while the list's data was being updated. In practice list data synchronization has not been a problem for PUC interfaces however, so we have not yet implemented this feature.

The PUC protocol also handles binary data differently, both to limit the number of large data transfers and to allow the controller device to specify a binary format that is compatible with its capabilities. When a state variable with a binary type is changed on the appliance, a state notification is sent to the appliance with the name of the state variable but without the new binary data. The controller device must then explicitly request the binary data if it can handle the data appropriately. The PUC specification of a binary typed state variable may indicate that the appliance is able to manipulate the binary data that it sends to the controller (e.g. scaling an image). If the specification indicates that such as appliance feature is available, then the the controller may specify additional parameters in its request for the binary data (such as the size of an image). The appliance will send the binary data once the request is received from the controller device. This process prevents extra transfers of potentially large binary data in two ways: 1) controller devices that cannot process the binary data will not request it all and 2) if a binary state is updated rapidly then the controller device can condense these updates and request new data at the speed with which the controller device can receive it.



CHAPTER 5

Specification Language[2]

The PUC specification language was carefully designed to include just the right amount of information to generate high-quality user interfaces while also being easy-to-use and concise. An important goal of the language was to allow experienced specification authors to create a specification in the same amount or less time as an experienced user interface designer would need to create a user interface for one platform. Accomplishing this goal would address a problem with many previous automatic and guided generation systems, whose specifications required much more time to create than simply building the user interface by hand. Informally, our experience with the specification language suggests that we have come close to accomplishing this goal while providing enough detail for interface generators to produce high-quality interfaces.

This chapter starts with a brief discussion of the design principles that we used in the creation of the specification language, building on the requirements and analysis of the hand-designed interfaces discussed in Chapter 3. It then describes the design of the language in

[2] The work in this chapter was originally described in Jeffrey Nichols, Brad A. Myers, Kevin Litwack, Michael Higgins, Joseph Hughes, and Thomas K. Harris. "Describing Appliance User Interfaces Abstractly with XML," in *Proceedings of the Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages*. Gallipoli, Italy. May 25, 2004. pp. 9-16

detail, highlighting the language's unique aspects. Finally, I will conclude with some informal evaluation and observations about the specification language and its effectiveness.

This design of the language is shown through an example specification for a basic VCR appliance specification. This VCR has five functions that users can manipulate: power, the common media controls including record and eject, channel, TV/VCR, and a list of timed recordings that will take place in the future. There are also two status indicators for determining whether a tape is in the VCR and whether that tape is recordable. The VCR has only one physical input, a standard television antenna, and two physical outputs, an antenna passthrough and the standard three wire yellow/red/white plugs for composite video and stereo audio. All of these features can be described in the specification language, as will be shown below. The full specification for the simple VCR can be found in Appendix A; snippets of the simple VCR's specification will be shown as each of the language's features is described.

This chapter focuses on the conceptual aspects of the language with limited discussion of syntax. Readers interested in authoring specifications should see the complete language reference, which is included in Appendix B or can be downloaded from the PUC web site at:

`http://www.pebbles.hcii.cmu.edu/puc/specification.html`

5.1 Design Principles

Before and during the design of the specification language, we developed a set of principles on which to base our design. The principles are:

Descriptive enough for any appliance, but not necessarily able to describe a full desktop application. We found that we were able to specify the functions of an appliance without including some types of information that earlier model-based systems included, such as task models and presentation models. This is possible because appliance interfaces almost always have fewer functions than a typical application, and rarely use direct manipulation techniques in their interfaces.

Sufficient detail to generate a high-quality interface, as based on the hand-designed user interfaces discussed in Chapter 3. Note that this principle is different than the first. It would have been possible to completely describe the appliance without the readable labels or adequate grouping information that are needed for generating a good user interface. For example, the Universal Plug and Play (UPnP) standard [UPnP 2005] in-

cludes an appliance description language that does not include sufficient detail for generating good interfaces.

No specific layout information should be included in the specification language, following the system requirement described in section 3.6.3.

Support generation for different devices and modalities, especially for small devices and both the graphical and speech modalities. It is important to note that although the previous principle helps to address this one, this principle also suggests that specifications may need to contain extra information to enhance support for particular devices or modalities. For example, specifications may need to include labels with pronunciation or text-to-speech information to support the generation of speech interfaces.

Short and concise are very important principles for the design of our language. Appliance specifications must be sent over wireless networks and processed by computing devices that lack the power of today's desktop machines. To ensure performance is adequate, the specification language must be concise. Why then choose a verbose format like XML as the basis for our language? We chose XML because it was easy to parse and there were several available parsers. XML is also a very compressible format, which can reduce the cost of sending specifications over the network, though the PUC system does not use any compression.

Only one way to specify any feature of the appliance is allowed in our specification language. This principle makes our language easy to author and easy to process by the interface generator. It also makes it impossible for an author to influence the look and feel of user interfaces by writing their specification in a particular way. Some examples of design choices influenced by this principle are shown later.

5.2 Language Design

The design of the specification language has been developed over more than six years. Although new features, such as complex data structure support and content flow information, have been added since the initial version, the basic elements of the language have remained the same. There are two main categories of information that can be described with the language, functional and content flow, which are discussed in the following two sections.

5.2.1 Functional Language Elements

The focus of the language is on the functional aspects of appliances, which directly influence the design of interfaces for them. The functional elements of the language allow a specification author to describe the features that an appliance has and how those features relate to each other. The main features of the specification language are:

- The functions of an appliance can be represented by either state variables or state-less commands. State variables have specific type information that describes how they can be manipulated by the interface. Commands and states are collectively called *appliance objects*.
- Each state variable has type information, which describes the values that state variable may have and helps the interface generator decide how a variable should be represented in the final user interface.
- Label information is also needed in the specification so that users can understand the functions of the appliance. The specification language allows multiple values to be specified for each label, so that, for example, strings of multiple lengths can be provided for use in interfaces for screens of different sizes and pronunciation information can be provided for a speech interface.
- The structures of the hand-designed interfaces (see Chapter 3) were often based upon dependency information. For example, suppose that an interface was being created for a shelf stereo system with a tape and CD player. When the power is off, a screen with only a power button widget would be shown, because none of the other objects would be enabled. When the power is on, a screen is shown with many widgets, because most of the objects are active when the power is on. We might also expect this interface to have a panel whose widgets change based upon whether the tape or CD player is active.
- The final representation of any interface can be described using a tree format. It is not reasonable to include the tree representation of one interface in the specification of an appliance however, because the tree may differ for different form factors. For example, the tree will be very deeply branched on a small screen WAP cellular phone interface, whereas the tree will be broader for a desktop PC interface. The specification language defines a group tree that is deeply branched. It is expected that this

information could be used for small screen and large screen interfaces alike, because presumably some of the branches could be collapsed in a large interface.

- Complex data types can also be specified, such as lists and unions. The specification of complex data is based on the tree structure used for other portions of the appliance.

- It was important to use domain-specific conventions as much as possible in the hand-designed interfaces, so that users could leverage their knowledge of previous systems to use the interfaces. There is a need for some way to include this information in the appliance specifications and the Smart Templates technique was developed to address this problem.

Each of these items is described in detail below.

5.2.1.1 Appliance Objects

Three types of appliance objects are supported in the specification language:

- **States** - Variables that represent data that is stored within the appliance. Examples might be the radio station on a stereo, the number of rings until an answering machine picks up, the time that an alarm clock is set for, and the channel on a VCR. Each variable has a type, and the UI generator assumes that the value of a state may be changed to any value within that type, at any time that the state is enabled. It is possible for state variables to be undefined, i.e. without any value. This commonly happens just after an interface is generated before any values have been assigned, but could occur for other reasons.

- **Commands** – Commands represent any function of an appliance that cannot be described by variables. They may be used in situations where invoking the command causes an unknown change to a known state variable (such as the "seek" function on a radio), or in situations where the state variable is not known (due to manufacturer choice or other reason, e.g. the dialing buttons on a standard phone would all be commands). In the VCR specification, the Eject function is represented by a command (see Figure 5.1b). Commands in the PUC specification language cannot have explicit parameters, as they may in other languages such as UPnP. Where parameters are needed, the author can use state variables and specify dependencies that require the user to specify those variables before the command can be invoked. We could

have allowed explicit parameters, but this feature would have overlapped with state variables, increased the complexity of the language, and broken our "only one way to specify" principle.

- **Explanations** – Explanations are static labels that are important enough to be explicitly mentioned in the user interface, but are not related to any existing state variable or command. For example, an explanation is used in one specification of a shelf stereo to explain the Auxiliary audio mode to the user. The mode has no user controls and the explanation explains this. Explanations also represent an initial attempt at including help information within PUC specifications, though they are rarely used.

Although there are differences between states, commands and explanations, they also share a common property of being enabled or disabled. When an object is enabled (or active), the user interface widgets that correspond to that object can be manipulated by the user. Knowing the circumstances in which an object will be enabled or disabled can provide a helpful hint for structuring the interface, because items that are active in similar situations can be grouped, and items can be placed on panels such that the widgets are not visible when the object would not be active. This property is specifyied using dependency information, which is discussed in section 5.2.1.4.

Appliance objects also have an optional priority property, described as an integer value from 0-10, which specifies the relative importance of an object compared to the other elements in the same group. If the priority of an object is not specified, that object is assumed to be less

```
<state name="Channel" is-a="channel">
  <type type-name="ChannelType">
    <integer>
      <min>
        <constant value="2"/>
      </min>
      <max>
        <constant value="128"/>
      </max>
    </integer>
  </type>
  <labels>
    <label>Channel</label>
    <label>Chan</label>
  </labels>
</state>
```

a.

```
<command name="Eject">
  <labels>
    <label>Eject</label>
  </labels>
</command>
```

b.

Figure 5.1. Examples of a) a state variable representing the current channel tuned by the VCR and b) a command for ejecting the tape currently in the VCR.

important than other objects in the group for which a priority value has been given. The priority property was an early addition to the language and has not been as widely used as expected. In part, this is because the order that objects appear within the specification often describes the same information as priority and is easier to understand and manipulate than priority numbers. The PocketPC and desktop interface generators do not make any use of priority information, although the Smartphone does use this information in some of its generation rules.

5.2.1.2 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, the Channel state in Figure 5.1a has an integer type. We define seven primitive types that may be associated with a state variable:

- `binary`
- `boolean`
- `enumerated`
- `fixed point`
- `floating point`
- `integer`
- `string`

Many of these types have parameters that can be used to restrict the values of the state variable further. For example, the integer type can be specified with minimum, maximum, and increment parameters (see Figure 5.1a).

The enumerated type is for small collections of values, each of which has a label. Internally, these values are represented by numbers starting with 1. For example, an enumerated type with 4 items can have a value of 1 through 4. Enumerated types must have labels defined for each of their values.

The types of fixed point, floating point, and integer all contain numeric values. Integers do not have a decimal component, while fixed point and floating point both do. Fixed point values have a fixed number of digits to the right of the decimal point, as defined by the required "Decimal Places" field. Floating point values have an arbitrary number of digits on either side of the decimal point. The fixed point and integer types also have an optional increment field that can be used to further restrict the values that the state variable may contain. If an increment is specified, then a minimum value must also be specified. When these parameters are specified, the value of the state variable must be equal to the minimum + n * increment, where n is some integer.

The string type contains a string value. Specification authors can specify a minimum, maximum, and average length for the string contained in this variable. These parameters can affect the size of the text box that the generated interfaces will display for a state of this type.

It is important to note that complex types often seen in programming languages, such as records, lists, and unions, are not allowed to be specified as the type of a state variable. Complex type structures are created using the group tree, as discussed below.

5.2.1.3 Label Information

The interface generator must also have information about how to label appliance objects. 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 than 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 called a *label dictionary*.

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 [CMU 1998]) and text-to-speech labels that may contain text using SABLE mark-up tags [Sproat 1998] 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 to the user. 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 sufficient screen space, but still have a reasonable label to use if space is tight. Figure 5.2 shows the label dictionary for the Play Controls group of the VCR, which has two textual labels and a text-to-speech label.

```
<labels>
  <label>Play Controls</label>
  <label>Play Mode</label>
  <text-to-speech text="Play Mode" recording="playmode.au"/>
</labels>
```

Figure 5.2. The label dictionary for the playback controls group of the VCR. This dictionary contains two textual labels and some text-to-speech information.

5.2.1.4 Dependency Information

The two-way communication feature of the PUC allows it to know when a particular state variable or command is active. This can make interfaces easier to use, because the controls

representing elements that are inactive can be disabled. The specification contains formulas that specify when a state or command will be disabled depending on the values of other state variables. These formulas can be processed by the PUC to determine whether a control should be enabled when the appliance state changes. Five kinds of dependencies can be specified, each of which specifies a state that is depended upon and a value or another state variable to compare with:

- **Equals** - True when the specified state has the specified value.
- **GreaterThan** - True when the specified state has a value greater than the specified value.
- **LessThan** - True when the specified state has a value less than the specified value.
- **Defined** - True when the specified state has any value.
- **Undefined** - True when the specified state does not have any value.

These dependencies can be composed into Boolean formulas using AND and OR. NOT may also be used.

I 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 use of these formulas for interface generation is discussed later in Chapter 8.

```
<active-if>
  <equals state="Base.Power">
    <constant value="true"/>
  </equals>
</active-if>
```

Figure 5.3. An example of a common type of dependency equation specifying that a variable or command is not available if the appliance's power is turned off.

5.2.1.5 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 start time for a timed recording might be placed next to real-time control for the current channel, creating

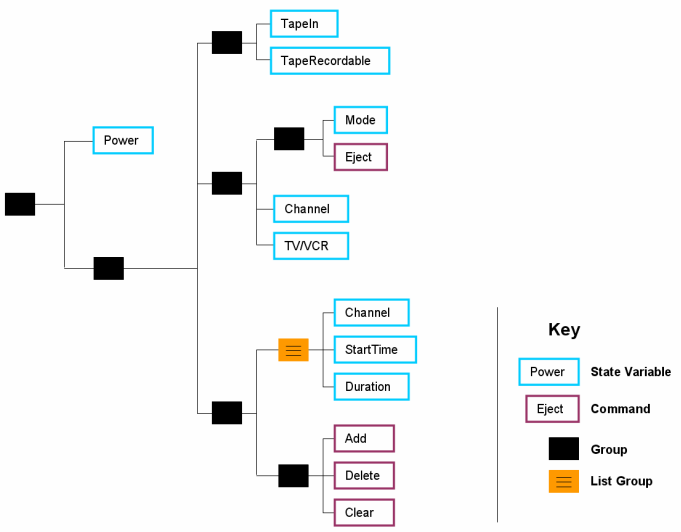


Figure 5.4. The group tree for the sample VCR specification.

an unusable interface. This is avoided by explicitly specifying grouping information using a hierarchical group tree.

The group tree is an *n*-ary tree that has a state variable or command at every leaf node (see Figure 5.4). 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. Designers are encouraged 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 controls across two screens. Interface generators for larger screens can ignore the deeper branches in the group tree and put all of the controls on one panel.

5.2.1.6 Complex Data Structures

The PUC specification language uses the group tree to specify complex type structures often seen in programming languages, such as records, lists, and unions. This approach simplifies the language and follows the principle of "one way to specify anything." If complex types were specified within state variables, then authors could have specified related data either as a single variable with a record data type or as multiple variables within a group. To support complex types, two special group elements were added.

Figure 5.5 shows an example of the `list-group` element added for specifying lists. Specifying a list group is similar to specifying an array of records in a programming language, and multiple list groups can be nested to create multi-dimensional lists. Each list group has an

implicit length state variable (named "Length") that always contains the current length of the list. If this variable is undefined, then the list currently has no members. The specification may define bounds on the length of the list in order to help the interface generator create a better rendering. An exact size may be specified, or a minimum and/or maximum size may be specified.

List groups also maintain an implicit structure to keep track of one or more list selections. The number of selections allowed may be defined in the specification ("one" and "many" are the only options currently), and the default is one if nothing is specified. If a list allows only one selection, then an implicit "Selection" variable is created which contains the index of the current selection (undefined means no selection). If multiple selections are allowed, then an implicit list group named "Selections" is created. This group contains a "Length" state (as all list groups do) and a "Selection" state which contains all of the selected indices.

We also developed a special dependency operator for lists, named `apply-over`. This element applies the dependencies it contains over some items in a list and returns a value depending on the value of its `true-if` property. The dependencies are applied over the set of items based on the `items` property, which may be set to *all* or *selected*. The `true-if` property may be set to *all* or *any*. If `true-if` is *all*, then the dependency formula contained in the `apply-over` element must be true for all elements in order for true to be returned. If `true-if` is *any*, then true is returned as long as the dependency formula contained in the `apply-over` is true for at least one of the elements.

The second special group is the `union-group`, which is similar to specifying a union in a programming language like C. Of the children within a union (either groups or appliance objects), only one may be active at a time. An implicit state variable named "ChildUsed" is

```
<list-group name="List">
  <labels>
    <label>Timed Recording</label>
  </labels>

  <min><constant value="0"/></min>
  <max><constant value="8"/></max>

  <selections access="read-write" number="one"/>

  <state name="Channel"> ... </state>
  <state name="StartTime" is-a="date-time"> ... </state>
  <state name="Duration" is-a="time-duration"> ... </state>
</list-group>
```

Figure 5.5. An example of a list group used in the VCR specification to describe the list of timed recordings that may be specified by the user.

automatically created within a union group that contains the name of the currently active child.

5.2.1.7 Smart Templates

Another important aspect of the specification language is Smart Templates. These templates allow the specification author to indicate that portions of a specification will have a high-level semantic meaning for the user. When an interface generator encounters these portions of the specification, it will attempt to generate an interface that matches the user's expectations. An author can specify that a Smart Template for a group, state variable, or command with the `is-a` parameter. For example, in Figure 5.1a the Channel state variable has been marked with the "channel" Smart Template.

Smart Templates are described in detail in Chapter 7.

5.2.2 Content Flow Language Elements

Information about content flow can be useful to describe the relationships between appliances that have been connected together in multi-appliance systems. The specification language allows authors to describe the input and output ports that an appliance possesses and the internal content flows that use those ports. Wiring information for a system of appliances can be combined with the port and content flow information from each appliance to build a model of content flow through the entire system. This content flow model is very useful for generating interfaces that aggregate functions from multiple appliances. The generation of aggregate user interfaces is described in detail in Chapter 9.

5.2.2.1 Ports

The input and output ports of an appliance define that appliance's relationship with the outside world. In order to match the user's intuitive understanding of ports, specification authors are encouraged to create a port for each of the physical plugs that exist on the outside of an appliance. Future tools could then use this information to help users correctly wire their systems. The specification language also supports port groups, which allow the author to give a single name to collection of related ports, typically ports that carry a piece of a larger content stream. For example, in Figure 5.6 the "Output" port group is a combination of the physical "Video" port with the physical "Left" and "Right" audio ports. The port group convenience makes it easy for specification authors to define that a video stream uses the "Output" port without needing to specify each of its consistuent ports.

```
<ports>
  <inputs>
    <port name="VHF/UHF Antenna" content-type="multi-channel-av" physical-type="coax" />
  </inputs>
  <outputs>
    <port name="VHF/UHF Antenna" content-type="multi-channel-av" physical-type="coax" />
    <port-group name="Output" content-type="av">
      <port name="Video" content-type="video" physical-type="RCA" />
      <port-group name="Audio" content-type="component-audio">
        <port name="Right" content-type="component-audio-right" physical-type="RCA" />
        <port name="Left" content-type="component-audio-left" physical-type="RCA" />
      </port-group>
    </port-group>
  </outputs>
</ports>
```

Figure 5.6. The ports section of the example VCR specification.

5.2.2.2 Internal Flows: Sources, Sinks, and Passthroughs

Each of the internal flow types is specified with three basic pieces: a dependency formula defining when the flow is active, a description of the ports associated with the flow, and a list of state variables, commands, and groups that can be used to modify the behavior of the flow. In the specification language, sinks are divided into two sub-types, recorder and renderer, that describe what the sink does with the content it receives. Figure 5.7 shows an example of a source content flow from the example VCR specification.

The ports that may be associated with a flow depend on the type of flow. Only output ports may be associated with a source, only input ports with a sink, and both are allowed for a pass-through. For each port, another dependency formula may be specified that defines when that port is active for that flow. Thus, to activate a particular port with a particular flow, both dependency formulas must be satisfied.

Channels are an important concept in content flow specifications. When a pass-through or sink receives a multi-channel input, a channel variable may be specified from the appliance that specifies the particular channel being tuned. The language can also specify that one channel of a multi-channel stream is being replaced by the appliance, which is used by example VCR specification to describe that the output of the tape source can appear on channel 3 (see Figure 5.7).

The set of variables, commands, and groups that modify the behavior of the flow is important for the generation of aggregate user interfaces. For example, the set allows the tint, brightness, and contrast functions of a television to be associated with the screen sink.

```
<content-group>
  <active-if>
    <equals state="Base.Power">
      <constant value="true" />
    </equals>
  </active-if>
  <content-group>
    <active-if>
      <equals state="Base.PoweredItems.Controls.TV/VCR">
        <constant value="true" />
      </equals>
    </active-if>
    <source name="Tape" content-type="av">
      <active-if>
        <equals state="Base.PoweredItems.Status.TapeIn">
          <constant value="true" />
        </equals>
        <not>
          <or>
            <equals state="Base.PoweredItems.Controls.PlayControls.Mode">
              <constant value="1" /> <!-- Stop -->
            </equals>
            <equals state="Base.PoweredItems.Controls.PlayControls.Mode">
              <constant value="6" /> <!-- Record -->
            </equals>
          </or>
        </not>
      </active-if>
      <output-ports>
        <port-group name="Output" />
        <port name="VHF/UHF Antenna" channel="3"/>
      </output-ports>
      <objects>
        <group name="Base.PoweredItems.Controls"/>
      </objects>
    </source>
```

Figure 5.7. The description of the video tape source content flow from the example VCR specification. Note that dependencies from the content groups that contain the source flow are ANDed with the source's own dependencies.

5.3 Evaluation of the Specification Language

Before discussing interface generation, there are several important questions to ask about the specification language:

- Is the language complete enough to specify the functionality of every appliance?
- Is the language easy to learn and use?

5.3.1 Completeness

Members of the PUC research group, including myself, several undergrads, a masters student and two staff members, have used the specification language to author specifications for 33 different appliances (see Table 5.1). We have tried to cover a large range of appliance types and to write specifications for several highly complex appliances, including a high-end Mitsubishi DVCR, a Samsung DVD-VCR combo player, two all-in-one printers from HP and Canon, and the navigation system from a GMC vehicle. Table 5.2 shows some statistics for the specifications that have been written so far. The table shows that PUC specifications on average are quite complex, particularly the GM navigation system specification which is

Table 5.1. Complete list of appliance specifications authored by the PUC research team

Home Entertainment Appliances
- Audiophase Shelf Stereo
- DirecTV D10-300 Receiver
- Gefen 2:1 HDMI Switchbox
- InFocus 61MD10 Television
- JVC 3-Disc DVD Player
- Mitsubishi HD-H2000U DVCR
- Panasonic PV-V4525S VCR
- Philips DVD Player DVDP642
- Samsung DVD-V1000 DVD-VCR Combo Player
- Sony A/V Receiver
- Sony Camcorder

Lighting Controls
- Intel UPnP Light
- Lutron RadioRA Lighting
- X10 Lighting

GMC 2003 Yukon Denali Systems
- Driver Information Console
- Climate Control System
- Navigation System

Office Appliances
- AT&T 1825 Telephone/Answering Machine
- Canon PIXMA 750 All-In-One Printer
- Epson PowerLite 770 Projector
- HP Photosmart 2610 All-In-One Printer
- Complex Copier
- Simple Copier

Desktop Applications
- Laptop Video Controls
- Microsoft PowerPoint
- Microsoft Windows Media Player 9
- PUC Photo Browser
- PUC To-Do List
- Task Manager

Alarm Clocks
- Equity Industries 31006 Alarm Clock
- Timex T150G Weather Alarm Clock

Other
- Axis UPnP Pan-Tilt-Zoom Surveilance Camera
- Simulated Elevator

Table 5.2. Maximum and average counts of various aspects of the PUC specifications written to date.

	Functional Elements	Groups	Labeled Groups	Smart Templates	Max Tree Depth	Ave. Tree Depth
Max*	136	64	46	30	11	6.20
Average*	36.25	16.93	11.6	6.21	4.79	3.59
GM Nav	*388*	*171*	*136*	*79*	*11*	*7.00*

* Not including GM navigation system specification

nearly twice as complex as any other specification. All of these specifications cover all of the functions of their appliance, giving us confidence that the language is capable of representing both the most common and most obscure functions of any appliance.

Although I cannot conclusively prove the language's completeness without writing a specification for every possible appliance, I believe there is sufficient evidence from the existing specifications to suggest that the language may be complete.

At the lowest level of description, we have seen in all of the specifications that state variables and commands are adequate for describing the functional elements of an appliance. At higher levels, the hierarchical group tree has been sufficient for representing organization and the dependency formulas have been descriptive enough to specify behavior while being restrictive enough to facilitate analysis that can be applied in the generated interfaces.

The main difficulty in the language design came from supporting complex data structures, particularly the lists that are found on many appliances. The design of these elements of the language was driven primarily by the GM navigation system specification, which contains many lists of complex data, such as destinations. Several iterations on this specification led to our current design, which combined grouping with list and union features and is capable of representing all forms of structured data. The design has since been used without modification on many other specifications, including those for many of the home entertainment and office appliances.

Another question in our design was support for interactive list operations, such as adding, deleting, and moving items. Such operations are quite common on appliances, such as manipulating a play list on an MP3, adding a timed recording on a VCR, or changing the route list on a navigation system. Standardizing these operations within the language did not seem appropriate, because there are many such operations, with many variations within a particular operation (e.g. add before, add after, add at end, etc.). Through experimentation with several specifications, including the navigation system spec, I decided to limit the language's expressiveness to list operations that can be specified using normal commands and state variables. This decision eliminates most direct manipulation operations, but has been sufficient for all of the specifications we have written. It may even be possible to support certain unrestricted direct manipulation operations, such as arbitrary moves, through the use of a Smart Template, though this has not been implemented in the current PUC system.

5.3.2 Learnability and Ease of Use

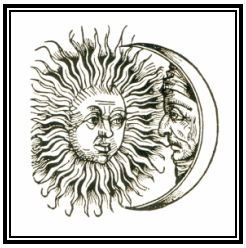
We have evaluated the learnability and ease-of-use of the specification language in one formal authoring study and many informal experiences with users both inside and outside of the PUC research group.

The formal study was conducted with three subjects who learned the language from reading a tutorial document (see Appendix D) and doing exercises on their own for approximately 1.5 hours. Subjects were then asked to write a specification for a low-end Panasonic VCR, which took on average 6 hours to complete. The focus of this study was on the consistency of the resulting specifications and not learnability per se, so the details of the study preparation and its specific results are not discussed here (see Chapter 6 for those details). The subjects were able to learn the language sufficiently in the short 1.5 hour period to write valid specifications for the VCR. This suggests that the language is very easy to learn.

Informally, we can draw some conclusions from the people who have learned and used the specification while working in the PUC research group. Over the course of six years, nine different people have used the language to write specifications for a number of different appliances. Each picked up the basics of the language in a day and was proficient within about two weeks.

Several people from the Technical University of Vienna and ISTI Pisa have used the PUC system and also learned the specification language (see Chapter 11 for more details). Although their specifications have not been as complex on average as those written by members of the PUC research team, they seemed able to learn the language from the online documentation easily and without needing to ask many questions via e-mail.

In all cases the most difficult aspects of specification writing seem to be identifying the variables and commands of an appliance, and organizing the variables and commands into the group hierarchy. I believe these tasks are inherently difficult however, and do not represent a weakness in the specification language. Experienced authors seem to develop a strategy where they start by identifying all of the variables and commands with little focus on organization, and then specify the group hierarchy after all variable and commands have been identified. Of course, identifying the variables and commands of an appliance may not be as difficult for the engineers that originally built the appliance. Thus, the specification language may be even easier for the makers of an appliance to use, once learned, than shown by the authoring study.



CHAPTER 6

Consistency[3]

The PUC system is the first to automatically generate interfaces that are consistent with interfaces the user has seen previously. This chapter discusses the meaning of consistency, both in general and for the PUC system, and describes the infrastructure needed for the PUC to generate consistent user interfaces. The rules for generating consistent interfaces are discussed along with the other interface generation rules in Chapter 8.

6.1 Understanding Consistency

Consistency has been a subject of research for the user interface community for many years, and there has been much debate about what consistency is and how to apply it effectively. According to Grudin, "a two-day workshop of 15 experts was unable to produce a definition of consistency" [Grudin 1989]. Reisner said that consistency is loosely defined as "doing similar things in similar ways" and that inconsistency occurs when "the designer and the competent user employ different assignment rules" [Reisner 1990]. Kellogg [Kellogg 1987]

[3] The work in this chapter was previously described in Jeffrey Nichols, Brandon Rothrock, and Brad A. Myers. "UNIFORM: Automatically Generating Consistent Remote Control Interfaces," in *Proceedings of the Conference on Human Factors in Computing Systems* (CHI). Montreal, Quebec, Canada. April 22-27, 2006. pp. 611-620

proposed that consistency does not have one single definition, but suggests that it can be defined within a framework with two types of consistency at three different levels. The two types of consistency are: internal consistency within an application, and external consistency between multiple applications. The three different levels are: conceptual, communication, and physical. Grudin [Grudin 1989] adds another type of consistency called analogical, which is meant to describe consistency with the world outside of the computer system.

Each of these definitions gets close to the concept of consistency, but none helps us operationalize consistency within an automatic system. Fortunately, there is a fair amount of work examining the factors that lead to consistency (or lack thereof). Most of this work breaks the user interface down into a formal model, usually a set of production rules, which describes the actions that users must take in order to accomplish their tasks.

Barnard [Barnard 1981] found that "positional" consistency is important within a command language, meaning that common parameters between functions should always be placed in the same position. This work was conducted on command line interfaces however, and may not apply to today's graphical user interfaces (GUIs).

Reisner [Reisner 1981] explored consistency through two drawing applications and found that users made fewer mistakes with the interface that was "structurally" consistent. In Reisner's case, structural consistency is defined in terms of the production rules describing the interface. An interface where all of the rules for a similar action have the same form would be structurally consistent, whereas an interface that uses rules of several different forms would be inconsistent. For example, shape selection in the first application was inconsistent because most shapes were selected by pressing the select button, flipping the switch for the desired shape type, and pressing the GO button, whereas no action was required to select a text shape because the keyboard was always available. The second application used the same rule, moving the cursor to the box specifying the desired shape type, and was thus structurally consistent. The consistency that Reisner explores is within an application however, and she did not examine how users who learned the first drawing application subsequently performed on the second. It is this latter form of consistency that the PUC system aims to provide.

Polson and his collaborators showed that the "common elements" theory of knowledge transfer from experimental psychology could account for the positive transfer effects observed from the usage of consistent user interfaces [Polson 1986]. In this case, the common elements were the shared steps needed to accomplish the same task in two different

applications. Through several experiments with word processing applications, Polson showed that the effects of transfer due to a consistent interface were substantial in several cases: within an application, between different versions of the same application, and between different applications [Polson 1988]. In all cases where transfer occurred, there were a large number of external cues to indicate to users that previous knowledge might apply. This suggests that the PUC must not only create new interfaces with steps that are similar to previous interfaces, but it must also make the new interfaces appear visually similar to the old interfaces so that users will realize that their previous knowledge can be reused.

There has also been work exploring how user interfaces for the same application can be made consistent across different platforms. Denis & Karsenty [Denis 2003] describe some of the common problems that arise when creating multi-platform applications, such as differing sets of functions, different partitioning of functions and data, and the ability to recover context when changing platforms. They identify that these are all problems of *continuity*, which they break down into knowledge continuity and task continuity. Knowledge continuity refers to properties of the interface, such as the labels and layout, and task continuity refers to the ability of users to change devices in the middle of their workflow. They suggest a set of design principles for dealing with these problems, such as using similar labels and organization across all interfaces. Florins [Florins 2004] expands on these ideas by relating them to existing theory and breaking the idea of knowledge continuity into three parts: perceptual, cognitive, and functional. Perceptual continuity is found when interfaces have a similar appearance. Cognitive continuity occurs when users understand the underlying concepts behind the interface. Functional continuity occurs when the same set of functions are available on both platforms. Although both Denis & Karsenty and Florins suggest some design guidelines to address continuity, none of these guidelines are sufficiently concrete to apply in the PUC's automatic interface generation.

The work thus far identifies benefits of consistency, but other researchers have found that there may be downsides if consistency is carried too far. In particular, Grudin [Grudin 1989] shows five examples where the consistent design decision is not beneficial for ease of learning, ease of expert use, or both. Grudin's conclusion is that the most important factor in an interface design should be matching the user interface to the user's tasks, not on consistency. Consistency should only be explicitly considered if it is beneficial to the user's tasks. I agree with Grudin's position and the PUC's consistency generation rules always favor usability of the interface over consistency.

My generation techniques are also able to avoid at least two of Grudin's issues with consistency. First, consider Grudin's illustrative example of the placement of knives throughout a house. Placing all knives in one drawer would be the consistent design, Grudin argues, because all knives would be in one central, easy-to-find location. The usable design, however, is to place knives in the location where they will be used (e.g. table knives in the kitchen, putty knives in the garage, swiss army knives with the camping equipment, etc.). In this example, there are many different placements for knives and, although Grudin has declared one particular design consistent, the best particular design for any given user is the configuration that user expects. I call this configuration the *personally consistent* design, because it is consistent with the user's personal view of where knives should be located around the home. The PUC interface generators attempt to achieve personal consistency, meaning that functions should have the appearance and location that the user expects. In this way, the PUC attempts to avoid the inconsistency that occurs when "the designer and the competent user employ different assignment rules" [Reisner 1990].

Grudin also notes that ease of learning and ease of use may require conflicting designs, which prevents a consistent design from being used between interfaces optimized for different purposes. Personal consistency helps here, because the PUC can generate interfaces that suit each user's particular situation: either a novice just learning to use an appliance's functions or an expert that is familiar with many appliances of the same type. What happens when the novice user has learned the interface and wishes to become an expert though? In this case, the flexibility of automatically generating interfaces allows the PUC to regenerate the interface with the design for experts in place of the design for novices.

6.1.1 Evaluating Consistency

The process of evaluating the consistency of an interface can be time-consuming and difficult. A number of systems have been created to help automate the collection of usage data that can help in the evaluation of consistency [Ivory 2001]. Sherlock and GLEAN have specific features for automatically evaluating the consistency of an interface. Sherlock [Mahajan 1997] uses a heuristic approach to evaluate task-independent qualities of user interfaces for consistency. This includes looking at visual properties, such as widget sizes and font choices, and terminology issues, such as inconsistent abbreviations and spelling errors. The system is capable of evaluating dialog boxes from Visual Basic and Visual C++ 4.0, but in principle the tool could be run on any user interface that is translated into Sherlock's textual interface de-

scription language. The output of Sherlock is a collection of tables that display each of the metrics. Designers look for consistency problems by searching the tables for outlier values.

GLEAN [Kieras 1995] makes predictions about human performance on a user interface based on a GOMSL (GOMS—Goals, Operators, Methods, and Selection Rules—Language) model. The use of GOMSL is important because of its unique ability to accurately predict transfer times between tasks [John 1996], which is useful for finding consistency problems between similar tasks. Unfortunately, a GOMSL model must be created for the user interface before any analysis can be conducted. There is no way to automatically generate such a model, though recent work has shown that it is possible for other types of GOMS models [John 2004].

6.1.2 Applying Consistency

A number of methods have been developed to help ensure interface consistency. Platform interface guidelines and toolkits, like those developed by Apple for the Macintosh, help designers make their applications consistent with others on the same platform. Usually these guidelines work best for common functions, such as by defining a standard menu structure that makes it easy to open and save files and access clipboard functions. More specific consistency guidelines have been proposed, such as those described by Nielsen [Nielsen 1993]. These include maxims such as "the same information should be presented in the same location on all screens." Another mechanism for maintaining consistency is called "design languages" [Rheinfrank 1996], which are used by designers to ensure that common features and branding are shared across a family of products.

ITS, SUPPLE, and DiamondHelp are all systems that address the consistency of user interfaces with automatic design. The ITS system [Wiecha 1990] was successful in producing consistent interfaces across a family of applications (such as all the displays for a World's Fair) and for multiple versions of the same application. Interfaces were generated using a rule-based approach, and consistency resulted because the rules applied the same interaction technique in every place where the same condition was found. Note that although the PUC does achieve some consistency by using the same generation rules for each interface, it cannot rely on the underlying appliance models being the same. Appliances of the same type may have substantially different underlying models, and one of my contributions is finding similarities between underlying models and creating consistency based upon these similarities.

SUPPLE [Gajos 2004] automatically generates layouts for user interfaces using an optimization approach to choose controls and their arrangement. Some initial research has been conducted on adapting SUPPLE to support the creation of consistent user interfaces for the same application across different platforms [Gajos 2005b]. Like ITS, SUPPLE cannot create consistent interfaces if the underlying appliance models differ.

DiamondHelp [Rich 2005] attempts to address the consistency problem for consumer electronics devices by combining a task-model based approach with a direct-manipulation interface. While one of DiamondHelp's goals is to provide consistency, the current system relies on designers to create the direct-manipulation portions of the interface and for each appliance to supply its own task model. DiamondHelp does not provide a way to search for possible inconsistencies across devices or to automatically adjust the interfaces to help the user transfer knowledge between interfaces.

6.2 Specification Authoring Study

I started work on consistent interface generation by studying how inconsistencies can arise in the user interfaces created by the PUC. The PUC interface generator uses a rule-based process (see Chapter 8) that is guaranteed to produce the same interface given the same appliance specification, so any inconsistencies arising in the interface will be due to differences in the specifications of two similar appliances. In order to understand how specifications can differ, I conducted two studies to investigate the following questions:

- How can specifications vary for different appliances that share similar functions?
- How can specifications vary for different authors?

These studies focus on specifications for VCRs, which require the use of many specification language features that are usually needed to specify complex appliances (e.g. lists, smart templates, etc.).

6.2.1 Study #1

The first study addressed the question of how specifications can vary for different appliances with similar functions by examining specifications that I wrote for three different VCRs. Two of the VCRs were complicated with many features, a Mitsubishi HD-HS2000U Digital VCR and a Samsung DVD-V1000 DVD-VCR combo-player, and the final VCR was the cheapest model that we could find at our local Best Buy store: a Panasonic PV-V4525S. I

wrote these specifications to ensure that the quality was high. I did not recruit other authors for this study because I wanted to control for differences that might arise between authors. I also tried to follow the manufacturers' designs for their appliances as much as possible, to be sure that any differences in specifications are due to differences in the appliances. All three specifications took me a total of about one week to complete.

In order to analyze the VCR specifications, I identified the state variables and commands, hereafter referred to as objects, which seemed to be shared across the appliances. Some objects were identical, such as the counter and status variables that tracked whether a tape was in the VCR. Many objects were similar but not completely identical. For example, the only differences between some objects were the labels, such as for the "TV/VCR" or "VCR/TV" Boolean states that are present on each VCR. Other objects contained some of the same values, but also supported other features that were not present across all of the VCRs. For example, all of the VCRs have a state variable that specifies whether the coax input is coming from the antenna or cable. The Panasonic VCR supports only these two options, and the Samsung adds an extra option called "Auto" in which it will automatically select the appropriate value. The Mitsubishi VCR does not have the auto value, but it supports two additional input types not found on the other VCRs ("cable box" and "digital cable").

Other functions shared across all of the VCRs were specified quite differently. For example, each of our VCRs supports a timed recording feature to specify TV programs that the user wishes to record in the future. The way to specify the time that the program would be recorded differed across devices. The Mitsubishi and Panasonic VCRs both have variables for the start time and stop time of the recording, while the Samsung has a matching variable for start time but a different variable that specifies the duration of the recording. In this case, the underlying data is quite different even though the function is identical.

We also analyzed the organization of the VCR specifications and found a few differences. In general, it seems that most of the same high-level groups were shared between the specifica-

a) Mitsubishi HD-HS2000U b) Samsung DVD-V1000 c) Panasonic PV-V4525S

Figure 6.1. VCRs used in the first study. The Panasonic VCR in (c) was also used in the second study.

tions, though the exact placement of those groups varied somewhat. For example, all of the VCRs have the Power state at the top-level with groups for Status and Setup. The Mitsubishi and Panasonic VCRs also have groups for Controls and Timed Recordings at the top-level. The Samsung DVD-VCR has these same groups, but they are located in a top-level VCR group because this appliance also must support its DVD and MP3 players.

6.2.2 Study #2

The second study examined the variations in specifications written for the same VCR by several different authors. For this study, I was particularly interested in seeing how organization varied between specifications. I used the Panasonic VCR from the first study and recruited 3 students in our university's electrical and computer engineering department to be subjects. I chose these subjects because I expect that specification authors in industry would have this background.

Unlike in the first study, these subjects had no knowledge of the PUC specification language when they started. Before writing the VCR specification, subjects were trained on the language through a written document with several exercises and examples from a to-do list application. I chose to use written training to ensure that the learning experience was the same for each subject. The to-do list application was used as an example because it incorporated every feature of the language, but was different enough from the VCR that it was unlikely to affect the subjects' specifications. Training and authoring took a substantial amount of time, about 1.5 hours and 5 hours respectively, so we allowed subjects to take the materials and VCR home with them and complete the study over the course of two weeks. Subjects were paid for their participation: $15 for completing the training and $50 upon returning the VCR and turning in a valid specification.

All three of the subjects' specifications contained two top-level groups for setup functions and basic controls. All also had a group for timed recordings, but not all placed the group in the same location. Two of the three made timed recordings a top-level group, while the other chose to place it in the basic controls group. Two of the three had an advanced controls group, with one placing this group at the top-level and the other putting it inside the basic controls group. Within the common groups, the subjects used different strategies to further organize the functions. For example, one subject organized functions based on whether they belonged to the TV and VCR, using this method to organize within both the basic controls and setup groups.

The subjects also placed objects at different locations in their hierarchy. For example, the repeat-play command was put in the advanced playback controls group of one specification and in the setup group in another one. The functions were also defined differently in some cases, as one subject used commands for the play, stop, and pause buttons while the other two used state variables.

6.2.3 Discussion

In these studies I found that specifications will have differences, even if written by the same author or for the same appliance. These differences may be found in the specification of similar functions and the organization of these functions. The number and variety of differences was particularly surprising and demonstrates the challenges the PUC faces when creating consistent interfaces. Next, I will combine these results with prior work on interface consistency to synthesize a set of requirements for consistency in the PUC system.

6.3 Requirements for Consistency

For the PUC, we define a consistent user interface as one that is easier to learn because it incorporates elements and organization that are familiar to the user from previous interfaces. In the context of appliance interfaces, this might mean that a new copier interface is easier to learn because it uses the same labels as a previously learned copier interface (see Figure 1.5) or that the clock is easier to set on a new VCR interface because the clock controls are located at the same place in the interface hierarchy. In order to facilitate this knowledge transfer between interfaces, Polson suggests that tasks must have similar steps and there must be sufficient external cues in both applications [Polson 1988]. To facilitate this, the PUC has the following requirements for its consistent user interfaces:

R.1 Interfaces should manipulate similar functions in the same way

R.2 Interfaces should locate similar functions in the same place

These two requirements help to ensure that user tasks will have similar steps, which can facilitate knowledge transfer. They also illustrate a fundamental separation in the PUC between *functional consistency* and *structural consistency*. Two interfaces are functionally consistent if the same set of controls is used for similar functions. Two interfaces are structurally consistent if similar functions can be found in similar organizational groups. These two types of consistency are independent and are addressed separately by the PUC.

In order for knowledge transfer to occur, there also must be sufficient external cues to indicate that the applications are the same. In many cases, the PUC gets an important external cue for free, because users are often aware of the type of appliance they are using and will have some memory of using similar appliances in the past. To reinforce that cue, we have the following requirements to help increase users' perceptions of consistency between interfaces:

R.3 Interfaces should use familiar labels for similar functions

R.4 Interfaces should maintain a similar visual appearance

The studies show that situations may arise where these requirements cannot be followed. For example, similar functions may have different representations that cannot use the same control. Unique functions may also affect the order in which controls appear on the screen or affect the layout if they require larger controls or have wider labels. In these situations there is a fundamental trade-off between maintaining consistency to a previous interface and appropriately rendering all of the new appliances' functions. To address this problem, the PUC could go against Grudin [Grudin 1989] and favor consistency. In this case, the PUC would move the unique functions to a separate panel so that they could not affect the layout. This solution has many negative consequences for usability however: important functions could be moved to a non-intuitive location, and the extra features of a similar function might appear to not exist. It seems better to favor usability in these situations, and therefore the PUC has the following requirement:

R.5 Usability of unique functions is more important than consistency for similar functions

I have found that a common result of this requirement is that our consistent user interfaces do not always have a similar visual appearance. However there may be some benefit to having a different visual appearance: work by Satzinger [Satzinger 1998] found that users were able to learn the user interface for a conceptually similar application more easily when the interface used the same labels but had a different visual appearance.

The first five requirements apply to the user interfaces that the PUC generates, and illustrate the actions that the PUC will take to ensure consistent interfaces. A pre-requisite for all of these requirements however, is:

R.6 Interface generators must provide a method to find similar functions across multiple appliances

Although this is a general requirement for any system that wants to create consistent interfaces, the implementation of this method is likely to be specific to the particular type of input that the system receives. In the case of the PUC, the input is written in the specification language, which provides a functional model of each appliance. The PUC's method for finding similarities may be applicable to other systems that use functional models, but may not apply to systems that use other types of input, such as task models.

The final requirement applies to the PUC's design. The PUC makes consistency decisions based on the previous interfaces that users have experienced, so it is possible for the PUC to generate consistently poor interfaces if users start with poor interfaces. In order to address this, the PUC must handle the following requirement:

R.7 Users must be able to choose to which appliance consistency is ensured

This requirement affects the PUC's consistent interface generation at a fundamental level, because its data structures must include information about each of the possible consistency choices and it must have some means to keep track of the current choice. To support this, I developed the mapping graph structure, which is used by all of the PUC's consistency algorithms and discussed in the next section. Although the architecture supports the ability for users to choose consistency, an interactive interface to support user choice is still the subject of future work.

6.4 Understanding and Finding Similarities between Specifications

In order to support consistency within the PUC, the interface generator must understand how a new specification is similar to previous specifications for which interfaces have already been generated.

6.4.1 Knowledge Base

The knowledge base is an important piece of the PUC's architecture for supporting consistency. It stores previously generated specifications, mappings between specifications, and information about the interface designs built from those specifications. The most important elements of the knowledge base are the mappings between functions on different appliances. These mappings can be automatically generated by rules or manually specified by the user, and in the future could made available for download over the Internet.

A mapping defines a one-to-one relationship between similar functions in two specifications. This approach simplifies the authoring of mappings, because the author of a mapping needs to only examine differences between two specifications without considering how these differences relate to other specifications. The processing of mappings is also simplified, because the consistency algorithms currently only need to understand the differences between two specifications: the specification being generated and the specification that is serving as the basis for consistency. The drawback of this approach is that mappings are required between all pairs of appliances with similar functions, which could be limiting if this technology becomes widespread and the similarities between many appliances must be included in the knowledge base.

The PUC supports six different types of mappings, each of which was identified from the specifications written for the authoring studies. Each mapping type is described in Table 6.1. Most of these types are generally applicable for creating mappings with any type of functional specification language, though one type is based on the PUC's unique Smart Templates construct (see Chapter 7).

In order to specify and store the mappings, an XML-based language was developed for specifying mappings, the schema for which can be found in Appendix C.2. The basic elements of the mapping language correspond to the six mapping types described in Table 6.1, and each mapping type has child elements that allow its parameters to be specified.

I chose not to use an ontological approach for the knowledge base for two reasons. First, current ontology languages, such as RDF [W3C 2006] and OWL [Herman 2006], seem to be most useful for understanding the relationships between concepts that can be described with nouns and adjectives. For example, an ontology might help a computer understand that a "bear" is an "animal" with "fur," and animals with fur are "mammals." The concepts necessary to achieve consistency are not nouns however but actions (verbs), and actions are much more complex because they are made up of multiple steps which may be constrained in a particular order. Differences between similar actions may occur in one or more steps or in the ordering constraints between steps. The more general ontology languages, such as OWL, may be able to represent actions, but there are two drawbacks: specialized processing is required to analyze the ordering constraints between steps and the description of an action requires substantially more overhead than a language designed to described actions, such as ConcurTaskTrees [Paterno 1997].

Second, ontologies often rely on hierarchy to understand how concepts generalize. The "bears are animals" example from above shows a small aspect of how hierarchy allows for generalization. To create a useful action ontology for appliances, each appliance operation would need to be placed into a hierarchy of one or more generalized operations. Determining each of these generalized operations and appropriately labeling them would be a significant and time-consuming challenge, and furthermore would probably not be useful unless it was complete for all appliances. My solution, with its one-to-one mappings, does not require the entire appliance space to be mapped before being useful and can be incrementally added to as more appliances are specified.

6.4.1.1 Mapping Graphs

Mappings between similar functions in multiple specifications are grouped together in a mapping graph. The central purpose of a mapping graph is to help determine which appliance should be used as the basis for consistency for a function. Every mapping belongs to a mapping graph, and there is a mapping graph for each set of similar functions in the knowledge base. For example, the power, media controls, and VCR/TV functions all have separate mapping graphs containing mappings specific to those functions. An example mapping graph for the media controls function is shown in Figure 6.2.

To find the specification to ensure consistency to, the PUC's consistency algorithms start at the node that represents the appliance being generated and traverse the mapping graph to find the node that the user has seen most often. Each node maintains a count of the times it has been used as the basis for consistency. As discussed earlier, it may be impossible to ensure consistency between similar functions if their specifications are too different and this is represented by the edges of the mapping graph. For example, in Figure 6.2 the Panasonic VCR represents play, stop and pause as a state variable while the Cheap VCR uses only commands. It is not possible to convert between these representations, so the mapping between them will have infinite cost. Costs allow mapping graph traversals to ensure that consistency can be maintained between the endpoints of the traversal result. You may wonder why I bother to include infinite cost edges in the mapping graph. When no zero cost edges are available, infinite cost edges may be traversed to ensure at least some consistency by using the labels of a similar function.

Table 6.1. Mapping types for consistency in the PUC system.

Name	Description
`general`	Allows a series of operations on one appliance to be matched with series of operations on another appliance, with support for repetition. The possible operations are invoking a command or changing the value of a state variable.
`state`	Maps two state variables. Particular values of the state may be mapped together, and a shortcut is available to define that the two states have entirely equivalent values.
`node`	Specifies that a node in the group tree from one specification is similar to a node in another specification. A node could represent a group, command, or state variable.
`list`	Specifies that two lists contain the same data.
`group`	Groups multiple mappings together. Groups cannot be nested.
`template`	Maps two Smart Templates (Smart Templates are described in detail in Chapter 7).

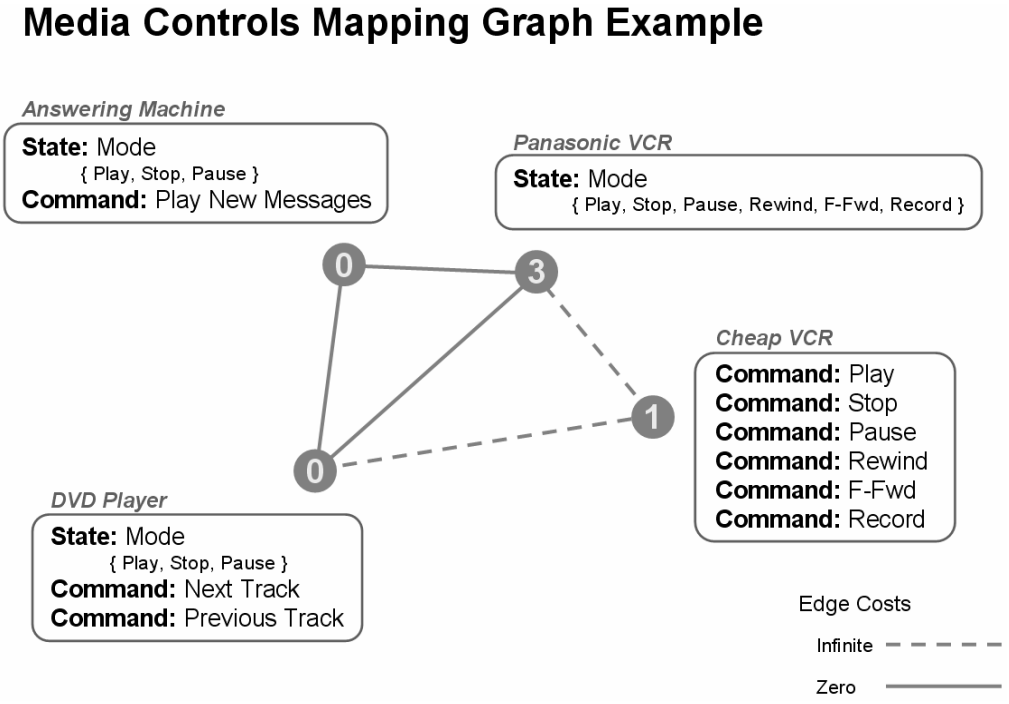


Figure 6.2. An example mapping graph for the media control functions, e.g. play, stop, and pause, on four appliances. The node counts indicate that the Panasonic VCR has been the basis for consistency three times (for itself, the answering machine, and the DVD player) and the Cheap VCR has been the basis for consistency once (just for itself). The answering machine and DVD player were generated to be consistent with the Panasonic VCR, and thus both have counts of zero.

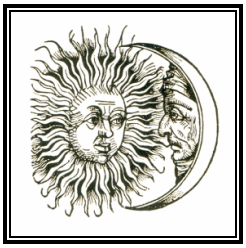
6.4.2 Automatically Finding Mappings

In collaboration with Brandon Rothrock, I have explored automatically extracting mappings from a new specification and previous specifications that the user's controller device has already stored. The challenge of automatic mapping is the lack of substantial semantic information about each function within the specification.

Our work in finding mappings between specifications is similar to previous work in schema and ontology matching from the database community [Shvaiko 2005]. Work in matching has tried numerous approaches, including matching with the help of a thesauras [Madhaven 2001] or the use of machine learning [Doan 2001]. Several matching techniques take advantage of the availability of a database containing many instances of the data specified by the schema. In contrast, instance information is not available for PUC specifications during generation and is not likely to be of much use anyway as most PUC state variables have very few possible values.

We built two separate matching systems. The first is based on our intuition about the PUC data structures, and makes use of names, label dictionaries, and variable types. The second is based on the similarity flooding technique [Melnik 2002] developed for schema matching, which also incorporates organization. The first system performs the best, finding about 60% of the mappings in our VCR test cases with about 20% of the total mappings found being false positives. Currently, neither system is successful enough to integrate into the PUC, however we have found the results of these systems to be useful as a starting place for a human to create mappings between two specifications.

Future work in this area will require investigating other means to improve the matching algorithms, such as incorporating a thesaurus of appliance terminology or leveraging the existing mappings among other specifications. We did not explore the latter approach because the size of our knowledge base is still quite small.



CHAPTER 7

Handling Domain-Specific and Conventional Knowledge[4]

A common problem for automatic interface generators has been that their interface designs do not conform to domain-specific design patterns that users are accustomed to. For example, an automated tool is unlikely to produce a standard telephone keypad layout. This problem is challenging for two reasons: the user interface conventions used by designers must be described, and the interface generators must be able to recognize where to apply the conventions through analysis of the interface specification. Some systems [Wiecha 1990] have dealt with this problem by defining specific rules for each application that apply the appropriate design conventions. Other systems [Kim 1993] rely on human designers to add design conventions to the interfaces after they are automatically generated. Neither of these solutions is acceptable for the PUC system. Defining specific rules for each appliance will not scale, and a PUC device cannot rely on user modifications because its user is not likely to be a trained interface designer. Even if the user was trained, he or she is unlikely to have the

[4] The work in this chapter was originally described in Jeffrey Nichols, Brad A. Myers, and Kevin Litwack. "Improving Automatic Interface Generation with Smart Templates," in Proceedings of Intelligent User Interfaces (IUI). Funchal, Portgual. 2004. pp. 286-288

time or desire to modify each interface after it is generated, especially if the interface was generated when needed to perform a specific task.

I have developed one solution to this problem called *Smart Templates*, which augment the PUC specification language's primitive type information with high-level semantic information. Interface generators are free to interpret the semantics of a Smart Template and, if appropriate, augment the automatically generated interface with the conventions expected by the user. Smart Templates are specially designed to integrate hand-designed user interface fragments that implement the conventions with an otherwise automatically generated interface. Templates are also designed to scale across different appliances without requiring help from the user after generation. Interface generators are not required to understand every template, and templates are designed such that the full functionality of every template is available to the user even if the interface generator does not understand that template.

7.1 Roles

The design, implementation, and use of a Smart Template involve people in four different roles (shown visually in Figure 7.1):

- *Users* own controller devices and make use of the interfaces containing Smart Templates produced by automatic interface generators.
- *Specification Authors* write specifications for appliances. I expect that most authors would work for the appliance manufacturers, though some specifications might be written by third parties. Specification authors learn about available Smart Templates from the Template Registrar, which they may then instantiate in their specifications. Specification authors are not required to use any templates in their specifications, but of course there is a strong benefit to doing so because of the potential increase in generated interface quality.
- *Interface Generator Programmers* build the software that automatically generates user interfaces. These programmers might be employed by the controller device manufacturer or a third party. Programmers learn about available Smart Templates from the Template Registrar and will choose which templates to implement based on the properties and requirements of their target controller device. There is no requirement for an interface generator to support any Smart Template, though some would be highly recommended (such as the date and time templates).

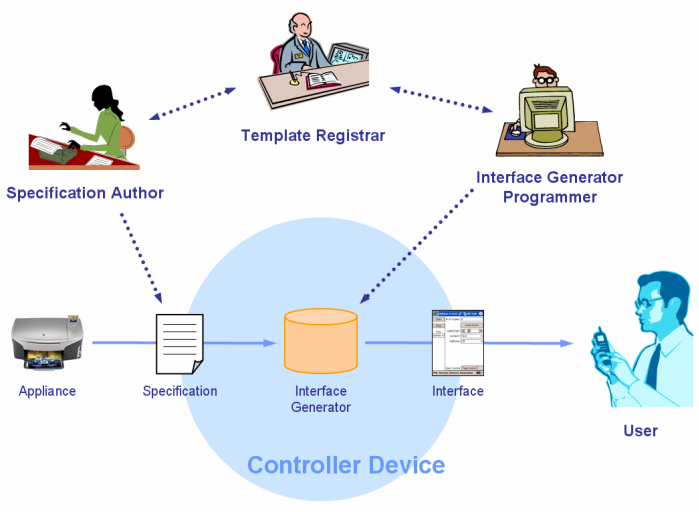


Figure 7.1. A diagram showing the four different roles in the creation, implementation, and use of a Smart Template in the context of the interface generation process.

- The *Template Registrar* is a global entity that manages the list of all Smart Templates and defines the required and optional parameters of those templates. New templates may be developed by the registrar or proposed by other entities.

7.2 Design and Use

There are three steps in the development and use of a Smart Template:

1. The template registrar defines the new template.
2. A specification author instantiates the template one or more times in an appliance specification.
3. An interface generator, which is already aware of the template, uses special hand-designed interface code to appropriately render the template as used in the specification.

Smart Templates must be defined in advance by the template registrar to ensure there is some agreement between the specifications authors, who instantiate templates in specifications, and the interface generator programmers, who write the code that renders the templates. Agreement does not need to absolute however, and flexibility is built-in to the design to allow some benefits to the user even when both the specification authors and generator programmers to not adhere fully to the Smart Template standards. The user will

always be able to access all appliance functionality, and in the worst case the user will be presented with an interface rendered without any special Smart Template rules.

The registrar defines a new Smart Template by giving the template a name and defining a set of specification restrictions for the template. A specification author instantiates a template by adding an `is-a` attribute to a group, variable, or command with the name of the template and the conforming to the template's restrictions within that section of the specification (see Figure 7.2). When the interface generator encounters a section of a specification referencing a template that it knows about, it can invoke special code written by the generator programmer to appropriately render the template. If the generator encounters a template that it does not know about, it will use its normal generation rules to render the template contents. This is possible because every Smart Template is defined using the primitive elements of the specification language. For example, Figure 7.3a shows an instance of a `media-controls` Smart Template rendered by a generator with no knowledge of that template and Figure 7.3b

```
<group name="Controls" is-a="media-controls">
  <labels><label>Play Controls</label></labels>

  <state name="Mode">
    <type>
      <enumerated>
        <item-count>3</item-count>
      </enumerated>
      <value-labels>
        <map index="1">
          <labels><label>Stop</label></labels>
        </map>

        <map index="2">
          <labels><label>Play</label></labels>
        </map>

        <map index="3">
          <labels><label>Pause</label></labels>
        </map>
      </value-labels>
    </type>
  </state>

  <command name="PreviousTrack">
    <labels> <label>Prev</label> </labels>
  </state>

  <command name="NextTrack">
    <labels> <label>Next</label> </labels>
  </state>

</group>
```

a.

```
<group name="Counter" is-a="time-duration">
  <labels> <label>Counter</label> </labels>

  <state name="Hours">
    <type>
      <integer/>
    </type>
  </state>

  <state name="Minutes">
    <type>
      <integer>
        <min>0</min> <max>59</max>
      </integer>
    </type>
  </state>
</group>
```

b.

```
<state name="SongLength" is-a="time-duration">
  <type>
    <string/>
  </type>

  <labels> <label>Length</label> </labels>
</state>
```

c.

Figure 7.2. Three specification snippets showing instantiations different Smart Templates. a) The instantiation of the `media-controls` template for the play controls on Windows Media Player. b) The instantiation of the `time-duration` template for the counter function on the Sony DV Camcorder. c) The instantiation of the `time-duration` template for the song length function on Windows Media Player.

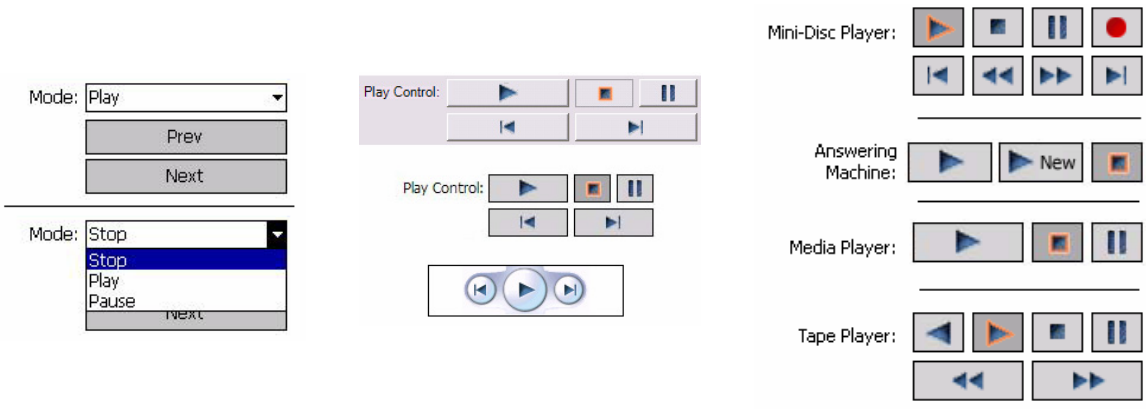
shows the same instance rendered by generators on several different platforms that did know about the template.

The restrictions on the specification allow Smart Templates to be parameterized, which allows them to cover both the common and unique functions of an appliance. Parameters are specified in terms of the primitive elements of the specification language and consist of a list of the state variables and commands that the template may contain along with definitions of the names, types, values, and other properties that these elements must have. Some of the elements may be optional to support functions that would not be used in all instantiations of a template. For example, two representations of play controls are allowed by the `media-controls` template: a single state with an enumerated type or a set of commands. If a single state is used, then each item of the enumeration must be labeled. Some labels must be used, such as Play and Stop, and others are optional, such as Record. If multiple commands are used, then each command must represent a function such as Play and Stop. Some functions must be represented by a command and others are optional. This template also allows three commands for functions that are commonly included in the same group as the play controls, including the previous and next track functions for CD and MP3 players, and the play new function for answering machines. Allowing many combinations of states and commands in a template definition allows a single Smart Template to be applied across multiple kinds of appliances (see Figure 7.3c).

The challenge for the template registrar is to find the different combinations of states and commands that an appliance implementer is likely to use. This makes it easier for appliance specification writers to use the templates, because there is no need to modify the appliance's internal data representation in order to interface with the controller infrastructure. For example, some appliances may not be able to export their playback state, and thus would want to use the option of specify each playback function with a command. Another example is for the `time-duration` template. Windows Media Player makes the duration of a song available as a single integer while our Sony DV Camcorder makes the playback counter available as a string (see Figure 7.2b and c for specifications). Both of these representations can be accommodated by the `time-duration` Smart Template, which allows the PUC to be implemented more cleanly with these appliances because no translation is needed to suit the requirements of the PUC. Note that the regular specification language is used to specify how the smart template is represented. No new techniques need to be learned by specification authors to use and customize smart templates.

Each interface generator implements special generation rules for each template. This allows each template to be rendered appropriately for its controller device using bits of hand-designed interfaces specifically created the template and the controller device. Sometimes these hand-designed interfaces include platform-specific controls that are consistent with other user interfaces on the same device. In the case of our `time-duration` Smart Template implementation, each platform has a different standard control for manipulating time that our interface generators use. Unfortunately, none of our platforms have built-in controls for media playback, so our `media-controls` Smart Template use renderings that I hand-designed to be appropriate for each of the different platforms. The Smartphone `media-controls` implementation does mimic the interface used by the Smartphone version of Windows Media Player however, and thus is consistent with another application on that device (see Figure 7.3b).

Smart Templates are also able to intelligently choose a rendering based upon the contents of the template. For example, each implementation of the `time-duration` template only renders the time units that are meaningful, and each implementation of our `media-controls` Smart Template renders buttons for only the functions that are available. The `media-controls` implementations on the PocketPC and desktop extend this by intelligently laying



a. b. c.

Figure 7.3. Renders of the `media-controls` Smart Template on different platforms and for different appliances. a) Media controls rendered with Smart Templates disabled for a Windows Media Player interface on the PocketPC platform. b) Media controls rendered for the same interface with Smart Templates enabled on each of our three platforms. At the top is the desktop, the middle is PocketPC, and the bottom shows Smartphone. The Smartphone control maintains consistency for the user by copying the layout for the Smartphone version of Windows Media Player, the only media player application we could find on that platform. This interface overloads pause and stop onto the play button. c) Different configurations of media playback controls automatically generated for several different appliances.

out the buttons on one or more lines depending on space, enlarging buttons of greater importance such as Play, and using a grid to create aesthetically pleasing arrangements (see Figure 7.3c).

The PUC interface generators can also robustly deal with content within a Smart Template that does not conform to the restrictions specified by the template registrar. When content is encountered that the interface generator does not understand, this content is passed back for rendering by the normal interface generation process. This approach is also used in other cases when the Smart Template rules understand the content but are not able to produce a

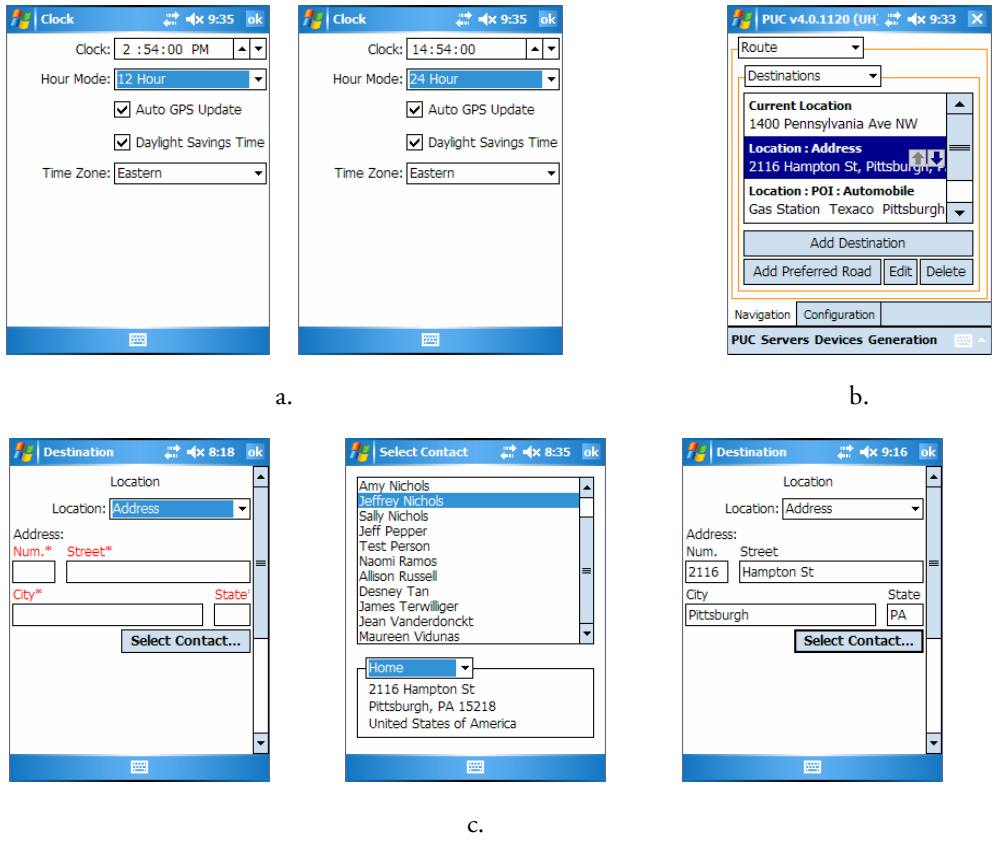


Figure 7.4. Screenshots of Smart Templates rendered as part of the GMC Denali navigation system on the PocketPC platform. a) Demonstrates the time-absolute Smart Template used for a clock function. The 12- and 24-hour option of the template changes the way time is rendered throughout the interface, as can be seen in the clock at the top of these screenshots. b) Demonstrates the list-commands templates integrated with one of the PUC's list controls. Several commands for adding and deleting items are located underneath the list control, along with the edit button that is part of the list control. Two move commands have also been integrated with the list control as the arrow buttons located on top of the selected list item. c) Demonstrates the address template's capability of integrating with the PocketPC's built-in Outlook contact database. The leftmost screen shows the interface before the user has pressed the Select Contact… button. Pressing this button shows the middle screen, which allows the user to select a contact from their database. Pressing OK from this dialog causes the selected information to be filled appropriately into the fields of the template (rightmost screen).

rendering better than the normal generator. For example, the `time-absolute` template has support for optional variables that define how time should be rendered, such an enumeration specifying whether to use a 12-hour or 24-hour time format (see Figure 7.4a). When these variables are encountered, the PUC PocketPC generator saves references to them but passes them back to interface generator for normal rendering. The references allow the template to know when the user the changes the variables, so that it can take these changes into account when rendering absolute time values elsewhere in the interface.

In other situations, an interface generator may explicitly decide not to implement an entire Smart Template because the interface generated by the normal interface generator is already sufficient. For example, a speech interface generator might not implement the `media-controls` template because the best interaction is speaking words like "Play" and "Stop" and this is the interface that would already be produced.

The code for a Smart Template on a controller device can also access special features or data that are specific to that controller device. For example, the `address` and `phone-number` Smart Templates in the PUC PocketPC interface generator were implemented to take advantage of the built-in Outlook contacts database that is present on every PocketPC. Whenever an appliance requires the entry of an address or phone number, the template provides a special button that opens a dialog box that allows users to pick one of their contacts. When they return to the PUC interface, the appropriate information from the database is automatically filled into the appliance's fields. This integration is particularly useful in the navigation system interface, where it allows users to quickly specify a destination to navigate to from their contacts (see Figure 7.4c). Another potential controller-specific implementation would be to take advantage of any special physical buttons that a controller device possesses. For example, if a controller had two special buttons for volume up and down, then the volume Smart Template could automatically allow those buttons to control the volume of the current appliance, if appropriate. The current PocketPC implementations of the `media-controls` template allow the left and right directional buttons to be used for next track and previous track, if those functions are available.

7.2.1 Implementing a Smart Template for an Interface Generator

Each implementation of a Smart Template must accommodate all of the required and optional parameters of that template. For many templates, this puts a significant burden on the implementer to handle many different possible inputs and reflect all of these differences in

the user interface. The implementation challenge becomes even greater for *merged templates*, which are Smart Templates that contain one or more other Smart Templates. For example, the `date-time` template is a merged template because it must contain one instance of both the `date` and `time-absolute` templates. Merged templates are more complex to implement because the implementer must be able to handle the different input combinations for all of the constituent templates.

The implementation complexity of a Smart Template can be reduced by separating the implementation task into two layers:

- The data-translation layer converts data back and forth between the appliance format defined by the specification and the template data structure. The template data structure contains all the possible data for a given template in one format along with flags that define the optional parameters used in an instance of the template. If the data in the template data structure changes, this layer converts the data back into the appliance's format and sends the changes to the appliance.
- The interaction layer creates a user interface based on the common data structure and allows the user to manipulate the template's data. When changes are made by the user, the template data structure is updated and the data translation layer is notified.

These layers separate the instantiation of a template in a specification from the generation of its interface. The template data structure is the sole point of communication between the layers. Using this approach is particularly beneficial in several situations:

- Templates that represent a single data type, such as date or time, may have many different formats in the specification. In this situation, implementing the data conversion is complex but the interface generation is basically the same no matter what the underlying data format is. Separating the data conversion simplifies the interface generation because the generation only needs deal with a single data format.
- Merged templates are easier to implement, because the data conversion layers from the constituent templates can be re-used to create a merged template data structure. This reduces the cost of implementing a merged template to the sub-task of creating interface generation algorithms.
- If code can be shared across platforms, as it can in my implementations for the PocketPC and Smartphone, then the data conversion layers can also be shared. New

interface generation layers will be needed to address differences in the platforms, but again the cost of implementing a template may be reduced substantially.

The layering approach does not remove all of the complexity from the implementation of a Smart Template however. Some differences in the specification of a template must be reflected in the generation for that template, which requires the complexity of the template to be addressed in both layers.

7.3 Smart Template Library

The PUC research team and I have implemented 15 different Smart Templates, as shown in Table 7.1. These templates collectively illustrate all of the features discussed in previous sections. A full description of all Smart Templates is available on the web at:

`http://www.pebbles.hcii.cmu.edu/puc/highlevel-types.html`

As we have built Smart Templates, we have found two uses for templates that improve the PUC interfaces but differ somewhat from our initial intentions:

- The PUC supports state variables with a binary type. These variables could contain images, sounds, or other data that cannot be easily communicated through the PUC's text-based communication protocol. Smart Templates were chosen to handle this binary information for several reasons. First, it did not seem appropriate for all interface generators to handle all kinds of binary data. Smart Templates were attractive then because they are optional by design, although in this case the lack of a Smart Template for handling a particular binary data type means that data cannot be presented in the interface. Second, the controls needed for displaying any binary data would need to be custom built for the particular platform, which Smart Templates are already capable of adding to the generated interface. Finally, binary-typed data requires extra communication to negotiate the particular sub-types of data (such as image formats) that a particular platform supports. This extra communication is likely to differ based on the type of binary data being managed, requiring different implementations for each type. The data translation layer of the Smart Templates for different binary data types is an ideal place for this code.
- List operations can also be handled by Smart Templates, allowing for special integration with the controls for handling list data that would not otherwise be possible. A challenge the PUC faced for handling lists was how to represent list operations.

There are several different kinds of list operations, such as add, delete, move, insert, etc., with so many different variations that did not seem practical to include as first-class elements in the specification language, e.g. add-first, add-last, add-after, add-before, and so on. Ultimately, I decided that most list operations could be specified as separate commands outside of the list they operated on. However, when building the generation rules for list interfaces, I found that list interfaces were easier to generate well when the interface generator could identify the commands that operated upon it. Furthermore, commands that added items to a list could not automatically display a dialog box to enable editing of the newly added item. The solution I chose was to create the `list-commands` Smart Template, which is a merged template that may contain multiple other templates. These constituent templates include `list-add`, `list-remove`, `list-clear`, `list-move-before` and several others. The list templates allow all operations to be grouped appropriately next to the list control, add operations to perform the correct dialog box opening behavior, and for move commands to be handled directly inside the list control rather than being displayed as separate commands away from the list. It is important to note that while the list templates allow for better list interfaces, it is possible for PUC interface generators to produce an adequate list interface with no knowledge of these templates.

7.4 Discussion

The unique feature of Smart Templates is their ability to integrate small snippets of hand-designed user interfaces into interfaces that are otherwise automatically designed. This allows the design conventions that users expect to be included in the appliance user interfaces, while still allowing flexibility for those interfaces to be generated on multiple platforms with features like consistency and aggregation. Smart Templates also provide some consistency across interfaces for different appliances, because they ensure that the same controls are used for similar kinds of functions (see Figure 7.3c).

Smart Templates do have some limitations however. The complexity and size of a template is limited by the cost of implementing that template on an interface generator. While the template registrar could standardize a template for a very large set of functionality, such as entire appliance like a VCR, the cost of hand-designing an interface flexible enough to accommodate the wide variety of functionality that a VCR may have is prohibitive. As the functionality of a template becomes a large percentage of an entire appliance, the benefits

possible from other automatic generation techniques, like consistency, will also be lost because the majority of the user interface is no longer being generated automatically.

It is important to note that the standardization needed for Smart Templates is quite different from that being pursued by current industry projects like UPnP [UPnP 2005]. UPnP standardizes the specification for each appliance. This means that all the manufacturers for printers, for example, must sit down in a room and agree on the set of functionality that all printers will have. User interfaces for printers are based on this standard. After a standard has been agreed upon, manufacturers may still add custom functions, but these functions will not appear in most UPnP printer user interfaces. Smart Templates, however, standardize functionality at a much finer-grain level. It should be much easier for manufacturers to agree on the features of media controls for example, as compared to a printer or VCR. Smart Templates also do not affect manufacturers' ability to innovate. New functions, even if they extend the functionality of an existing Smart Template, can always be added to an appliance and rendered by the normal interface generator.

It is not unusual for consumer electronics manufacturers to produce both appliances and high-end remote controls that can be used with those appliances. In this situation, specification authors and interface generator programmers might work for the same company and it is reasonable to believe that manufacturers might develop their own in-house Smart Templates that improve their appliances interfaces when used with their remote controls. However, since these proprietary templates would not be understood by other interface generators, the interfaces generated for that appliance could be sub-par as compared to competing appliances. This could be particularly bad if the proprietary template replaced a common template, such a `media-controls`, preventing even basic conventions from being applied in the appliance interface.

Table 7.1. Description of all implemented Smart Templates in the PUC system.

Template Name	Description
`address`	Represents all aspects of a street address, including street name, number, city, state and zip code. Primarily used by the navigation system, but also supports entry of zip codes on appliances that need location information to configure program guides, etc. On the PocketPC platform this template also integrates with the built-in Outlook database to allow users to automatically insert contact information into the fields of a PUC interface.
`channel`	Represents the channel on a television, station on a radio, or the tuning parameter of some other appliance. This template is needed to support the automatic manipulation of channels by the multi-appliance interfaces discussed in Chapter 9, but also offers better support for channel manipulation than is currently afforded by the automatic interface generation. For example, up and down buttons can be provided for manipulating channels along with the arbitrary numeric entry that the PUC normally supports.
`Date`	Represents calendar dates. Many date formats are supported to match the internal implementation of a particular appliance: including integers, strings, and multi-variable formats. This template also allows for use of the special date manipulation control from each controller device's platform.
`date-time`	Merged template of date and time-absolute. The individual templates are used to transform the specification format into a common format that can then be used with a special control, if available, that allows the simultaneous manipulation of date and time.
`dimmer`	A specialized template that represents a dimmer control for a lighting source. This template supports an arbitrary scale with optional support for direct on and off functions. This template is primarily used by the Lutron RadioRA lighting control.
`four-way-dpad`	Represents a four-way directional control pad, as might be found on a DVD player or navigation system. This template also includes optional support for an "enter" button that appears in the middle of the four directional buttons. In general, I hope that specification authors can avoid use of this control for navigation on-screen menus that could be replicated on the PUC. There are situations however, such as for navigation of map content that cannot be displayed on the PUC, where this control may be needed.
`image`	Represents an image of any type. Currently, this template is used in the interfaces for the PowerPoint and photo browser app. Images are transmitted to the PUC through state variables with binary types. This template negotiates with the appliance to get an image with a size and type compatible with the controller device.
`image-list`	Handles a list of objects which must include one object that handles the `image` template. The typical representation of this template in a user interface is two overlapping views: a list of thumbnails of all the images and a single record view that shows a larger image along with any other fields in the list. This template is used primarily in the interface for the photo browser appliance.
`list-commands`	Represents the common operations that might be supported by a list, such as add, remove, clear, and move. The use of a template allows these operations to be better integrated with the list controls, particularly for moving items.

`media-controls`	Represents the functions for controlling the playback of a variety of media, including common functions such as play, stop, and pause and less common functions such as play new for answering machines, reverse play for audio tapes, and multiple playback speeds for VCRs and DVD players.
`phone-number`	Represents a phone number and/or the controls for entering a phone number, such as the standard phone keypad. Like the address template, on the PocketPC this template can integrate with the built-in Outlook database to allow users to automatically insert numbers into the fields of the PUC interface.
`status-icons`	The status-icons template is a grouping template in which one or more special templates representing a variety of status indicators, typically represented as icons, are contained. Currently ten different indicators are supported by the PUC, including ones for specifying whether a tape is in the appliance, whether the device is busy, etc (see Figure 7.5 for a complete list).
`time-absolute`	Represents an absolute time of day (as opposed to a duration of time). This template is commonly used for the clock function of an appliance but also has many other uses. The date template can also handle other related preference variables, such as whether the user prefers a 12- or 24-hour time and whether daylight savings time is currently in effect. Some of these preferences will affect the representation of other time-absolute templates on the same appliance. Many different representations for time are supported, including integer, string, and multi-variable representations.
`time-duration`	Represents a duration of time, such as would be required for a stopwatch appliance. As for the time-absolute template, many different representations of a time duration are supported, including interface, string, and multi-variable representations.
`zoom-controls`	Represents controls for zooming in an appliance interface, typically with the standard looking glass icons. This template is intended primarily for appliances with content that cannot be displayed on the PUC.

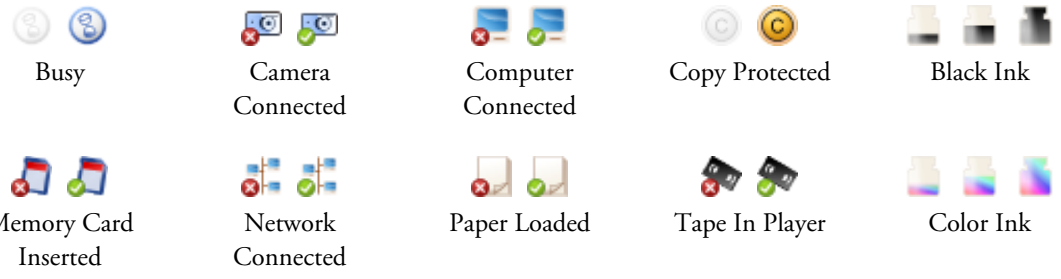
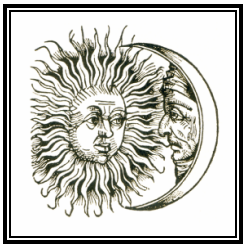


Figure 7.5. Icons currently supported by the PUC status icons Smart Template.



CHAPTER 8

Interface Generation[5]

This chapter discusses the interface generation algorithms used by the PUC system. All the interface generators use a rule-based approach that operates in two stages. In the first stage, an appliance specification is converted to an abstract user interface with no platform specific details. The second stage converts the abstract user interface into a platform-specific concrete user interface. The most important contributions of the interface generators are the use of mutual exclusions to determine the structure for the abstract interface and algorithms that manipulate new interfaces to that they are consistent with previous interfaces that the user has seen.

[5] This work in this chapter was originally in described in three different papers: Jeffrey Nichols, Brad A. Myers, Michael Higgins, Joseph Hughes, Thomas K. Harris, Roni Rosenfeld, Mathilde Pignol. "Generating Remote Control Interfaces for Complex Appliances," in *Proceedings of the 15th annual ACM symposium on User Interface Software and Technology* (UIST). Paris, France. October 27-30, 2002. pp. 161-170, Jeffrey Nichols, Brandon Rothrock, and Brad A. Myers. "UNIFORM: Automatically Generating Consistent Remote Control Interfaces," in *Proceedings of the Conference on Human Factors in Computing Systems* (CHI). Montreal, Quebec, Canada. April 22-27, 2006. pp. 611-620, and Jeffrey Nichols and Brad A. Myers. "Controlling Home and Office Appliances Using Smartphones," in *IEEE Pervasive Computing*. July-September 2006. pp. 60-67.

8.1 Generation Platforms

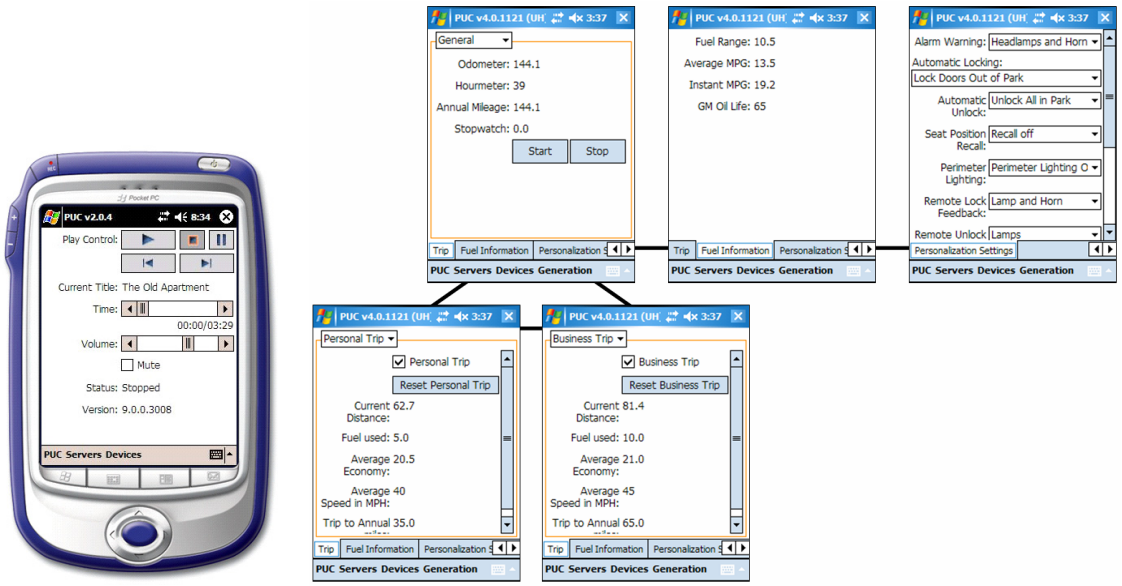
Interface generators have been implemented for graphical interfaces on three different platforms, PocketPC, Smartphone, and desktop, and another generator was built for speech using Universal Speech Interface (USI) [Rosenfeld 2001] techniques. All of the graphical generators were built using the .NET Compact Framework, which allows code to be shared across platforms for common functions such as parsing specifications and communicating with appliances. Interface generation rules are also shared: many between the PocketPC and desktop and only a few between the PocketPC and Smartphone.

The speech interface generator was produced as a Masters project for a student in Carnegie Mellon University's Language Technologies Institute [Harris 2004]. This chapter will briefly summarize how this interface generator works and describe the interfaces it produces, but this generator is not discussed in detail because I was only peripherally involved in its construction.

8.1.1 PocketPC and Desktop

The PocketPC interface generator is the PUC's primary development platform and most features were implemented first on PocketPC before being migrated to the other platforms. 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 240x320 or 480x640 screen with touch-sensitivity. A 4-way directional pad is also available along with four physical application buttons (see Figure 8.1a). Our current interface generator focuses on the touchscreen and only the `media-controls` Smart Template makes use of the physical buttons (see Chapter 7).

The PocketPC interface generator produces interfaces that appear much like other PocketPC applications, so that PocketPC owners can leverage their existing knowledge to use the interfaces. The generator attempts to avoid scrollbars whenever possible, which causes many interfaces to include overlapping panels, manipulated using tabs or another organizing control. Dialog boxes are also created in certain situations. These techniques allow all of the complex functionality of an appliance to be accessible through one user interface. The PocketPC has menus at the bottom of its screen, but these are reserved for controlling the interface generator itself rather than for controlling appliances. This decision was influended by the preliminary user studies, which showed that users did not typically look for appliance



a. b.

Figure 8.1. a) An interface for Windows Media Player generated on a PocketPC. b) The full interface generated for the GMC Yukon Denali driver information console.

functionality in the menus (see Chapter 3). Figure 8.1 shows two example interfaces generated for Windows Media Player and the driver information console (DIC) of a GMC Yukon Denali.

The input and output technologies used on the PocketPC, a touch-sensitive medium-sized screen, are similar to those of the desktop, and a result the generation rules used for the PocketPC are also used for the desktop. The main difference between the desktop and PocketPC generators is the choice of controls, which is slightly better on the desktop because more controls are available than on the PocketPC. For the remainder of this chapter, whenever I discuss rules for generating PocketPC interfaces it can be assumed that these same rules are also used for generating desktop interfaces unless otherwise noted.

8.1.2 Smartphone

The Smartphone interface generator uses Microsoft's Windows CE-based Smartphone platform. This platform is also a set of hardware requirements and a Windows CE operating system. Unlike the PocketPC, the required screen is smaller (220x176) and it does not have any touch-sensitivity. Instead, interaction takes place through a 4-way directional pad, a nor-

mal phone keypad, home and back buttons, and two soft buttons with labels that are shown on the Smartphone's screen (see the left side of Figure 8.2).

The Smartphone interface generator creates interfaces that follow Microsoft's Smartphone user interface guidelines. As for the PocketPC, I 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. The 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 8.2 shows a Smartphone and the full generated interface for the GMC Yukon Denali DIC. The interface generator tries to keep as much of the interface in the list format as possible, but sometimes a variable cannot be manipulated within the constraints of the list. In this case, an editing pane is created that contains the appropriate controls. The 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 the locations of commonly used functions.

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. This button is currently

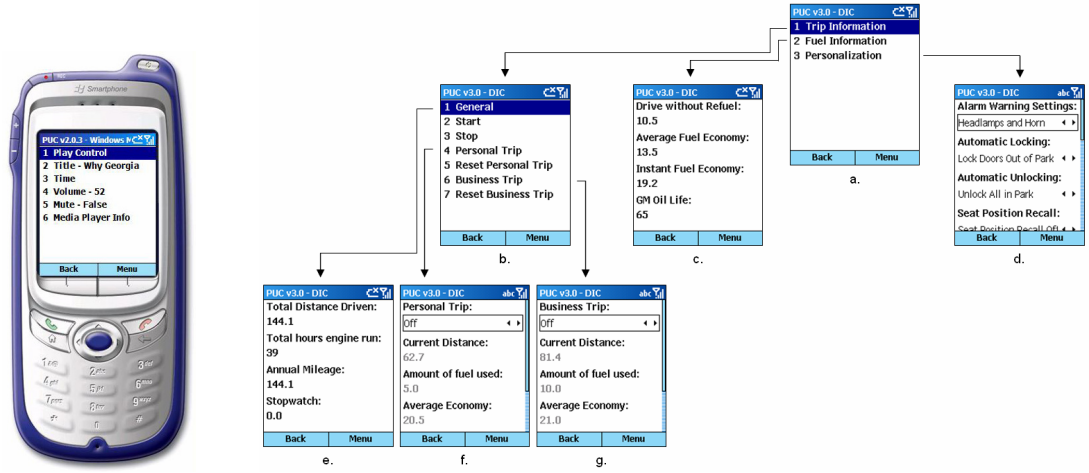


Figure 8.2. A Smartphone displaying a PUC interface for Windows Media Player and the automatically generated interface for the Driver Information Center in a 2003 GMC Yukon Denali SUV. The user navigates through list panes (a,b) to get to summary (c,e) and editing panes (d,f,g).

assigned to always perform the back function, though I have experimented with other approaches.

8.1.3 Speech

The speech interface generator creates an interface from a PUC specification, using USI interaction techniques [Shriver 2001]. The goal of these universal speech techniques is to balance the flexibility of natural language interfaces, which are difficult to implement and typically do not convey their limitations to users, with the structure of specialized hierarchical menu-based speech interfaces. The structure is given by a set of common interaction primitives that are designed to be learnable in five minutes but applicable to all of the interfaces that a user encounters. These interaction primitives allow users to navigate, manipulate, and understand any interface built with the USI techniques. Studies have been conducted to evaluate users' acceptance of these interfaces and the transference of USI skills from one interface to another. These studies show users prefer USI-based interfaces to natural language interfaces though work is still needed to help users learn and remember the interaction primitives [Tomko 2004].

The speech interface generator differs from the graphical interface generators in that it connects to multiple appliances (if multiple appliances can be found on the network), requests their specifications, and then automatically generates an interface that allows control of all of the appliances collectively. 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 [Ward 1990], which is used by the USI library to parse user utterances. A grammar is generated for each appliance 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 appliance-specific grammars, together with an appliance-independent USI grammar, are compiled into a single combined grammar representing everything that the speech interface will understand. Note that this is different from the interface aggregation for multiple connected appliances discussed in Chapter 9, because the combined speech interface is organized by appliance with no re-organization for functions from different appliances that would be used together. The speech interface generator has

been implemented and tested with the Audiophase shelf stereo, the Sony camcorder and multiple X10 devices. Figure 8.3 shows an example of a user interacting with this system.

The speech interface translates the group tree from each appliance specification into a USI-interaction tree. This tree is a more concrete representation of what interactions can take place between the system and the user, which 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.

User	control system
System	control system (system voice)
User	*Options*
System	control x10, control audiophase
User	control x10
System	control x10 (x10 voice)
User	*options*
System	global off, all lights on, all lights off "..."
User	control audiophase
System	control audiophase (stereo voice)
User	*Options*
System	stereo power, x-bass, volume "…"
User	stereo power
System	<stereo is turned on>
User	*Options*
System	stereo power, x-bass, volume "…"
User	*More*
System	mode, tuner, CD
User	stereo power *is, options*
System	on, off
User	tuner *is, options*
System	tuner radio band, tuner seek forward, tuner seek reverse
User	Tuner
System	Tuner
User	*what is* radio station?
System	radio station is 106.7
User	radio station *is* 98.5
System	<changes station to 98.5>

Figure 8.3. An example interaction using the speech interface to control a shelf stereo and X10 lighting.

There were a few unique challenges in 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"). This might be addressed by adding support for "verb" labels in the specification language that specify command words that could be used to manipulate a state variable from a speech interface. 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. The specification language does not require groups to have labels, making appropriate names hard or impossible to find. It may be necessary to require labels on each group for speech interfaces, but more work is needed to determine if this is necessary. Using a speech interface also has benefits. For example, choosing from a long list of names is easy with speech, yet can be challenging for visual interfaces.

8.2 General Concepts

An overview of the PUC generation process for graphical interfaces is shown in Figure 8.4. The process is made up of six consecutive phases that fall into two categories. Each of the generation phases creates a new interface data structure based on the previous structure, taking the attributes of the appliance and the controller device into account. Each of the consistency phases modify the current interface data structure to match elements of previous interfaces that the user has previously seen. These consistency phases make use of the knowledge base to find matching elements and choose the appropriate modification.

There are three types of interface data structures used in the generation process. The first is the specification structure, created by parsing a specification document written in XML. The second is the abstract user interface, which contains more complete knowledge about how the interface should be organized and an abstract choice of controls for each of the appliance's functions. The third is the concrete user interface, which varies depending on the target platform for interface generation. Because the first two structures are common across all generation platforms, all of the rules for creating the abstract user interface and the consistency rules that operate on the abstract user interface can be shared among the different platforms.

Four of the six phases are designed to produce consistent user interfaces. The mapping phase analyses the new appliance specification to automatically find similarties to previous specifications the users have seen. This is the only phase that adds new information to the knowledge base. The functional and abstract structural phases modify the abstract user interface to ensure that functions appearing in previous interfaces are manipulated the same way and appear in the same location in the new interface. The concrete phase is used to clean up visual consistency problems created by the interface generator's transformation of the abstract user interface to the concrete interface, such as changing the orientation of panels or adding additional organization. I discovered in the creation of the consistency algorithms that most of the work needed to ensure consistency can take place in the third and fourth phases, which operate on the abstract user interface. This means that most of the consistency rules can be shared across platforms without modification.

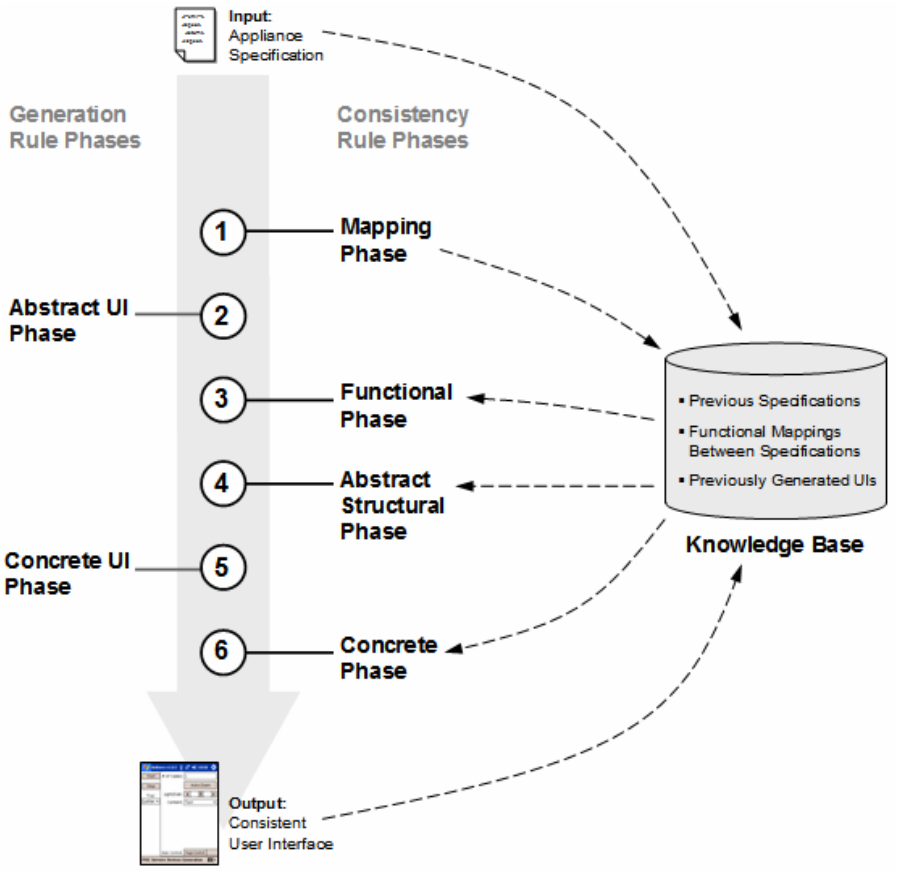


Figure 8.4. PUC interface generation process diagram

8.3 Generating the Abstract Interface

The abstract interface is created from the appliance specification. The structure of the interface is taken from the specification's group tree and modified by an analysis of dependency information. Abstract interaction objects (AIOs) are assigned for all state variables and commands and placed within the structure of the abstract interface. The result is a set of abstract controls within a hierarchical tree that can be transformed into concrete interfaces for the various platforms. The two most importants parts of this process, the analysis of dependency information for structure and the assignment of AIOs, are discussed below.

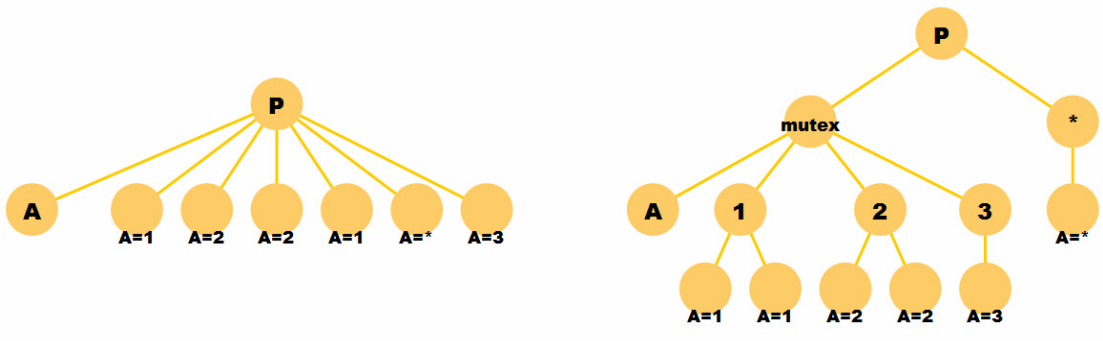
8.3.1 Mutual Exclusions in Dependency Information

The interface generator uses the specification's group tree as the basis for the structure of the abstract interface, but this tree is just one specification author's interpretation of the structure that the appliance should have. Some structure may be missing and the relative importance of different pieces of the existing structure may not be clear. For example, an appliance specification for a shelf stereo might have a group node whose branches represent the main modes of that stereo (CD, Tape, Radio, etc.). The generator needs to discover that this group is of particular importantance for this appliance and create an interface structure that will always make these important branches clear. For most appliance modes, it is best for each mode to have its control placed on a separate overlapping panel to save space and improve usability by hiding controls that are not active. In some cases, the specification author may not have included structure for a mode and the interface generator also needs a means of detecting this and, if possible, inserting the appropriate structure.

Dependency information can be used to address both of these issues with the concept of *mutual exclusion*. If two sets of components are shown to never be active at the same time, then the interface generator can decide that these sets of components should be placed in different groups and that those groups should be marked with as important to carry through in the concrete interface. Another way to solve this problem might be to have the specification authors place a marker on all groups that have mutually exclusive branches. Relying on dependency information instead of a marker makes specification design easier. Instead of finding all cases of mutual exclusion and making the tree fit, the author can give the tree any hierarchy that seems appropriate and the dependency information can help the generator infer structure from it. Structure can even be inferred from a flat tree with only one group.

Unfortunately, the task of determining mutual exclusiveness for an arbitrary set of state variables and commands can be shown to be NP-hard. The difficulty of the problem is reduced by considering mutual exclusion with respect to a single given variable and its location within the specification group tree. The algorithm starts by obtaining a list of all state variables that other commands and states depend upon. In practice, this is a small number of variables for most specifications. The algorithm iterates through this list, starting with the state that is depended upon by the most other variables and commands. Usually the states that are most depended upon represent higher-level modes and the algorithm prefers to create high-level structure earlier in the process. Note that we might also have tried to solve the mutual exclusion problem using a brute force algorithm, but I chose not to use this approach because the generation algorithms needs to run on processing- and memory-constrained devices and N (the number of appliance objects in the specification) could easily be one hundred or greater,

For each state, the algorithm finds its location in the group tree and gets a pointer to the state's parent group. Dependencies on the state are collected from each of the parent group's children and are analyzed to find mutual exclusion. If mutual exclusion is found, the group



a. Before b. After

Figure 8.5. A demonstration of changes made to the tree structure when mutual exclusion is found. The circles represent nodes within the interface structure, which could represent groups, state variables or commands. a) In the *before* tree, the node marked A represents a state variable that can have values from 1-3. The node marked P is the parent group. The A= formulas shown below the remaining groups show the dependencies of each group on the variable A. The A=* node is not dependent at all on A. b) In the *after* tree, a new mutual exclusion group has been added that contains the A state variable and child groups for each set of mutually exclusive groups. The nodes not dependent on A are moved into their own group under the same parent but outside of the mutex group.

tree is re-arranged so that all children in each mutually exclusive set and any children not dependent on the state are each placed in their own group. The parent group is then marked as containing mutual exclusion, which will be taken into account when transforming the abstract user interface into a concrete interface. Figure 8.5 shows an example of how the tree is transformed to take mutual exclusive situations into account.

8.3.2 Choosing Abstract Interaction Objects

Abstract Interaction Objects (AIOs) are abstract representations of the controls, like buttons and sliders, used in the concrete user interface. The idea was developed by Vanderdonckt et al. [Vanderdonckt 1999] to build models of user interfaces that could be ported across different platforms, and I use them for the same reason in the PUC's abstract user interface. The PUC also uses AIOs to delay selection of the concrete control until more properties of the concrete interface are known, such as the available layout space or context of use (within a list, on a panel, etc.).

The interface generator chooses an AIO for each state variable and command using a decision tree [de Baar 1992]. A different tree is designed for each interface generator, because each platform supports a different set of controls that must be accounted for in the tree. Parameters used in the tree include whether the object is a state or a command, the type of the state variable, constraints on the type of the variables, whether the variable is read-only, etc. For example, commands are always represented with a button AIO, and read-only state variables are usually represented by a label AIO.

8.4 Modifying the Abstract Interface for Consistency

After the abstract user interface is created, two phases modify the interface to ensure consistency with previous interfaces. These phases are:

1. **Functional Phase:** each mapping is examined and the abstract user interface may be modified to ensure *functional consistency*.
2. **Abstract Structural Phase:** the organization of the abstract user interface is modified for consistency. This phase helps to ensure *structural consistency*. Following this phase, the PUC transforms the abstract interface into a concrete user interface.

Although the functional phase precedes the abstract structural phase, these two phases could be executed in the opposite order. I chose this order for implementation reasons, because it is

easier to find functions in the abstract user interface before the structural phase moves them around.

All of the rules is these phases must simultaneously balance consistency with previous interfaces and the usability of the new interface. This is particularly important when dealing with functions that are unique to the new interface. Changes for functional consistency can be applied without disrupting the usability on unique functions, but structural changes have broader impact with potentially unintended consequences. The rules for ensuring structural consistency must be carefully applied to ensure that the usability of unique functions is not substantially diminished.

8.4.1 Heuristics for Unique Functions

In order to address the issue of usability for unique functions, three heuristics for the structural consistency rules are followed when making changes that involve unique functions. Heuristics were necessary for two reasons. First, the best approach for dealing with a unique function is to analyze how that function is related to the other functions of the appliance for which we have mappings. If these relationships can be understood, then the unique function can be moved or changed with the functions that are most strongly related to it. Unfortunately, understanding how a unique function relates to other elements of a specification is difficult because the specification language contains very few cues to discover this kind of information. The second reason for using heuristics is the lack of any formal studies that suggest specific methods for balancing the usability of unique functions with consistency. In particular, my central premise is that unique functions should be kept with the consistent interface rather than, for example, be moved to a special panel where all unique functions are collected. Testing this premise and conducting detailed studies of the unique function issue are subjects for future work.

The heuristics are:

- Move unique functions only with their siblings.
- Unique functions in the middle of a group follow their preceding sibling.
- Unique functions at the beginning or end of a group are not reordered.

The central assumption behind these heuristics is that unique functions should be kept near their surrounding elements from the initial specification. This is particularly important for

moving, when functions may end up far away from their initial location in the specification. The re-ordering heuristics only violate this assumption for functions that are located at the beginning of end of a group. This was added because functions at the top are usually important and should be left in place. Functions at the bottom are usually the least important and any reorganization upward makes these items too prominent.

8.4.2 Functional Modifications

The functional phase ensures functional consistency by inspecting each function in the new specification, determining whether there is a previous function with which the new function should be consistent, and then making changes to the abstract interface to ensure consistency.

The previous function to be consistent with is found by traversing the mapping graph (see Chapter 6). If a previous function is found, then a set of functional consistency rules are used to transform the new specification into a form that is consistent with the previous specification. In order to determine the particular rule to apply, the PUC iterates through the rules until it finds a rule that matches the mapping. The PUC will use the first rule that is found, so the rules must be carefully ordered to ensure that those with more specific matching conditions will be tested before those with more generic matching conditions. The PUC currently implements seven functional consistency rules, as shown in Table 8.1. Each rule modifies a portion of the new specification to match the previous specification. For example, the `change-invoke-to-change` rule will convert between a state variable and a state variable with a command that must be invoked before a variable change will take effect. As part of

Table 8.1. The PUC's functional consistency rules.

Name	Description
`State`	Ensures that similar variables use the same label and, if possible, the same control
`invoke-to-invoke`	Ensures that similar commands use the same label
`change-to-repeated-invoke`	Converts between the situation where changing a variable on appliance is the same as repeatedly invoking a command on another
`change-invoke-to-change`	Converts between the situation where one appliance has a state variable and the other has a state variable with a command that must be invoked before a variable change will take affect
`time-end-to-duration`	Converts between the situations where one appliance uses a start time and an end time and another appliance uses a start time and a duration
`Template`	Ensures that smart templates use the same label and control style
`Node`	Ensures that nodes mapped with a node mapping use the same label

this conversion, a command in the new specification must be hidden or a new command must be added. If the command is hidden, the converted state variable will automatically invoke the command when its value is set by the user. If a command is added, then the appliance will not be informed of a variable change until the user invokes the new command in the interface.

8.4.3 Structural Modifications

The goal of the abstract structural phase is to ensure that functions are located in the same place in new interfaces. This phase is divided into two sub-phases: moving and then re-ordering. Moving rules only need to ensure that functions are placed in similar groups, and then the re-ordering rules can ensure that the functions have a consistent ordering within their groups. Both of these sub-phases rely on mappings, such as node, template, and list mappings, that identify similar groups across specifications. Uniform uses this information to rearrange groups so that they have the same structure as a previous specification.

8.4.3.1 Moving

The moving sub-phase traverses the abstract user interface's group tree and searches for mappings.

There are two difficulties that may arise when processing mappings to determine whether a move is necessary. First, the new specification may have mutual exclusive groups and the planned move may cross the boundaries of these groups. The easiest case is when an item that is not dependent on the mutually exclusive state is moved into one of the mutually exclusive groups. No extra work is needed because the moved item will always be available in its new location in the interface. The opposite case, when a dependent item is moved out of the mutually exclusive groups is trickier, because the item may be disabled and it may not be clear why from the item's new location. The most difficult case occurs when an item needs to move from one group to another that is mutually exclusive of the first. In this case, without any extra work, the moved item would never be available because its dependencies would conflict with the organization that is automatically generated by the interface generator. Rules that address this problem are discussed below.

The second difficulty arises when a mapping references more than one variable or command in a specification and these items are not adjacent. I refer to this situation as a split mapping, because the items are split across multiple locations in the specification. Split mappings arise

most often in two situations: when the options of a single function in one specification are represented as two separate functions in another specification or when a single function in one specification is duplicated in a second specification because it is available in more than one context. An example of the first situation can be found in the Mitsubishi DVCR and Samsung DVD-VCR specs, where the front display setting is controlled by one state variable on the Samsung but by two variables on the Mitsubishi that separately control the brightness and content of the front display. An example of the second situation can be found on the HP and Canon all-in-one printers. The HP specification has global buttons for starting the copy,

Table 8.2. Consistency moving rules implemented in the PUC.

Name	Split?	Mutex?	Description
Move Single Item	No	No	Moves adjacent items to a location in the new specification that is similar to the location of similar items in the previous specification.
Move Single Moded Item	No	Yes	Performs the same move as the "Move Single Item" rule, but handles situations when this move will cause items to move across a mutually exclusive group boundary. For example, the Canon all-in-one printer fax setup functions are used in the Setup mode but in fax mode on the HP all-in-one printer and generating a consistent interface between these printers requires moving items into a group with conflicting dependencies. In such cases, this rule removes dependencies on the moved items and automatically handles any needed mode switches in the background when the user uses a moved item.
Move Split Moded Items	Yes	Yes	Handles moves for a mapping in which a single item in one specification is mapped to multiple items in another specification and each of the multiple items is mutually exclusive of the others. For example, this rule handles the situation for the buttons that start copying, faxing, and scanning on the HP and Canon all-in-one printers.
Move One to Many Split Items	Yes	No	Similar to the move split moded items rule, this rule also handles moves for a single item in one specification that is mapped to multiple items in another specification. None of the multiple items may be mutually exclusive of each other for this rule, however.
Bring Together Split Dependents	Yes	Yes	Works for group mappings that contain a state mapping and one or more task mappings. This typically means that one appliance has a set of functionality that is moded whereas at least some of that functionality is available simultaneously in the other specification. If that functionality is split in either spec, this rule handles moving the dependent pieces to the appropriate locations.

fax, or scan functions, depending on the current mode. The Canon has separate buttons for starting copies, starting faxes, and starting scans. Each set of buttons on the Canon is mutually exclusive of the others however because the printer can only be in one of these modes at any given time.

There are five different rules for moving items. One of these rules handles the simplest case of moving items that are adjacent in both specifications and do not cross any mutually exclusive group boundaries. The remaining rules address particular combinations of split and mutually exclusive situations. Table 8.2 shows the different moving rules.

An important feature of the moving sub-phase is its data structure, called the "containment stack." The purpose of the containment stack is to keep track of similar parent groups as the sub-phase traverses through the tree. Two stacks are created, one for the new appliance and another for the previous appliance. Only mappings between these two specifications are included in the containment stack, so any entry in the stack is known to refer to an existing location in both specifications. For example, the containment stack for the clock group in the Mitsubishi DVCR and Samsung DVD-VCR is shown in Figure 8.6a. The clock is located in a different group in these specifications, which is reflected in the containment stack.

All of the moving rules check the containment stacks to see if the top-most group mappings are different. If the mappings are different and other requirements of the moving rule are met, then the mapping's objects are moved to the group that corresponds to the previous specification's top-most group mapping. For example, suppose that we are generating the Samsung DVD-VCR interface to be consistent with the Mitsubishi DVCR. The top-most group mappings are Base.Setup for the DVD-VCR and Base.Status for the DVCR, which are different. These items in each specification are also adjacent and the move will not cause an item to cross a mutual exclusive group boundary, which means that the "Move Single Item" rule will be applied. The moving rule will take the clock group and move it to the DVD-VCR's Base.Status group, which is similar to the group with the same name on the DVCR (see Figure 8.6a). Note that the moving algorithm also chains appropriately. For example, the clock group on the DVD-VCR contains a variable that specifies the channel from which clock information can be extracted. The DVCR has a similar state variable, but is located in the setup group instead of the clock group. Before the PUC moved the clock group, the clock channel variable had the same containment in both specifications, but afterward this is no longer true. When the algorithm is applied to this channel variable, the difference in containment is found because the containment stacks are calculated from the variable's

current location. The algorithm will then move the channel variable back to its consistent location in the setup group (see Figure 8.6b). Note that this movement is visible in the generated interfaces: Figure 8.7b shows the clock group under the Setup tab with the clock set variable, and Figure 8.7c shows the clock group in the Status tab without the clock set variable.

All of the moving rules will also copy any appropriate missing structure from the previous specification into the new specification. For example, suppose that there had been an additional group named "DateTime" between the clock group and the Base.Status group in the Mitsubishi DVCR specification. If this had been the case when generating the Samsung interface to be consistent with the Mitsubishi, the moving rule that moved the Clock group to the Base.Status group would have created a new group named "DateTime" in the Base.Status

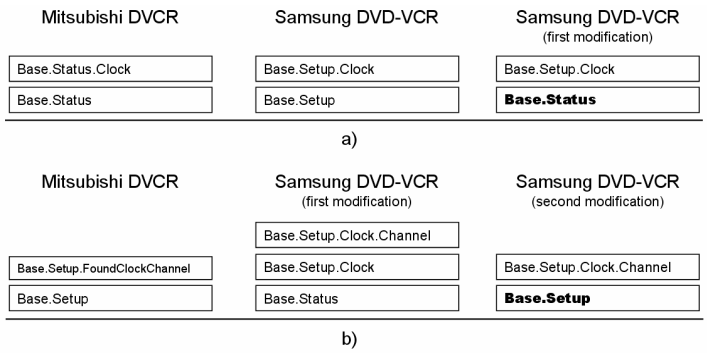
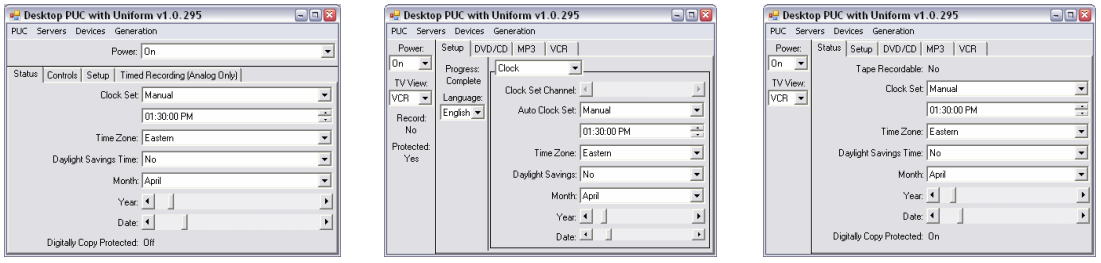


Figure 8.6. Containment stacks for the previous specification (Mitsubishi DVCR), the new specification (Samsung DVD-VCR), and the results of two consecutive movements. a) Shows the movement of the clock group, and b) shows how the rule chains with the movement of the clock channel state.



a. Mitsubishi DVCR
Without Consistency

b. Samsung DVD-VCR
Without Consistency

c. Samsung DVD-VCR
Consistent with Mitsubishi DVCR

Figure 8.7. User interfaces generated by the desktop PUC for a Mitsubishi DVCR and a Samsung DVD-VCR without consistency and the Samsugn DVD-VCR generated to be consistent with the Mitsubishi DVCR. Note that the clock functions are located under the Status tab for the Mitsubishi DVCR, under Setup for the Samsung DVD-VCR, and in a new Status tab in the consistent interface. Also note that Controls and Timed Recordings from the DVCR are located under the VCR tab on the Samsung DVD-VCR.

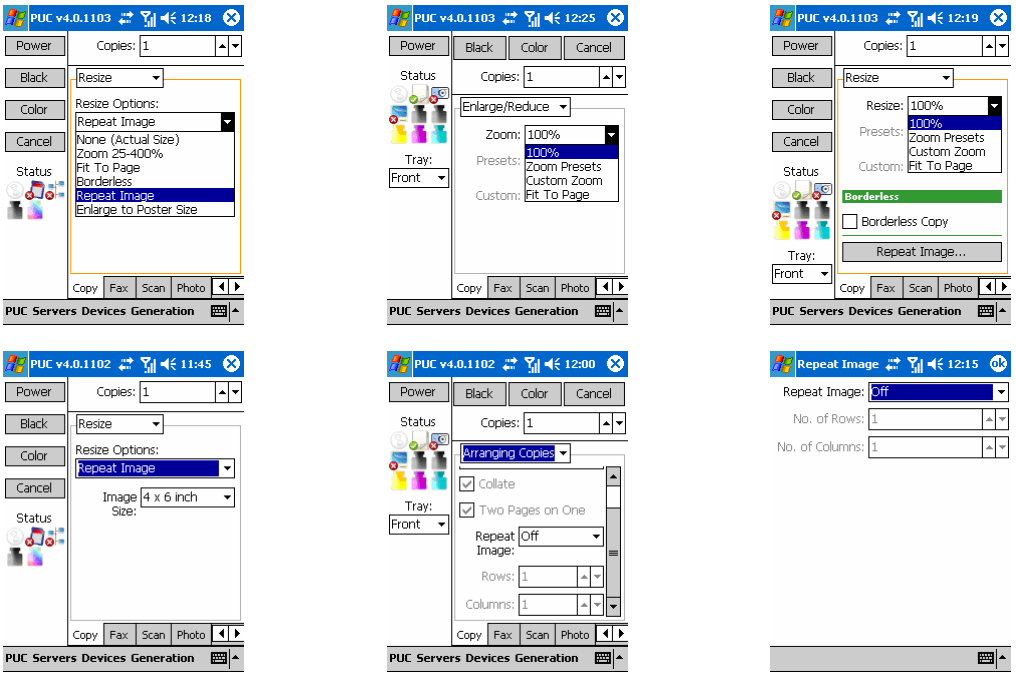
group and placed the Clock group into the new DateTime group. Any labels from the DateTime group in the Mitsubishi specification would have been replicated in the newly created group. Recreating the missing structure gives the interface generator more options for adding organization when creating the concrete interface.

The "Move Single Moded Item" rule handles moves that cross the boundaries of mutually exclusive groups. As mentioned above, there are two situations where such a move may create an unusable situation when the moved items are disabled, either because the interface generator may accidentally make it impossible to enable the items or the action required to enable the items is located far away from the moved items. In both cases, this rule causes any the moved item's dependencies to not be used in the generated user interface. Thus the item is always enabled, even if its actual dependencies indicate that it should not be. If the user manipulates the item when it should be disabled, the interface generator makes the appropriate mode changes in the background, confirms the users change, and then returns any modes back to their original settings without displaying any of these changes in the user interface. For example, the HP all-in-one printer has its fax setup functions located in its fax mode whereas the Canon all-in-one printer has the same functionality in its setup mode. When the HP interface is made consistent with the Canon, the fax setup functions are moved to the setup tab where, without the appropriate modifications, they could never be enabled. This rule allows these users to change the fax setup settings. When the user modifies a fax setup function, the rule causes the interface generator to automatically change to the fax mode, apply the user's change, and then switch back to setup mode. All of this can be done quickly in the background without causing any noticeable confusing changes in the user interface.

The "Move One To Many Split Items" rule handles the situation where one item in a specification is mapped to multiple split items in another specification, such as for the front display functions in the Mitsubishi and Samsung VCRs discussed above. The behavior of this rule differs, depending on direction in which consistency must be ensured. If the new specification contains the split items and the previous specification contains a single item, then all of the split items are moved to a location similar to that of the single item. If the situation is reversed, then the single item is duplicated in all of the locations where the split items were found in the previous specification. A future version of this rule might try to modify the single item to match each of the split situations, but I could not find a generalizable way to do this from the information currently available in the mappings. For example, when making the Samsung interface consistent with the Mitsubishi rules with more informa-

tion might be able to split the Samsung's overlapping brightness and content control for the front display into two independent controls to match the function split on the Mitsubishi. The control for brightness could be placed in a matching location for the brightness control on Mitsubishi, and the same could be done with the control for content.

The "Bring Together Split Dependents" rule handles a very particular situation that we have found in several specifications. A specific example appears in the HP and Canon all-in-one printers. The HP all-in-printer has several options for resizing a copied image, including zooming, creating a poster across multiple pages, and repeating the same image multiple times on a single page. Only one of these options can be chosen at a time on the HP. The Canon all-in-one printer also has these options, but several of these features are independent of each other. In particular, the image repeat function can be used with any of the other resizing options. Speaking more generally, this rule addresses the situation where two appliances share the same functionality but some of that functionality is moded on one appliance but not on the other. This situation is described in the knowledge base with a group



a. HP printer without consistency

b. Canon printer without consistency

c. Canon printer consistent with HP

Figure 8.8. A demonstration of the "Bring Together Split Dependents" rule. The top image shows the location of the resize mode function and the bottom shows the location of the repeat image function and its parameters. In column c, the dialog box shown on the bottom is opened by pressing the "Repeat Image…" button shown in the top image.

mapping that contains a state mapping and multiple task mappings involving the states from the state mapping.

A moving rule is needed in this situation because the unmoded functionality may not be located in the same place within the specification as the mode state variable. If the new specification contains the unmoded functionality, then this functionality must be moved near the state variable. If the previous specification contained unmoded functionality, then the moving rule creates a new button that makes appropriate mode change and places this button in a similar location to where the unmoded functionality was in the previous specification. If there were any functions dependent on this mode, they are also moved to the location of the new button (see Figure 8.8).

8.4.3.2 Re-ordering

The reordering sub-phase moves functions within groups to ensure a consistent ordering. For example, Figure 8.9 shows that the parameters for a timed recording have a different ordering between the Mitsubishi DVCR and the Samsung DVD-VCR. This sub-phase, like the moving sub-phase, traverses the abstract user interface until it encounters a mapping. Reordering rules operate on the children of a node however, so, unlike the moving sub-phase, there is no need to apply these rules to leaf nodes in the group tree.

Before the reordering rules are executed on a group, the sub-phase determines the previous specification with which the group will be made consistent. The sub-phase then creates a "block list" data structure for the group in the new specification and its equivalent group in the previous specification. The block list is important because it allows rules to analyze and manipulate functions as if the functions are in a list, when the underlying representation of structure in the abstract user interface is a tree. The tree structure can become problematic when a function's objects span multiple levels of tree hierarchy, as in the case of the "When" mapping on the Samsung DVD-VCR (shown in Figure 8.9). Each set of adjacent objects

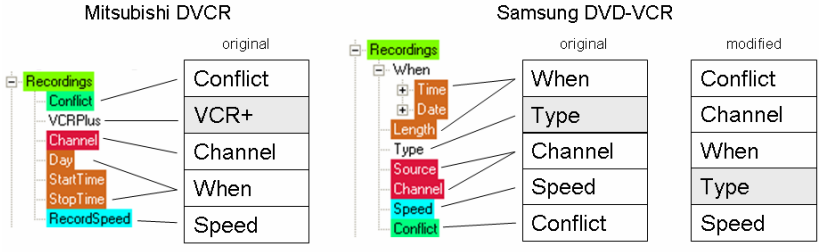


Figure 8.9. Block lists created for the timed recordings groups of the Mitsubishi DVCR and Samsung DVD-VCR. "VCR+" and "Type" are unmapped blocks in the block lists.

with the same mapping becomes a block, which is stored in the list in the order they would appear in a generated interface (see Figure 8.9). Consecutive unmapped objects are also stored as blocks in the list. The block lists are processed by the reordering rules, resulting in a new block list that specifies the final ordering for the group.

The current reordering phase has a rule that starts by searching the block lists from the new and previous specifications to find blocks with the same mapping. These blocks are re-ordered to match the previous specification. Unmapped blocks are moved with the block that precedes them, in accordance with the second unique function heuristic. For example, notice in Figure 8.9 that the Type block followed the When block to its new location. Unmapped blocks at the beginning or end of a block list are not re-ordered, in accordance with the third unique function heuristic (see section 8.4.1).

8.5 Generating the Concrete Interface

Once the abstract user interface has been created and modified to ensure consistency, it is time to generate the concrete user interface. In the next two sections I describe how the concrete interface is generated for the PocketPC, desktop, and Smartphone platforms.

8.5.1 PocketPC and Desktop

The PocketPC and desktop interface generators use the same process for generating their concrete interfaces, which is split into two steps. The first step generates an initial version of the user interface with only the most important structural elements. The second step looks for layout problems in the initial version of the generated interface, such as panels that require scroll bars to display all of their content and controls that are not wide enough to display their labels. Where problems are found, rules attempt to fix the problems by adding organization or changing the layout. Using this approach, the user interface is built with the minimum amount of organization needed for the size of the screen and the number of appliance functions. Minimal organization is beneficial, because it allows users to see all of the functionality of the interface in fewer screens and reduces the number of clicks necessary to navigate between functions.

8.5.1.1 Creating the Initial Interface

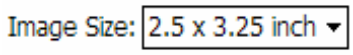
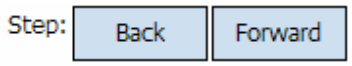
The concrete interface is represented by an "interface tree" that describes the panel structure of the user interface. The children of the root of this tree represent the different windows

used in the interface and the leaf nodes represent panels in the interface. Branch nodes specify how their child panels are placed relative to each other, either as a set of panels separated with vertical or horizontal edges or a set of overlapping panels. Each panel described in a leaf node contains a list of rows, which describe how controls should be placed relative to each other. Rows have five different possible layouts, as shown in Table 8.3. Of these layouts, the two labeled formats are preferred because they create a simple grid with labels on the left and controls on the right. Some controls, because of their content, must be displayed across the entire panel and thus use the full width row. The full width row and the multiple items row are also used if the available screen space is limited.

The interface tree is assembled by traversing the structure of the abstract user interface and applying a set of concrete interface construction rules. There are two types of these rules: one set creates new panel structure based on mutually exclusive groups found in the specification, and another set find the appropriate concrete interaction object (CIO) for each AIO and place that CIO into a row.

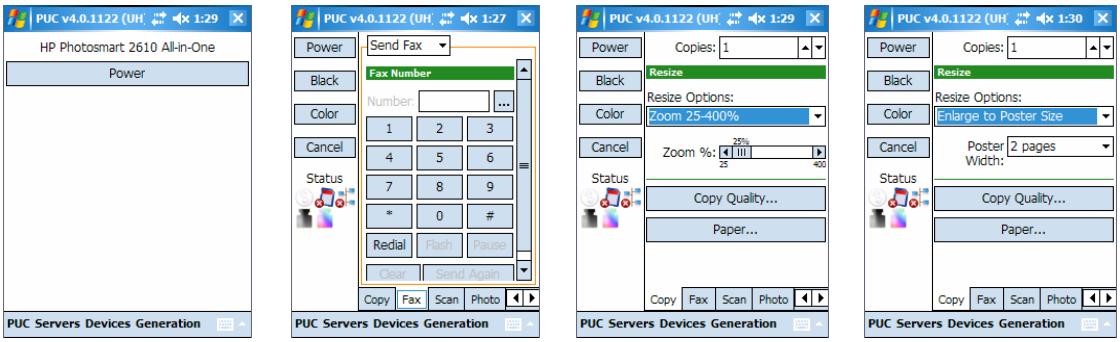
There are three rules that add structure to the concrete interface based on mutual exclusion. Two of these rules look for particular properties of the mutually exclusive situation and create structure if those properties are found. If none of these rules create structure, then the

Table 8.3. The row layouts supported by the PocketPC interface generator.

Row Name	Example	Description
Labeled One Column		A single control is shown on the right and an optional label is shown on the left. The space allowed for the label is fixed for a panel, typically at 40% of the available width.
Labeled Two Column		Two controls are shown on the right and an optional label is shown on the left. The label space is again fixed per panel at the same value as the labeled one column row.
Full Width		One control occupies the entire width of the panel.
Multiple Items		Multiple controls occupy the entire width of the panel. Controls are sized to fit their needs and, if necessary, will be proportionally enlarged to fill the full panel width.
Overlapping Panels	See Figure 8.10c-d	This row allows an overlapping panel to be added inline with the rest of the controls. This is used for the fourth mutual exclusion structure rule, discussed below.

final rule creates a default structure that fits any situation. The most commonly used mutual exclusion rule creates overlapping panels that are controlled either by a tab control and a sidebar for any non-dependent items (see Figure 8.10b-d). This rule has two requirements: there must be a set of controls that are mutually exclusive for every value of the mutually exclusive state, and the state must have labels for all of its values. The first requirement is necessary because the overlapping panel control will also control the possible values of the state variable. If there was a not a set of controls for one value of the variable, then no panel would be created and the user could not change the state variable to that value (blank panels are also not allowed). The second requirement ensures that proper labels are available for each of the panels. The third requirement is that tabs have not been used yet in the interface, because multiple sets of tabs can be quite confusing. When tabs do exist, structure is added by the third rule discussed below.

The second organizing rule is a special case rule for handling the power button. The power button is a unique situation because typically it is the first item in the specification and all of the other functions in the specification are only active when the power variable is set to one of its particular values (usually Boolean true). If this situation is found, the second rule will create two overlapping panels at the very top of the interface. Both panels will have a power button on them, which allows the user to turn the appliance on or off. In the off state, a special appliance screen is shown with a label showing the appliance name and a large power button. The on state shows all of the controls for the appliance, appropriately organized (see



a. b. c. d.

Figure 8.10. Screenshots of the HP all-in-one printer interface demonstrating the three mutual exclusion rules. a) The special power off screen generated by the second rule. The remaining shots are of the power on view. b) The fax mode of the all-in-one printer. This mode is accessed through tabs at the bottom of the screen created by the first rule. Another tab can be seen in screen shots c and d. c) The copy mode of the all-in-one printer with the resize options set to zoom 25-400%. The resize options state is a mode with several different options created by the third rule. d) Another view of the copy mode with the resize options set to poster size.

Figure 8.10a-b).

The third rule places the non-dependent items and the mutual exclusion state in a panel as they normally would be. Immediately after the mutual exclusion state, an overlapping panels row is added containing a panel for each of the mutually exclusive sets of controls. When the user manipulates the control for the mutual exclusion state, the top-most overlapping panel is changed to display the panel with the currently active controls (see Figure 8.10c-d).

As the interface tree creation process traverses the abstract user interface tree, it maintains a reference to the concrete panel that represents the current portion of the abstract interface. A set of rules add rows to this panel whenever an AIO is encountered in the abstract interface. The process of adding a new row requires two steps: first an appropriate CIO must be chosen for the AIO and then an appropriate row must be chosen for the CIO. The CIO is chosen by querying the AIO with a set of criteria, from which a platform-specific method of the AIO will return the most appropriate CIO for that platform and criteria. There are two criteria currently supported: whether or not a read-only CIO is needed and whether the CIO should work within a list. Other criteria, such as maximum size, could be supported in the future.

Based on the CIO, a set of three rules choose the appropriate row. Any remaining rules are not applied once a rule has created a row. The first rule checks to see if the current CIO is located in a labeled group with just one other CIO and that both of the CIOs are either buttons or checkboxes. In this case, a Labeled Two Column row is created for the two CIOs using the group label. The second rule checks the CIO to see if requires a full width layout and, if so, creates a full width row for the CIO. The third rule creates a Labeled One Column row for the CIO using a label, if any, from the AIO. As mentioned above, these rules prefer the Labeled row formats because they create a simple, regular grid of labels and controls. If a Labeled row format does not provide sufficient space for one of its controls, then the format may be changed by one of the layout fixing rules discussed in the next section.

After the abstract user interface has been completely traversed, all panel structure has been added, and all CIOs have been created and placed into rows, then the concrete sizing of all panels, rows, and CIOs is determined. Size is determined by a two-pass depth-first traversal of the concrete interface tree. The first pass calculates the minimum and preferred size for all panels based on the CIOs they contain. The second pass determines the actual size of each

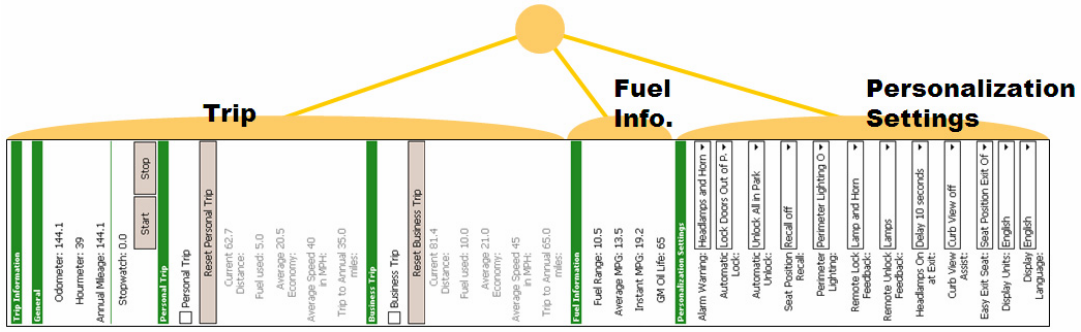


Figure 8.11. The interface generated for the GMC Yukon Denali driver information console without the use of any layout fixing rules (rotated to fit better on the page). The high-level structure from the abstract user interface underlying this panel is shown above the interface.

Table 8.4. The layout fixing rules used by the PocketPC interface generator.

Problem	Name	Description
Insufficient Height	Fix with Tabs	Breaks a long panel into multiple overlapping panels that are controlled by tabs. This requires that the panel only contains labeled groups, so that all controls can be placed in a tab and each tab has a name.
	Fix with Combo Box Panel	Breaks a long panel into multiple overlapping panels that are controlled by a combo box at the top of the panel group. Like the Fix with Tabs rule, this requires that the panel only contains labeled groups, so that all controls can be placed on a panel and each panel has a name in the combo box.
	Fix with Vertical Split	Breaks a long panel into two separate panels that are oriented vertically. Any items not in groups are moved into one panel and the remainding items are placed in the other panel. This allows one of the other fixing rules to add more organization in the second panel.
	Fix with Horizontal Split	Similar to the Fix with Vertical Split rule, but creates panels that are oriented horizontally instead of vertically.
	Fix with Row of Buttons	Looks for multiple consecutive buttons on a panel and compresses them into the fewest number of rows possible given the width of the panel and the buttons' labels.
	Fix with Dialog Box	Looks for a labeled group of controls on the panel, moves the group of controls to a dialog box and replaces them on the original panel with a button that opens the dialog box.
Insufficient Width	Fix One Column with No Label	If the problem occurs on a one column row with no label, the one column row is replaced with a full-width row that only displays the control (no space is left for a label).
	Fix Two Column with No Label	If the problem occurs on a labeled two column row with no label, the controls are moved to a row that does not save any space for displaying a label.
	Fix Too Wide One Column	If the problem occurs on a one column row with a label, then the label and control are placed into separate rows with the label above the control.

panel based on available screen space and the minimum and preferred sizings. During the second pass, if an actual panel size must be smaller than the panel's minimum size or if a row cannot be as wide as its minimum width, then a "layout problem" is noted by the interface generator. These layout problems are addressed by the layout fixing rules discussed in the next section.

8.5.1.2 Fixing Layout Problems

There are nine rules for fixing layout problems of insufficient height or width (see Table 8.4). For each layout problem, the rules are tested in order until a rule is found that can provide a fix to the problem. After a rule fixes a problem, the new interface is searched for any layout sub-problems that became apparent after the fix was applied. This process is repeated until either no layout problems remain or the remaining problems cannot be fixed by any of the rules. For example, the interface for the GMC Yukon Denali driver information console before the layout fixing rules would appear as one long column of controls, as shown in Figure 8.11. This column is much longer than the screen height, so layout fixing rules are applied to fix this problem. The first rule is the Fix with Tabs rule, which finds that the long panel can be broken up into three labeled groups. Because no tabs have been used to organize this interface yet, the rule will break the long panel into three overlapping panels based on the labeled groups. Tabs are used to allow the user to navigate between these panels. The longer trip section can then be broken down further using the Fix with Combo Box Panel rule to create three overlapping panels for the Trip Information, Personal Trip, and Business Trip groups (see the final interface in Figure 8.1b).

8.5.2 Smartphone

The Smartphone interface generator produces list-based interfaces, rather than the panel-based interfaces of the PocketPC and desktop interface generators. This different style of 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 list hierarchy is built starting with the topmost group in the abstract user interface. A list is constructed by making each child group that is labeled into a child list. Every AIO that is encountered is added to the list as an item. Groups are not required to have labels, so not all groups in the abstract interface will have corresponding child lists in the concrete 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 abstract interface structure. The result of the list building process is each set of functions being available from a separate list that is accessed from the same parent list (see Figure 8.12a). 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 8.12b-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.

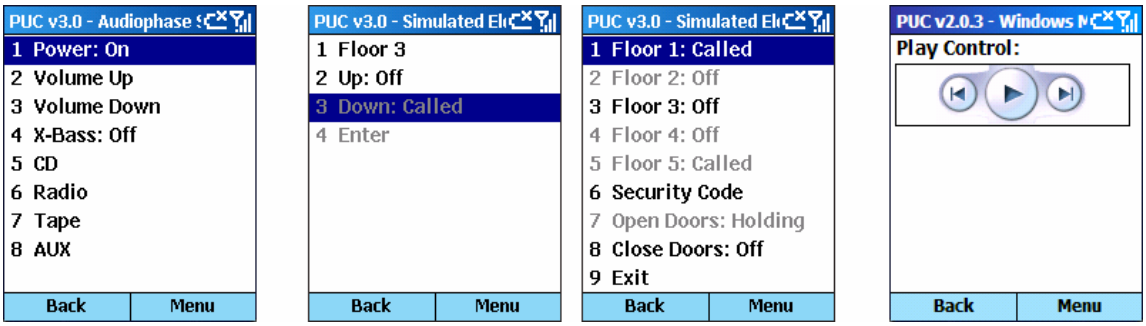
There are two constraints of the Smartphone interface that need to be addressed:

- 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.

There are currently 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 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." Restrictions on how a



a. b. c. d.

Figure 8.12. 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.

function may be displayed are based on the AIO that was selected for that function in the abstract user interface.

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 8.13a). 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 8.2c,e and Figure 8.13a).

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 8.13b). 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

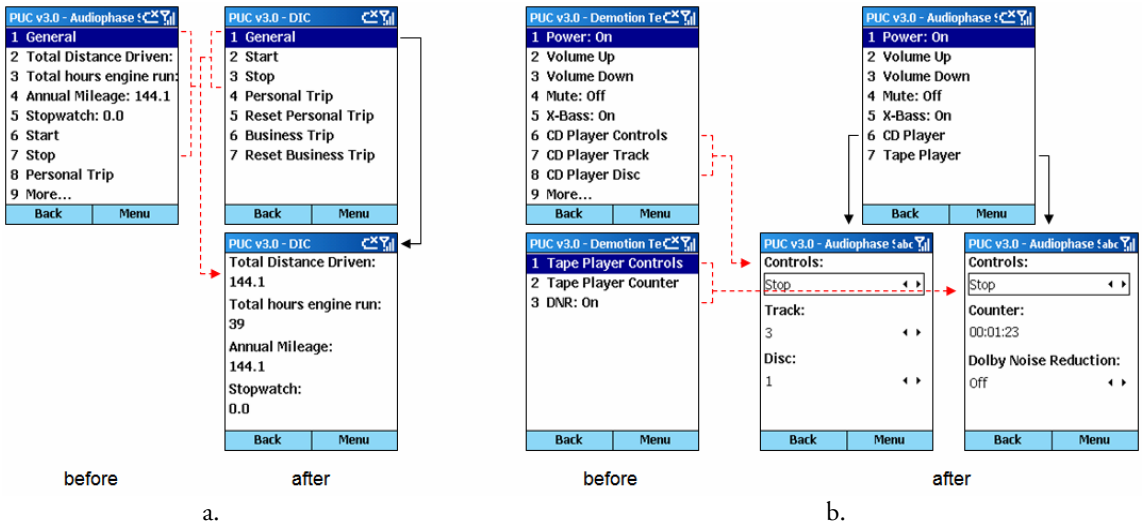


Figure 8.13. Diagrams showing how the first (a) and second (b) rules for optimizing the list structure behave. Black solid arrows indicate how the screens are connected and red dashed lines indicate changes made by the rules. Note that in (a) some items were list-only and thus were promoted to the top-level list while the others were placed on a panel. The items all happen to be labels, so this panel is a summary pane.

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 built in reverse priority order until the current list contains nine items or less.

I have experimented with several different methods for assigning a function to the "most common" soft button on the Smartphone. Initially the PUC 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 I felt that it might be more useful to assign common functions from the appliance to the button instead. I investigated two approaches: a static approach using priority information from the appliance specification and an adaptive approach based on recorded usage information.

The 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, the PUC chooses 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. The PUC selects 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, the PUC uses the algorithm from the first method to select the function. The soft button is currently changed every time the screen changes or

the user invokes a function and experimenting with other times is the subject of future work. 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.

No formal evaluation has been conducted 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 there are no regular users of the Smartphone interface generator 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 the PUC system currently does have any way for a specification author to include icons as a label for functions.

8.6 Modifying the Concrete Interface for Consistency

The preceding sections have discussed the first five phases of the PUC's interface generation process. At this point, the process has transformed an appliance specification into a abstract user interface, modified the abstract interface to ensure consistency, and transformed the abstract interface into a concrete user interface that is specific to a particular platform. Before the final user interface is displayed to the user, several rules check the concrete interface for inconsistencies with previous interfaces and provide fixes for any that are found.

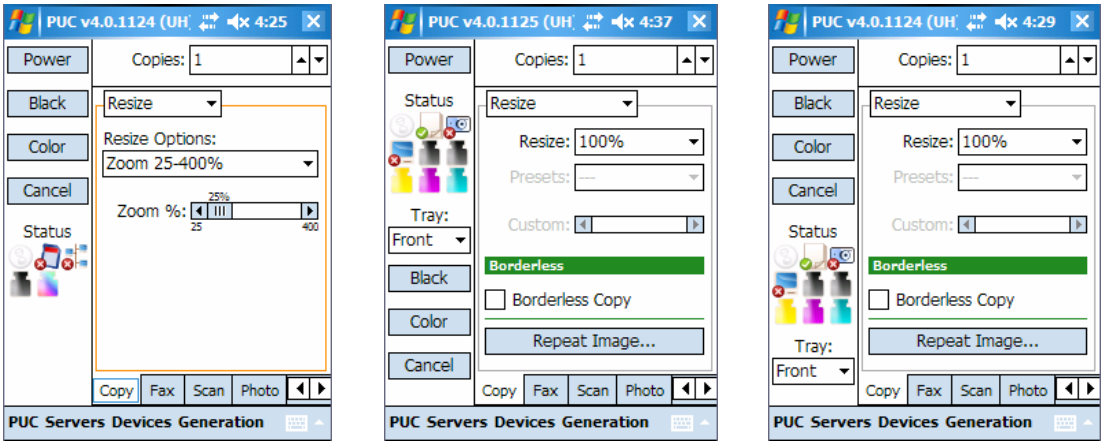
Inconsistencies may occur for at least two reasons. First, for the PocketPC and desktop there is often a mismatch between the deep hierarchy of the abstract interface and the flat hierarchy of the concrete interface. This mis-match sometimes introduces inconsistencies in the ordering of controls within a panel, which could not be discovered because of the extra depth in the abstract user interface. Second, the concrete interface generation may have used differ-

ent controls for organization or placed panels in different orientations than in the previous user interface. Three concrete consistency rules for the PocketPC and desktop have been implemented to address these issues. There are currently no concrete consistency rules for the Smartphone interface generator because, at least so far, there has not been a need for them.

The difficulty for implementing these rules is to determine which panels correspond between the new interface and the previous interface. The current approach for determining correspondence is to link each panel to the highest-level group it contains within the abstract user interface. A panel in the new interface corresponds to a panel in the previous interface if the groups of both panels have a mapping between them. This is not a perfect algorithm, because sidebars, for example, often mix controls from several groups at different levels within the specification, however it seems to be effective for the cases that have been tried to date.

The first concrete consistency rule addresses the re-ordering of rows within panels. When two panels are found to correspond between the previous and new interface, the rows from both panels are converted into block lists, the new block list is re-ordered based on the previous list, and then the list is converted back to rows. The process used is similar to the one used for re-ordering the abstract user interface. Figure 8.14 shows an example of how this re-ordering rule affects the Canon all-in-one printer interface when it is generated to be consistent with the HP all-in-one printer.

The second and third rules address inconsistencies that may have arisen during the genera-



a. HP without consistency

b. Canon consistent with HP without concrete rules

c. Canon consistent with HP with concrete rules

Figure 8.14. Interfaces generated for the HP and Canon all-in-one printers demonstrating the effects of the concrete interface re-ordering rule. Note the difference in the order of the Black, Color, and Cancel buttons.

tion of the concrete interface. The second rule adds overlapping panel organization to a user interface if it was used in the previous interface and more than one control can be placed on each of the overlapping panels. The exact type of overlapping panel widget is chosen based on the previous interface. The third rule modifies the sidebar panels that the PUC interface generator sometimes creates around overlapping panels. This rule checks the orientation of the side panel, which may be either horizontal or vertical, and ensures that the orientation is the same as in the previous interface.

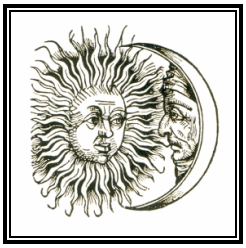
8.7 Results and Discussion

After the concrete interface is generated and modified by the concrete consistency rules, then the interface can be presented to the user. The generation process is fast enough to run on actual PocketPC and Smartphone devices, which differs from most other interface generation systems for handhelds which have offloaded generation to a remote server [Ponnekanti 2001, Gajos 2005a]. The most complex appliance interface that the PUC has generated, the GMC Yukon Denali navigation system, takes approximately two minutes to generate on a Hewlett-Packard iPAQ h4150 PocketPC device with a 400 MHz Intel XScale processor running the .NET Compact Framework 2.0. Most appliances take less than one minute to generate. All interfaces, including the navigation system, generate in less than 5 seconds on a current desktop computer. The consistency rules do not noticeably affect generation time on either the PocketPC or Smartphone. The biggest impact on performance time for the PocketPC is the layout fixing phase, which must re-layout the interface after every layout fix in order to evaluate whether new layout sub-problems have been created.

The interface generator has been used to successfully generate interfaces for all of the 33 specifications that the PUC team and I have authored. As discussed in Chapter 5, this collection of specifications covers a wide range of appliance types and includes many complex specifications. Chapter 10 shows that the generated interfaces can be more usable than the human-designed interfaces for two complex all-in-one printers, which I believe are representative of the many complex appliances with which the PUC system could be used. That chapter also shows that the consistency algorithms shown here have usability benefits.

The interface generator does have some trade-offs. The rule-based approach that I chose has several benefits, including allowing for timely generation of interfaces on low-resource handheld devices and the ability to always produce the same output given the same input, but it

also has several drawbacks common to all rule-based systems. In particular, new rules must be designed carefully to avoid unintended interactions with existing rules. Interface generation also involves balancing many trade-offs and rules are poorly suited to making decisions that optimally balance these trade-offs. I have used two different rule-based approaches to iteratively improve interfaces during generation (the PocketPC and desktop interface generators start simple and add complexity while the Smartphone generator starts complex and simplifies), but neither can guarantee an optimal result. Both are also sensitive to the structure of the abstract interface and the order in which rules are applied. Optimization algorithms are less predictable however and require far more computational resources to process. A hybrid approach may be most successful, but this is a subject for future research.

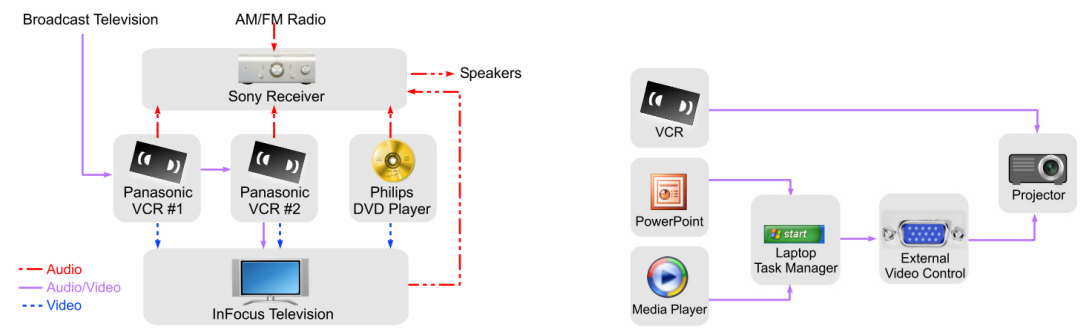


CHAPTER 9

Aggregating User Interfaces[6]

The last chapter discussed how the PUC automatically generates user interfaces for a single appliance. This chapter describes how the PUC is able to generate a user interface that combines functionality from multiple appliances that are connected in a system. A key input to this generation process is a content-flow model, which describes the different routes that content take within the system from a source appliance, possibly through one or more passthrough appliances, to a sink appliance where that content is either displayed for the user or recorded for later use. From the content flow model, the PUC creates two types of interfaces: a flow-based interface that allows users to quickly specify their high-level goals for the system of appliances, and aggregate interfaces that provide low-level control while a flow is active. The interface aggregation features of the PUC are collectively known as Huddle.

[6] The work in this chapter was originally described in Jeffrey Nichols, Brandon Rothrock, Duen Horng Chau, and Brad A. Myers. "Huddle: Automatically Generating Interfaces for Systems of Multiple Appliances," in *Proceedings of the 19th annual ACM symposium on User Interface Software and Technology* (UIST). Montreux, Switzerland. Oct. 15-18, 2006. pp. 279-288



a) Home Theater b) Presentation Room

Figure 9.1. Configuration of appliances in two multi-appliance system scenarios: a) a home theater, and b) a presentation room.

9.1 Scenarios

Throughout this chapter I will use two scenarios to demonstrate the PUC's interface aggregation features: a home theater and a presentation room.

The home theater setup (see Figure 9.1a) includes five appliances: an InFocus television (61md10), a Sony audio receiver (STR-DE935) with attached speakers, a Philips DVD player (DVDP642), and two identical Panasonic VCRs (PV-V4525S). This setup supports many common tasks, such as watching television, watching a movie from either a DVD or videotape, and listening to the radio. It also supports a number of more complicated tasks, such as copying from a tape in VCR #1 to a tape in VCR #2, or watching television on one channel while recording up to two other channels. Sometimes tasks can be mixed, such as watching a sporting event on television while listening to a radio broadcast of the play-by-play. Certain tasks are impossible with this setup, such as recording a DVD to videotape, recording the radio, or recording from a tape in VCR #2 to a tape in VCR#1. As we will show below, the PUC's flow-based interface makes it clear to the user which flows are not possible.

The presentation room configuration has three physical devices (see Figure 9.1b): a projector, a VCR, and a laptop. The laptop's functions however, have been separated into several independent "logical appliances" which include the PowerPoint and Windows Media Player applications, the task manager, and control of the external video port. This configuration supports common presentation tasks such as showing slides, showing video from the laptop, and showing video from a VCR tape.

9.2 Content Flow for Understanding Systems of Appliances

In order to generate aggregate user interfaces, the PUC needs some knowledge of how each appliance in a system relates to the others. Current integrated user interfaces, such as those that can be built with technology from Crestron or AMX, require new configuration information for each system of appliances. This approach requires too much work for end users however, who will typically pay professional system integrators to create their integrated interfaces. For the PUC, my goal is to support the generation of aggregate interfaces with minimal effort on the part of the end user. Ideally, any work required by the end user should also scale well with the number of appliances connected in a multi-appliance system.

I found that the content flows between appliances in a multi-appliance system were useful for understanding the system and generating aggregate interfaces. In particular, content flows have two important properties:

1. Content flows seem to be closely related to user goals with multi-appliance systems. For example, in a home theater, the user may want to watch a DVD movie, which involves seeing the video on the television and hearing the audio through the stereo's speakers. To accomplish this, each of the appliances in the home theater must be configured to allow content from the DVD player to flow to the appropriate places.
2. The content flows of a system can be described as the separate flows within each appliance combined with a wiring diagram showing how all of the appliances are connected. This is an important property, because it divides the modeling work among the manufacturers of the appliances. The only system-specific input needed by the PUC's aggregation algorithms is a diagram showing how the appliances are connected, which can be supplied by another application, a future wiring technology or the user. Furthermore, this separation also pushes most of the work to the manufacturers because most of the the complexity in any content flow model is found in the internal flows within each appliance.

In order to support content flows, two sections were added to the PUC specification language to specify the physical ports of the appliance and the internal content flows that use those ports (see section 5.2.2). The content flows within an appliance are represented using three different structures:

- **Sources** represent content that originates within the appliance, such as from a DVD player playing a DVD or a VCR playing a videotape. Display devices that have internal tuners, such as televisions receiving broadcast signals through antennas, are not defined as sources however, because the content does not originate inside of the tuning device. Instead, broadcast signals are described as a special "external source" that must be routed through a tuner to be viewable by the user.
- **Sinks** represent locations where content may either be displayed to the user or stored for later retrieval. For example, the television screen, receiver speakers, and VCR tape (for recording) may all be sinks for content in our home theater scenario.
- **Passthroughs** represent an appliance's ability to take in some content as an input and redirect it through one or more of its outputs. For example, the InFocus televi-sion in our scenario has the capability of taking the audio it receives as an input and making it available as an output for other appliances. Tuning appliances, such as cable television set-top boxes, are also represented as passthroughs, which usually take a multi-channel input from an antenna and output single channel data.

The passthrough structure is particularly important, because it allows Huddle to track the flow of content from its origination point, through multiple appliances, to its final destination. Previous systems, such as Speakeasy [Newman 2002] and Ligature [Foltz 2001], have used only sources and sinks to model the path of data within a system. Using their approach, it is difficult to know whether the content a device is receiving as input is being redirected through an output, which makes determining the full content flow impossible. Without knowledge of the full content flow from start to finish, the task the user is trying to perform could not be determined and a useful interface could not be generated for it.

9.3 Aggregation Architecture

The PUC controller device performs all of the aggregation of content flow information from the appliance specifications and generates aggregate user interfaces based on this information. Even when an aggregate interface is available, the PUC still allows users to use the interfaces generated for the individual appliances. An overall view of the aggregation architecture is shown in Figure 9.2.

The PUC's interface aggregation requires three types of input in order to function. First, it requires a wiring diagram that describes how the multi-appliance system is wired together,

which is currently is specified by hand in XML (see the schema in Appendix C.3). The wiring diagram contains a number of wire <begin, end> pairs corresponding to the physical wires that connect the appliances. The second input required is the set of all PUC appliance specifications from each of the appliances in the multi-appliance system. By combining this information together, the PUC creates a complete model of the possible content flows through the entire system (see the center portion of Figure 9.2), which is then used to generate user interfaces.

The aggregate interface generation also makes use of the knowledge base that is primarily used to generate consistent interfaces (see section 6.4.1). In the context of aggregation, this information allows the PUC to create interfaces that organize functions from multiple appliances in a meaningful way.

The PUC produces two kinds of interfaces to help users interact with their multi-appliance systems. The Flow-Based Interface (FBI) allows the user to quickly create and activate content flows between appliances by tapping or dragging the icons for desired sources and sinks

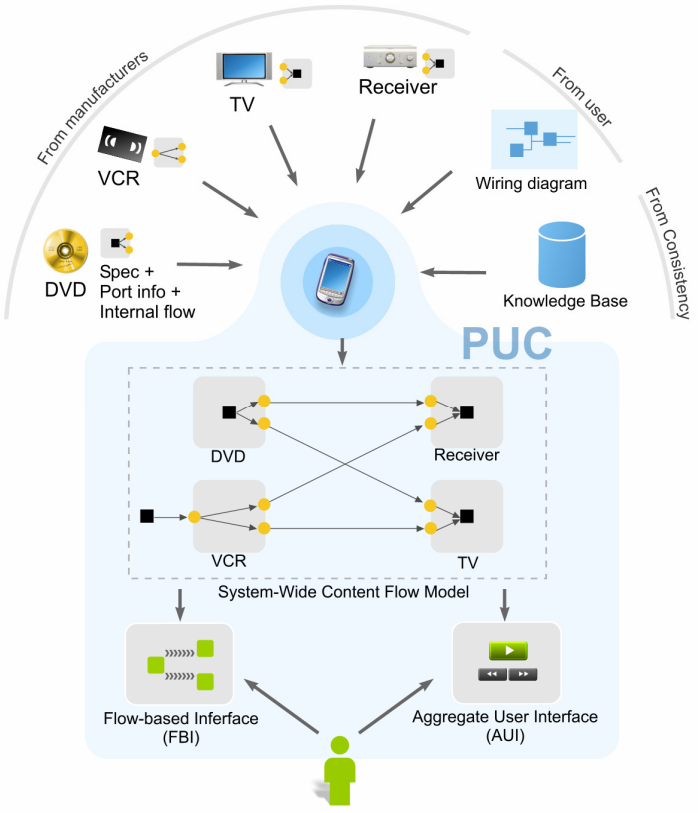


Figure 9.2. Architecture of the aggregate interface generation features.

onto the screen. The goal of this interface is to make high-level tasks easy to execute with the multi-appliance system. To date, the flow-based interface has only been implemented on the PocketPC platform. Modifications would be needed for phone- or speech-based interfaces to replace the direct manipulation interactions used in the current version of this interface.

The PUC also generates Aggregate User Interfaces that combine functions from multiple appliances into a single user interface. Various types of aggregate user interfaces support different tasks within the multi-appliance system. The *Active Flow Control* aggregate combines the most common control functions associated with the active content flows into a single interface, with the goal of making common content manipulations (such as volume control) easy to access. The *Setup* aggregates make infrequently used configuration parameters easy for the user to access, with the goal of supporting expert usage of the appliance system. Finally, aggregates can merge some functions that occur on multiple appliances into a single point of control on one interface. This allows the user to do such things as set the current time in an aggregate and have this change be automatically broadcasted to each appliance in the system.

9.4 Flow-Based Interface

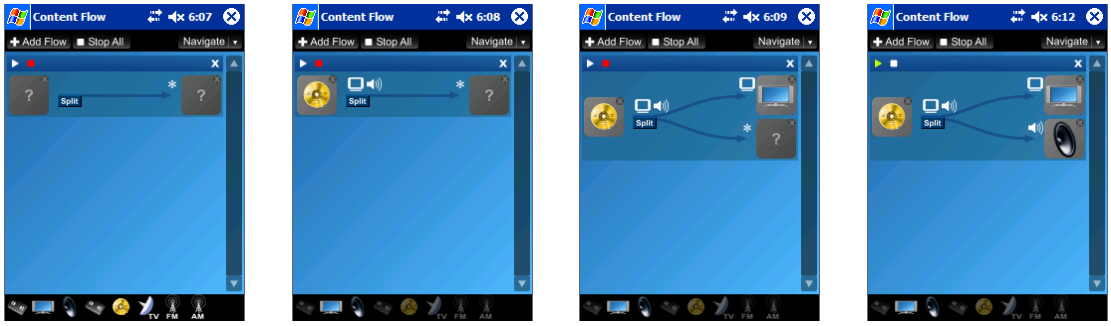
The FBI is designed to allow users to quickly specify a flow from one source of content to one or more content sinks. For example, the user might specify a flow from the DVD Player's disc to the television's screen and the receiver's speakers. When the user activates this flow, the PUC inspects the dependencies of each of the flow's elements, generates a plan to satisfy these dependencies, and executes that plan to enable the flow. If the flow cannot be enabled, perhaps because of other active flows that the user has already specified, the system will prompt the user with a dialog box and attempt to help the user resolve the problem. Several examples of the FBI are shown in Figure 1.7 and Figure 9.3.

To make the idea behind the FBI clear, the interaction that a user would have with the interface in order to start watching a DVD movie will be described. Figure 9.3a shows the FBI in its initial blank state. Near the top of the screen is a blank flow with empty spaces for a source and sink, with an arrow between them. At the bottom of the screen is the appliance bar, which contains an icon corresponding to each appliance that has a source or sink in the system. The appliance bar may grow upward to allow space for all of the available appliances in a system to be shown. Currently there is no limitation on how much the appliance bar can grow, but this seems unlikely to become an issue because in my experience it is rare for a

multi-appliance system to have more source and sink appliances than can fit in two rows of icons. Appliance icons are currently assigned by a hard-coded matching function in the interface generator, but a future version could download icons from the appliances or the Internet. Additional flows may be added to the screen, by pressing the "Add Flow" button at the top of the screen, and the scrollbar on the right allows for scrolling when more flows have been added to the list than can be shown. I envision two usage scenarios for this interface: the user creates flows for each of the common tasks and switches among them, or the user uses just one flow and modifies it as necessary to suit the current task.

When a user wishes to begin a flow, an icon is dragged from the appliance bar to one of the empty spaces in the blank flow. The empty space highlights when the icon is dragged over it, indicating that the appliance icon may be placed there. Once the DVD icon has been placed in the source location (see Figure 9.3b), content type icons appear on left side of the arrow, and the icons in the appliance bar corresponding to appliances that cannot be sinks for the DVD player source are grayed out. This includes icons that correspond only to sources, such as the broadcast television and radio icons, and the VCR icons that cannot be sinks for DVD content because of our home theater's particular wiring configuration. The user can now see that the receiver and the television are the only available appliances that will work with the DVD player. In this scenario, the user first drags the television to the empty sink space on the flow. At this point, the green play button will become enabled because this configuration corresponds to a valid flow (Note that the television speakers can be a sink for audio content). The asterisk above the arrow on the right side indicates that the flow-based interface will infer the type of content to route to the television based on the specified sinks.

The user now wishes to add the receiver as an additional sink. To do so, the "Split" button is



a. b. c. d.

Figure 9.3. An example of using the Flow-Based Interface to configure a DVD player to play video through a television with the audio routed through the stereo's speakers.

pressed underneath the arrow on the left. This causes the flow to be split into two arrows and a new empty sink space to be created (see Figure 9.3c). To add the receiver's speakers as a sink, the user can then drag the receiver to the empty space. In this scenario, the PUC is able to automatically infer that the TV speakers should not be used, because an audio sink was added to the flow. If the user wanted audio to come from both sets of speakers, this wouldbe indicated by tapping the content type icon next to the television and selecting the audio/video content type.

The user can now click the play icon in the flow's title bar, which invokes the planner to automatically activate this flow. If a successful plan is found, the appliances will be automatically configured, the play icon will turn green, the stop icon will turn white, and a bubble will appear (similar in appearance to the bubble shown in Figure 1.7d) to inform the user that the flow has been activated. Figure 9.3d shows the interface once the user has dismissed the bubble. If a plan cannot be found, a bubble will appear to help the user resolve the problem (see Figure 1.7d). One difficulty with planning algorithms, such as the GraphPlan algorithm that the PUC uses, is that they cannot produce useful error messages when planning fails. Therefore, the PUC first uses two approximation checks to search for conflicting appliance variables and active flows, allowing a more useful error message to be produced.

The first conflict check searches the dependencies of the newly specified flow to see if any read-only variables have values that make activating the new flow impossible. Such variables usually reflect the physical status of the appliance, which the user can address once informed. For example, the DiscIn variable of the DVD player might be set to false when the user pressed the play icon in our previous example. If this happened, the system would then ask the user if they can rectify this problem. Although the current language for this error message can be somewhat stilted, we do provide predefined strings for common problems, such as there being no disc in the DVD player (see Figure 1.7d).

After the PUC checks for variable conflicts, it checks to see if any currently active flows conflict with the new flow. To perform this check, it examines the dependency information associated with the newly specified flow and the dependencies for any active flows, looking for variables that must have more than one value for the flows to be active simultaneously. If this situation is found, then the PUC immediately goes back to the user to ask which of the conflicting flows the user wants to use.

Once the PUC has found that no obvious conflicts exist, it executes the planning algorithm to find a valid plan for activating the new flow. If a plan is found, then the system will carry out that plan to create the right configuration of variables that will activate the new flow and maintain the state of any existing flows. The planning algorithm may still fail however, such as when second-order dependencies conflict. In our experience, these conflicts are rare, but when they occur the PUC asks the user to choose between finding a plan that activates the specified flow and disables the currently active flows or finding a plan that activates the specified flow without considering the effects on other flows. If a plan is found in either case, the user is prompted again before carrying out the plan to make it clear which flows will be deactivated by the new plan.

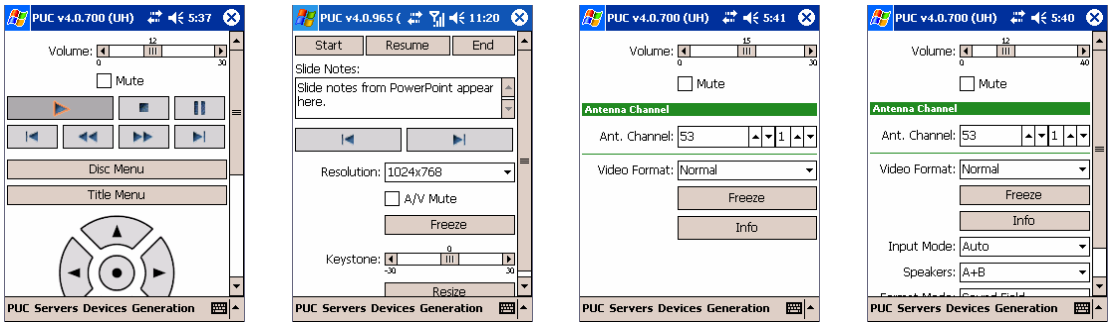
The FBI also provides a way to navigate to the other aggregate user interfaces and the PUC interfaces for the individual appliances. In the upper right corner of the FBI is a "Navigate" pull-down menu, which allows the user to navigate to the different aggregate user interfaces that can be generated (discussed next). Double-clicking on any appliance icon, either in the appliance bar or in any flow, allows the user to navigate to the full interface for that individual appliance.

9.5 Aggregate User Interfaces

The FBI provides an interface for users to accomplish high-level goals within the multi-appliance system, such "watching a DVD movie" with a home theater. There is still a need however to provide the user with finer-grain control of the individual appliance functions. For example, the user may wish to pause the DVD while it is playing to take a phone call, or go to the next slide in a PowerPoint presentation. A user might also discover that the movie is too dark requiring adjustment of the brightness of the television, or that the keystone setting needs to be adjusted on the projector.

To address these problems, the PUC provides the user with several Aggregate User Interfaces (AUIs) that combine functions from each of the appliances in the system to create useful task-specific interfaces. The PUC currently can generate four different AUIs: Active Flow Control, Active Flow Setup, General Setup, and Merged Functions.

It is important to note that the user also has access at any time to the full interfaces for each appliance. Thus it is not our goal to provide access to the full set of appliance functionality



a. b. c. d.

Figure 9.4. Active flow control interfaces: a) playing a DVD movie with the video shown on the television and audio coming through the stereo's speakers, b) presenting Powerpoint slides through a projector, c) watching broadcast television with audio playing through the television's speakers, and d) watching broadcast television with audio playing through the receiver's speakers. Note that the volume control in (c) and (d) appear the same, even though they are actually controlling different appliances.

through any particular aggregate interface, but instead to provide interfaces to meaningful sets of functionality from the appliances in the system for the user's current task.

9.5.1 Active Flow Controls

The Active Flow Controls AUI combines commonly used functions that are related to the currently active flows. Figure 1.8 and Figure 9.4 show examples of active flow control aggregates generated when the active flow is playing a DVD to the receiver and television, controlling a slideshow in a presentation room, copying a tape from one VCR to another, and watching television with the audio coming from the television or receiver speakers.

The PUC identifies functions to be used in the Active Flow Controls AUI in two stages. In the first stage, functions are extracted from the appliance's specifications that are either mentioned in the flow dependencies for the currently active flows, or are noted as being related in the appliance's content flow model. In the second stage, these functions are filtered to select only the most common functions that users will likely want to manipulate. The PUC uses two heuristics in the filtering stage because no information is available in the specification to directly identify the commonly used functions of an appliance.

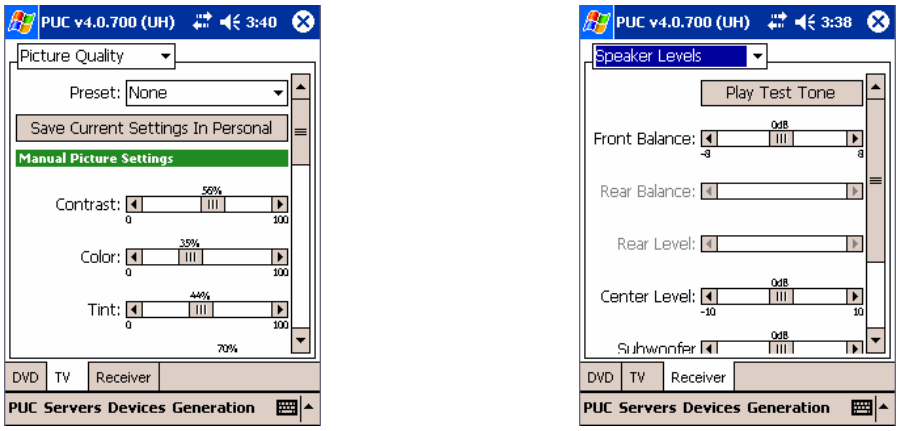
The first heuristic is to eliminate any functions associated with "Setup." Nearly all specifications contain a high-level group with the name "Setup" or something similar. This group will often be identified in the knowledge base used for achieving consistency. The PUC uses

the knowledge base's mapping information to identify the Setup group in each appliance and then filters out any functions that are contained in these Setup groups.

The second heuristic eliminates any functions that, if used or modified, would always cause the flow to stop being active. This eliminates all power functions (which can be easily accessed elsewhere), the input-select variables from the stereo and television, the VCR/TV functions of the VCRs in some situations, and a number of other variables that may be common but would overlap with the functioning of the FBI. The exceptions to this rule are media control functions, such as play, stop, and pause, which are always included. Although the user may deactivate a flow by pressing stop or eject, we feel that users would be annoyed if these functions were not easily available and that users can easily recover if they use these functions in a way that deactivates a flow.

9.5.2 Active Flow Setup

The Active Flow Setup AUI combines setup functions that are related to the currently active flows. The PUC identifies the functions for this aggregate using the first stage of the process used for the Active Flow Controls AUI. Unlike for that earlier aggregate however, the second stage filtering process for this aggregate takes only functions that are found within the "Setup" group. This process typically finds functions that affect the output of the currently active flows but will be used infrequently, such as the brightness and contrast controls for the television and the speaker level controls for the receiver. This AUI still does not include any controls that could inactivate the flow however, since those are best controlled using the FBI.



a. b.

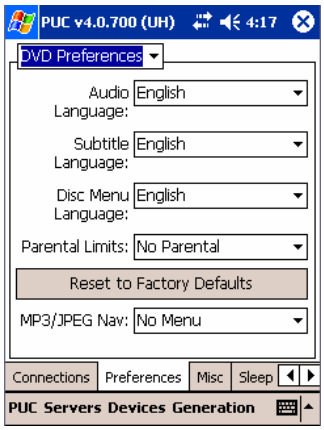
Figure 9.5. Two shots of the Active Flow Setup AUI for the DVD player to receiver and television flow. Note that the interface is organized by appliance, as shown by the tabs at the bottom of the screen.

The Active Flow Setup AUI is organized by appliance, because a desired setup function is typically easier to find with this organization. See Figure 9.5 for two shots of the Active Flow Setup interface generated for the flow for the DVD to receiver and television. We originally tried to organize this aggregate by content type, which in the case of a home theater would give top-level groups for audio and video, but this created lower quality interfaces. The approach worked reasonably for appliances whose functions could be classified by their place in the content flow, such as the receiver and television which in some flows receive only audio and video content respectively. However we found that for appliances which handled both audio and video content that this approach relied too much on the knowledge base's ability to identify sub-groups that corresponded to Audio and Video. It is possible that this approach might be viable with improvement to the consistency sub-system.

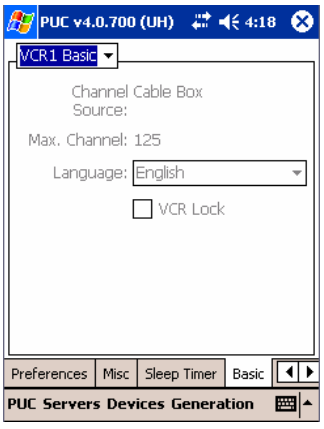
9.5.3 General Setup

The General Setup AUI (see Figure 9.6) combines setup functions across all of the appliances that are not related to any content flow. These functions typically include things such as parental content restrictions, time functions, software upgrade controls, and the configuration of defaults. The functions for the AUI are extracted by iterating over specifications for each of the appliances and eliminating all of the functions that were used in the previous two AUIs. An additional filtering step removes any functions that are not in the "Setup" group.

The General Setup AUI is organized first by any high-level collections of functions that we



a.



b.

Figure 9.6. Two shots of the General Setup AUI for our home theater setup. Note that in both shots, the tabs at the bottom of the screen represent high-level concepts within which the functions are organized by appliance (combo boxes at top).

can identify as existing on more than one appliance, and then by appliance. We first attempt to identify these high-level collections with the knowledge base but also search for any top-level groups within the Setup groups that may have the same name. Groups with the same name may not be identified by Uniform because their contents are not identified as being similar enough. These groups are often catch-all groups such as "Preferences," which have a large variance in the types of functions that they contain.

9.5.4 Merging Controls

There are a few settings across a system of appliances where a single value should be set once and then migrated to all appliances rather than requiring the user to laboriously set the value on each appliance. Examples include the time on the clock, the language (e.g., English), and the sleep timer (that turns off the appliance after a selected number of minutes). Other settings that occur across appliances, however, should not be merged. For example, it is usually wrong to set the channel of the VCRs and television to the same value simultaneously or to set all the devices to be powered on at the same time. Even setup functions cannot always be combined depending on the particular function and how similar the functions are across appliance types. For example, the DVD player and the television both have a contrast setting, but it would be inappropriate to set both of them simultaneously. The Merged Functions AUI handles the small number of functions that are appropriate to combine (see Figure 9.7). As with previous aggregates, the knowledge base is used to identify similar functions across appliances that should be merged.

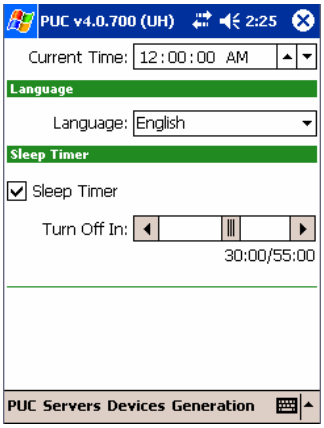


Figure 9.7. The merged function AUI featuring the clock, language, and sleep timer functions on a single panel.

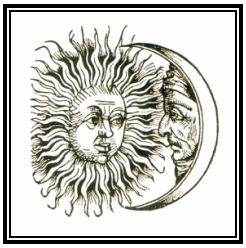
As future work, it could be interesting to explore how the volume function might be merged across appliances using the flat volume technique developed by Baudisch, et al [Baudisch 2004]. While this work has been shown to apply quite well to volume, it is unclear whether it would be applicable for other kinds of settings, such as brightness or contrast.

9.6 Discussion

The most important concept behind the design of the PUC's interface aggregation is its use of a content flow model to help users accomplish their high-level goals. This approach seems to work well for the constrained domains of home theaters and presentation rooms, and I believe that it can be extended to support many more features than we have discussed here. For example, with more detailed modeling of content types, Huddle should not only be able to find content flows for users' goals, but also to find the optimal path for the particular content that the user is viewing. This is a particularly important problem as the types of content within the home theater grow to encompass numerous high-definition video and audio standards which may be supported at varying levels by different appliances and different types of wires.

A problem I have been considering is how content flows can help the PUC understand that the lights should be dimmed in order to view a projected PowerPoint presentation. An extreme solution would be to extend the content flows all the way to the final content sink at the user's eyes and ears, although this would require extensive modeling of each room, its lighting, and the user's perceptual capabilities in order to be successful. A more practical approach may be to provide basic models of which lights and projectors interact, perhaps with a few "environment content sinks" for the most important locations in a room that can allow the PUC to reason about the interactions of appliances within the environment.

The content flow concept should prove extensible to other appliance domains, such as video-conferencing systems and even manufacturing processes. The aggregation approach seems to be limited by the correspondence between the content flows and the tasks that the user wants to perform. Where there is correspondence, such as in the scenarios considered here, this approach works well. One scenario where the PUC may not be effective is in the kitchen, where tasks often center on recipes. It seems that many recipes use the same content flow through appliances, which suggests that the content flow may not be descriptive enough to generate useful task-based interfaces for kitchen appliances.



CHAPTER 10

Usability Evaluation[7]

Two studies of the generated interfaces were conducted to examine the usability of interfaces generated by the PUC. The first study compared the generated interfaces to existing human-designed interfaces for the same functionality, with the hypothesis that interface quality is no longer a limiting factor for automatically generated interfaces. The results were that users of the automatically generated interfaces were *twice as fast* and *four times more successful* than users of the existing interfaces for a set of eight independent tasks with varying difficulty.

The second study examined the PUC's algorithms for automatically generating interfaces that are consistent with the user's previous experience (see Chapters 6 and 8). The hypothesis was that automatically generated interfaces can provide benefits beyond those shown in the first study through user customizations that would be impractical for human designers to provide. In this study, users were first trained on the same eight tasks from the first study using one interface. After users could successfully perform these tasks, they were asked to perform the same tasks on a second, different, interface with similar functionality. The results showed that users were *twice as fast* when the second interface is generated by the PUC to be

[7] The work in this chapter was recently submitted for publication as Jeffrey Nichols, Duen Horng Chau, and Brad A. Myers. "Demonstrating the Viability of Automatically Generated User Interfaces," *Submitted for Publication*.

consistent with the first interface, as compared to when the second interface is generated with the consistency algorithms disabled.

Both user studies compare interfaces for two different all-in-one printer appliances: a Hewlett-Packard (HP) Photosmart 2610 with a high-quality interface including a color LCD and a Canon PIXMA MP780 with a few more features and an interface that turned out to be harder to learn than the HP. These two represented the top-of-the-line consumer models from these manufacturers and the most complex all-in-printers available for home use at the time of their purchase. All-in-one printers were chose as the appliances in these studies for two reasons:

- Complex appliances are typically more difficult to use than trivial ones and I wanted to test the PUC with appliances that would be challenging for its generation algorithms. All-in-printers seem to be at least as complicated, if not more so, than many of the other appliance types that have been explored (containing 85 variables and commands for the HP and 134 for the Canon). The two chosen for these studies have several different main functions, including copying, faxing, scanning, and photo manipulation, that all must be represented in the user interface. They also have many special configuration options for each of the main functions, which make the initial setup process difficult and time-consuming.
- It was not possible for the PUC to actually control the all-in-one printers, but simulating this control was easy to achieve by configuring a computer to print documents on the printers with the correct appearance based on the task the user was currently performing. This resulted in a realistic setting for users of the PUC interfaces, which allows for better comparisons of the PUC interfaces with the existing manufacturers' interfaces.

The existing manufacturers' interfaces from both printers were used for the comparisons conducted in the studies. The PUC-generated interfaces were presented on a Microsoft PocketPC device (see Figure 10.1).

The discussion of the two user studies starts with a description of the interfaces that were compared and the common protocol used for both studies. This is followed by sections presenting and discussing the results for each of the studies.

a. HP printer without consistency

b. Canon printer without consistency

c. HP printer consistent with Canon printer

d. Canon printer consistent with HP printer

Figure 10.1. PocketPC interfaces generated by the Personal Universal Controller (PUC) for the two all-in-one printers discussed in this paper.

10.1 Interfaces

The studies compare PUC-generated interfaces with the manufacturers' human-designed interfaces for the same appliances, and compare PUC-generated interfaces with and without consistency for the two different printers. The manufacturers' interfaces for the two all-one-printers are shown in Figure 10.2.

PUC specifications of both all-in-one printers were needed in order for the PUC to generate interfaces. I wrote the initial specification for the Canon printer and a staff member wrote the initial specification for the HP printer. Different writers were used for the two specifications because these specifications are used for the consistency user study and I wanted the specifications to contain similarities and differences that might be found in a realistic scenario where the specifications were written separately by different manufacturers.

The specifications were also written using an approach that actual specification writers are expected to take. Writers were generally faithful to the design of the actual appliances, but also took advantage of the features of the PUC specification language. For example, the language allows for multiple labels for each function and we added extra labels with further detail where necessary. The PUC language also calls for writers to include as much organizational detail as possible in order to support generation on devices with small screens, and we

b. Canon PIXMA MP780

a. HP Photosmart 2610

Figure 10.2. The all-in-one printers used in our studies, with a larger view of the built-in user interface.

also followed this guideline. The initial specifications were tested with the interface generators to ensure correctness and went through several iterations before they were deemed of high enough quality to be used for the studies. Note that this testing is similar to debugging a program or iteratively testing a user interface and is necessary to ensure that no functions are forgotten, understandable labels are used, etc. The advantage of the PUC system is that these improvements are only needed once and will migrate properly to interfaces generated on any platform.

Note also that both specifications included all of the features of their appliances, even the features not tested. Therefore, the resulting generated user interfaces are *complete* in that they represent all of the features that could be accessed from the appliance's own user interfaces. The specification for the HP consists of 1924 lines of XML containing 85 variables and commands, and the specification for the Canon is 2949 lines of XML containing 134 variables and commands.

The PUC's consistency algorithms also need information about the similarities between specifications (see Chapter 6). An automatic system was used to generate an initial set of mappings between the two all-in-one printer specifications. I then revised the resulting mappings to produce the complete set used in the consistency study.

The two specifications and the mappings between them were then used by the PUC to produce the four different interfaces used in the studies: PUC HP without consistency, PUC Canon without consistency, PUC HP generated to be consistent with the PUC Canon interface, and PUC Canon generated to be consistent with the HP (see Figure 10.1). Combined with the built-in interfaces for the two printers, this results in the six total interfaces used in the studies. Complete screenshots of these interfaces can be viewed in Appendix F.

10.2 Protocol

The subjects who used the PUC interfaces first had a short tutorial on the interface of the PocketPC handheld device. This was necessary because the PUC's design assumes that users will be familiar with the device they are using, and the PocketPC has several interface quirks that can frustrate users who are not aware of them (e.g. the Ok button in dialog boxes is located in the title bar at the top of the screen). Since the intention of the PUC is to work on people's own personal devices, it is reasonable to expect that they will be familiar with the user interface of the device itself.

All subjects performed a block of eight tasks on one of the six interfaces just described. After completing all of the tasks, the subjects received instruction on the quickest method of performing each of the tasks they had just performed. After receiving instruction on a task, subjects were required to perform the task again until they did not make errors. Additional instruction was available for the tasks as needed by the subject. Once the instruction period was completed successfully, the subject performed a second block of the same eight tasks on a different interface for the other all-in-one printer. The goal of this instruction was to make subjects as capable as possible with the first appliance before testing them on a second appliance. This simulated the scenario of a user who is experienced with one appliance encountering a new appliance with similar functionality, which is where we would expect the most impact from the PUC's consistency algorithms.

During both task blocks, users were required to figure out how to perform each task on their own and were *not* provided with a user manual or any other instruction on how to use the printer interfaces. Users were allotted a maximum of 5 minutes to perform each task and were not allowed to move on to the next task until they succeeded or the maximum period was complete. We chose 5 minutes based on several pilot studies that suggested that most subjects would finish within that window or else would never succeed. We recorded the time that it took subjects to complete each task. If a subject did not finish within the allotted period, we recorded his or her completion time as 5 minutes and marked the task as not being completed.

Our protocol has two independent variables: the type of interfaces that a subject used and the order in which the subject used the two all-in-one appliances. Three different configurations of interface type were used in our studies:

- **Built-in:** One built-in interface followed by the other built-in interface (e.g. HP followed by Canon).
- **AutoGen:** PUC interface without consistency for one appliance (e.g., HP) followed by the PUC interface without consistency for the other (e.g., Canon).
- **Consistent AutoGen:** PUC interface without consistency for one appliance (e.g., HP) followed by the PUC interface for the other appliance (e.g., Canon) generated to be consistent with the first interface (e.g., HP).

The Consistent AutoGen configuration is designed to fulfill the assumption of the PUC's consistency algorithms, which assume that users will receive a benefit from consistency when they encounter a new device because they are familiar with a previous interface.

These three configurations allow testing of both usability and consistency. Usability is tested by comparing the Built-in configuration with either of the others. Consistency is tested by comparing the AutoGen and Consistent AutoGen configurations. To test each of these configurations with both of the possible orderings (HP followed by Canon and vice versa) a 3x2 between-subjects study design was used. A within-subjects design is not possible because learning must be carefully controlled to compare performance for both the usability and consistency studies.

10.2.1 Tasks

Eight tasks were designed for subjects to perform during each block of the study. The tasks were chosen to be realistic for an all-in-one printer, cover a wide range of difficulties, and be as independent from each other as possible (so success or failure on one task would not affect subsequent tasks). The last point was especially important to minimize the possibility that a subject might notice an element used in a future task while working on an earlier task. This effect was also minimized somewhat by presenting the next task description only after subjects had completed their previous task; however, this does not prevent subjects working on their second block from remembering the tasks from the first block.

The tasks used, in the order they were always presented to subjects, are listed below. The order of tasks was not varied for each subject so that whatever learning effects might exist between the tasks, despite best efforts to eliminate such effects, would be the same for each subject. The task wording is paraphrased for brevity (exact wording for the tasks can be found in Appendix E):

1. Send a fax to the number stored in the third speed dial.
2. Configure the fax function so that it will always redial a number that is busy.
3. Configure the fax function so that any document received that is larger than the default paper size will be resized to fit the default.
4. Configure the fax function so that it will only print out an error report when it has a problem receiving a fax.

5. Make two black-and-white copies of the document that has already been placed on the scanner of the all-in-one printer.
6. Imagine you find the copies too dark. Improve this by changing one setting of the device.
7. Given a page with a picture, determine how to produce one page with several instances of the same picture repeated.
8. The device remembers the current date and time. Determine where in the interface these values can be changed (but changing them is not required).

The tasks were carefully written to not use language that favored any of the user interfaces being tested. In some cases this was easy because all interfaces used the same terminology. In other cases words were used that did not appear in any of the interfaces. We also used example documents, rather than language, to demonstrate the goal of task 7.

10.3 Participants

Forty-eight subjects, twenty-eight male and twenty female, volunteered for the study through a centralized sign-up web page managed by Carnegie Mellon University. Most subjects were students at either CMU or the University of Pittsburgh and had an average age of 25 and a median age of 23. We also had 3 subjects older than 40 years. Subjects were paid $15 for their time, which varied from about forty minutes to an hour and a half depending on the configuration of interfaces being used. Subjects were randomly assigned to conditions.

10.4 Evaluation of Usability

To evaluate the usability of the PUC interfaces, the task completion times and failures for the Built-in condition were compared with the other two conditions. For this analysis, the data from the first block in each condition is of the most interest because the second block is influenced differently by the subjects' experiences in the first block.

10.4.1 Results

Figure 10.3 shows the average completion time for each of the tasks on each appliance, comparing the Built-In condition with the other two conditions combined (which I will refer to as the PUC condition). Note that data from the AutoGen and Consistent AutoGen condi-

tions can be combined here because the same interfaces are used in the first block of both conditions. To compare completion times and failures in the first block, we conducted several one-way analyses of variance (ANOVAs). For all of these analyses, n=8 in the Built-In condition and n=16 in the PUC condition. Table 10.1 shows the data in more detail with analyses comparing user performance for each task.

On the HP appliance, subjects were significantly faster for total task completion time using the PUC interface ($F_{1,22}$ = 12.11, $p < 0.002$), completing all of the tasks in less than half the time (M=5:54 for the PUC interface vs. M=13:12 for the built-in interface). Subjects also failed significantly less often using the PUC interface ($F_{1,22}$ = 5.69, $p < 0.03$), with a fifth as many failures using the PUC interface as compared to the built-in interface (2 total failures for all users vs. 9).

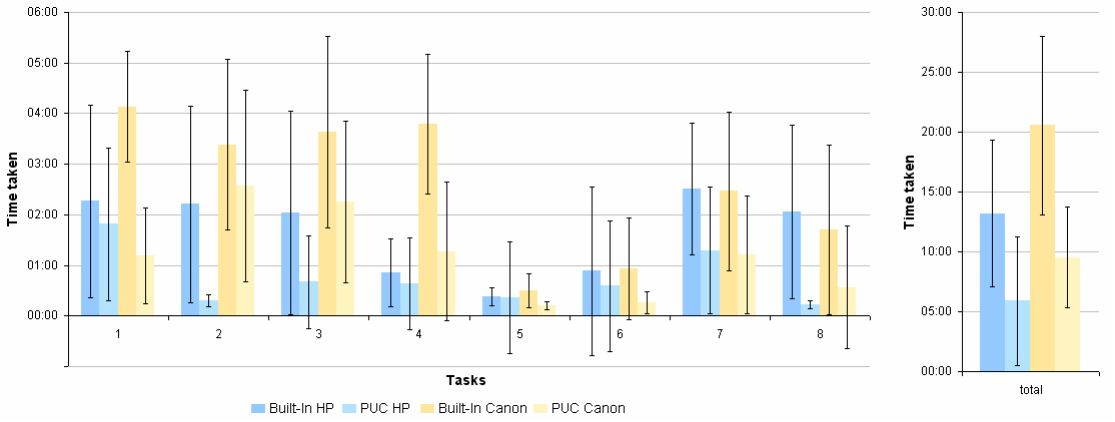


Figure 10.3. Results of the first block of tasks, showing the Built-In condition compared with the other two for each appliance.

			Tasks								
			1	2	3	4	5	6	7	8	Total
Time	HP	Built-In	02:16	02:12*	02:02*	00:51	00:23	00:53	02:31*	02:04*	*13:12**
		PUC	01:49	00:18*	00:40*	00:39	00:22	00:35	01:18*	00:13*	*05:54**
	Canon	Built-In	04:08*	03:23	03:38†	03:48*	00:30*	00:56*	02:28*	01:42†	*20:33**
		PUC	01:12*	02:34	02:15†	01:17*	00:12*	00:16*	01:13*	00:34†	*09:32**
Failures	HP	Built-In	2	2	2	0	0	1	1	1	*9**
		PUC	2	0	0	0	0	0	0	0	*2**
	Canon	Built-In	3*	3	5*	3†	0	0	1	1	*16**
		PUC	0*	5	2*	1†	0	0	1	1	*10**

Table 10.1. Average completion time and total failure data for the first block of tasks. The PUC condition is the combination of the AutoGen and Consistent AutoGen conditions. N = 8 for the Built-In condition and N = 16 for the PUC condition. * indicates a significant difference between the Built-In and PUC conditions for that appliance ($p < 0.05$), and † indicates a marginally significant difference ($p < 0.1$). Completion times and total failures were compared with a one-way analysis variance and failures per task were compared with a one-tailed Fisher's Exact Test.

Subjects overall had more difficulty using the Canon interfaces as compared to the HP interfaces across all conditions ($F_{1,46} = 6.25$, $p < 0.02$), but we still see the same significant benefits for the PUC interface over the built-in interface. Again, subjects were significantly faster using the PUC ($F_{1,22} = 21.88$, $p < 0.001$), with average total completion times of 9:32 for the PUC interface and 20:33 for the built-in interface (again about half the time). Subjects also failed significantly less often using the PUC ($F_{1,22} = 6.57$, $p < 0.02$), with 10 total failures for all users over all tasks using the PUC interface and 16 total failures using the built-in interface (about 1/3 fewer failures on average).

We also performed the same analyses comparing the Built-In condition and the combined PUC condition for the data from the second block of tasks. All of these analyses were significant and matched the results for the first block, except for the number of failures over all tasks for the HP printer. In this case there were too few failures to make this analysis possible: zero failures for all 16 subjects using a PUC HP interface and only one failure for the 8 subjects using the built-in HP interface.

10.4.2 Discussion of Usability

The results show that users perform faster over all eight tasks using the PUC interfaces as compared to the printers' built-in interfaces.

For the Canon printer, the PUC interfaces are significantly faster for nearly all of individual tasks: tasks 3 and 8 are marginally significant and only task 2, automatically re-dialing a busy number, was not found to be different at all.

Task 2 was also the task most failed by users of the PUC interfaces for the Canon by a wide margin. I believe task 2 was particularly hard for users because the Canon printer has many configuration features for sending and receiving faxes, which are complex, seemingly overlap with unrelated functions, and use language that is difficult to understand. These functions were difficult to represent cleanly in the PUC specification language and this may have carried their complexity through to the generated interfaces.

There are fewer individual tasks on the HP printer for which the PUC interface was significantly faster than the built-in interface: only tasks 2, 4, 7, and 8. I believe this is because the HP printer already has a well-designed interface and seemed to perform well, especially for the easier tasks. The tasks where the PUC interfaces excel are generally the more difficult tasks, like tasks 2 and 3, which require the users to find obscure settings deep in the interface.

The five minute maximum completion time was chosen with a goal of limiting failures to between 5-10% of the total tasks. In this data there were 48 subjects performing 8 tasks each for 384 total tasks, and 37 failures were recorded. This gives a 9.6% failure rate, which is high but still within our goal range. Since the time measurements were cut off at 5 minutes, one might worry that this biased the results. However, more than 70% of the failures are found in the Built-In condition. This suggests that our results, which already show the Built-In condition to be slower overall, are likely to be correct since allowing more time would only have made that condition slower.

This study of usability, at least for the first block of tasks, compares the performance of novice users. There is then a question of whether the PUC would be equally successful for expert users. As users become experts, they are less likely to make mistakes, which would probably benefit the harder-to-use Built-In appliance interfaces more than the PUC interfaces. However, fewer steps are required to navigate to and use most functions in the PUC interfaces. Furthermore, the PUC interfaces provide more visual context for the user's current location in the interface. We believe that these features would allow users to become experts with the PUC interface faster than the Built-In interfaces and perform faster once experts. The results of the second study suggest this may be true, as discussed in the next section.

10.5 Evaluation of Consistency

To evaluate consistency, the completion times of interfaces in the AutoGen and Consistent AutoGen conditions are compared for the second block of tasks. We also compare the Built-In condition to the Consistent AutoGen condition, to see how consistency might further improve today's appliance interfaces.

10.5.1 Results

Figure 10.4 shows the average completion times for each task in the second block for the AutoGen and Consistent Auto-Gen conditions. Table 10.2 shows the same data in more detail and includes the Built-In condition and failure data for all the conditions. Again, one-way ANOVAs are used to compare the completion times of the various conditions. Failures are not discussed here because nearly all subjects were able to complete all their tasks in the AutoGen and Consistent AutoGen conditions (results of the analyses of failures are shown in Table 10.2).

Subjects who used the Canon printer and then the HP were significantly faster for total task completion time using the consistent PUC interface compared to the normal PUC interface ($F_{1,14}$ = 10.01, p < 0.007) and the built-in interface ($F_{1,14}$ = 64.48, p < 0.001). The total completion time for the consistent PUC interface was on average more than twice as fast as the normal PUC interface (M=2:10 vs. M=4:54) and more than four times faster than the built-in interface (M=2:10 vs. M=8:55).

After first using the HP printer, subjects were also significantly faster using the consistent PUC interface for the Canon printer, both compared with the normal PUC interface ($F_{1,14}$ = 7.60, p < 0.02) and the built-in interface ($F_{1,14}$ = 16.89, p < 0.002). The average total com-

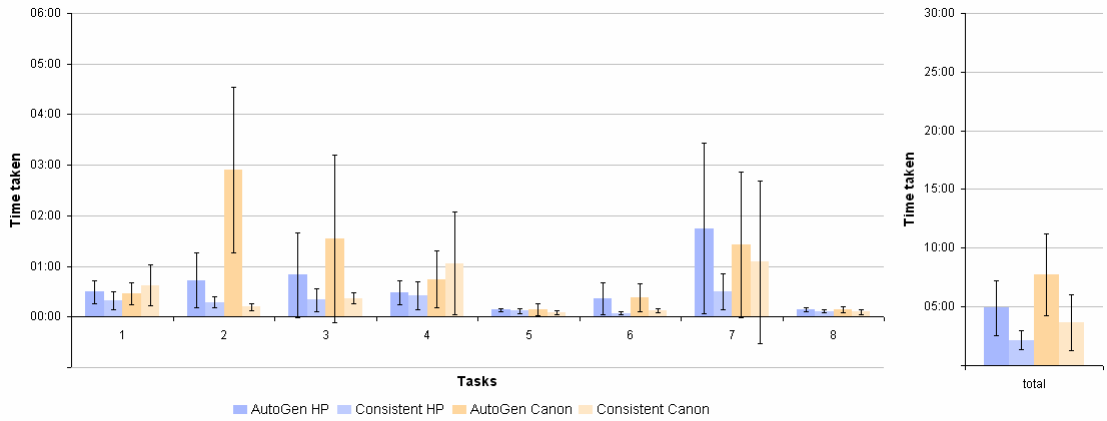


Figure 10.4. Results of the second block of tasks, showing the AutoGen condition compared to the Consistent AutoGen condition for each appliance.

			Tasks 1	2	3	4	5	6	7	8	Total
Time	HP	AutoGen	00:29	00:43*	00:50	00:29	00:08	00:22*	01:45†	00:08	*04:54**
		Consistent	00:20	00:17	00:20	00:25	00:07	00:04	00:30	00:07	*02:10*
		Built-In	01:38*	01:23*	00:37†	00:39†	00:18*	00:16*	03:19*	00:45*	*08:55**
	Canon	AutoGen	00:28	02:54*	01:33†	00:44	00:09	00:23*	01:25	00:09	*07:45**
		Consistent	00:38	00:12	00:22	01:03	00:05	00:08	01:05	00:06	*03:39*
		Built-In	03:15*	02:24*	02:42*	02:14	00:11†	01:42*	02:42†	00:35*	*15:44**
Failures	HP	AutoGen	0	0	0	0	0	0	0	0	*0*
		Consistent	0	0	0	0	0	0	0	0	*0*
		Built-In	0	0	0	0	0	0	1	0	*1*
	Canon	AutoGen	0	2	1	0	0	0	0	0	*3*
		Consistent	0	0	0	0	0	0	1	0	*1*
		Built-In	4*	2	3	2	0	2	2	0	*15**

Table 10.2. Average completion time and total failure data for the second block of tasks. N = 8 for all conditions. * indicates a significant difference between that row's condition and the Consistent AutoGen condition for that appliance (p < 0.05), and † indicates a marginally significant difference (p < 0.1). Completion times and total failures were compared with a one-way analysis variance and failures per task were compared with a one-tailed Fisher's Exact Test.

pletion time for the consistent PUC interface was again more than twice as fast as the normal PUC interface (M=3:39 vs. M=7:45) and more than four times faster on average than the built-in interface (M=3:39 vs. M=15:44).

If the appliance used by subjects is not considered, it is possible to compare the total completion times of the two blocks of tasks for each of the three conditions. The Consistent AutoGen condition is significantly different from the first block to the second ($F_{1,30} = 10.45$, $p < 0.004$). The difference is marginally significant for the Built-In ($F_{1,30} = 3.24$, $p < 0.09$) condition and not significant for the AutoGen ($F_{1,30} = 2.46$, $p < 0.14$) condition.

10.5.2 Discussion of Consistency

The results show that users perform faster over all eight tasks using the consistent interfaces as compared to either of the other interfaces. Much of this effect for both appliances is due to four tasks: 2, 3, 6, and 7. This was expected, because the normal PUC interfaces for these appliances were already consistent for tasks 1 and 8, and thus did not benefit from any change in the consistent interfaces. We had hoped to see consistency effects for the remaining tasks, but other factors seem to have affected tasks 4 and 5.

The change made to ensure consistency for task 5 (copying) involved changing the placement of the copy and cancel buttons on one screen (see Figure 10.1). Apparently the visual search for the new button placement did not affect subjects' speed compared to the normal PUC interfaces.

One change was made to ensure consistency for task 4 (changing the fax error printing). The function needed for this task is located with other fax configuration functions, which are located in different places on the two appliances: in the fax mode on the HP and in the setup section of the Canon interface. The change for consistency performed by the PUC is to move all the configuration functions to the location where the user originally saw them. From observations of subjects' actions, it appeared that this manipulation worked in the studies. Unfortunately, the error reporting function was also different between the two appliances in a way that the PUC's consistency system could not manipulate. When using the HP interface made to be consistent with the Canon interface, users needed time to understand how the functions were different before they could make the correct change. When using the Canon interface consistent with the HP interface, the interface generator made the unfortunate choice of placing the needed functions in a dialog box accessible by pressing a button.

The button to open the dialog was placed next to several other buttons, which distracted subjects from the button they needed to find.

For tasks 2 and 6 we see a significant benefit for consistency for both appliances. Tasks 3 and 7 both have a marginally significant benefit for consistency on just one appliance (task 3 on the HP and task 7 on the Canon). Similar to task 4, both tasks 3 and 7 are slightly different on the two appliances in ways that the PUC's consistency system cannot change. We believe this means that subjects were not able to leverage all of their previous knowledge and had to spend some of their time thinking about how the appliances worked, thus slowing them down.

It is important to note that there are no situations where the PUC's consistency algorithms make the interface significantly worse for users, even for task 4 on the Canon interface generated to be consistent with the HP. The consistency system is able to provide benefits when there are similarities between the appliances and it does not hurt the user when there are differences.

A question to ask is whether the benefits that appear to be from consistency could be due to some other factor in the generation process. I do not believe this is likely, because the rules added for consistent interface generation only make changes to the new interface based on differences with a previous interface that the user has seen. These rules do not perform other modifications that might improve the user interface independent of consistency.

10.6 Discussion

The studies presented here do have some limitations. I used only one type of appliance, all-in-one printers, and only tested two instances of this type. As discussed earlier, I believe that the all-in-one printers we chose are representative of complex appliances as a whole. They also require the use of many of the PUC specification language's most advanced features, such as lists and Smart Templates (see Chapter 7). Although only two all-in-one printers were used, they were carefully chosen to both be complex and representative of different common interface styles. We also chose the HP in part because it had, in my estimation, the best interface of any all-in-one printer available.

These two studies together have shown that the PUC can generate interfaces that exceed the usability of the manufacturers' own interfaces. Using automatic generation to create appliance interfaces allows flexibility in the design of the interface, which allows interfaces to be

modified for each particular user. The consistency feature studied here is one example, and our second study showed that consistency can be beneficial to users. Manufacturers may object to consistency however, because branding may be removed from interfaces and, worse still, branding from a competitor may be added in its place. Our position is that branding which affects the usability of an appliance, such as custom labels for certain functions or particular sets of steps needed to complete particular tasks, is not good for the user and the consistency system should be allowed to modify them. However, branding marks, such as company names, logos, etc., should be preserved appropriately. Support for branding marks and consistency of those marks is a feature that may be added to the PUC system in the future.

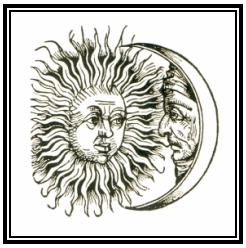
An important question is: what allows the PUC to generate interfaces that are better than the built-in interfaces on the appliances? And what would be needed to improve the built-in interfaces? I believe PUC interfaces are better than the appliance interfaces for many reasons. First, the PUC does not use the same instance of a control for multiple functions. All buttons, sliders, etc. presented in a PUC interface are used for only one function. In contrast, most appliances overload multiple functions on their buttons. For example, both printer interfaces provide a number of multi-purpose buttons on their control panel, including directional pads, ok buttons, and number pads (see Figure 10.2), whose behavior changes depending upon the function selected through the printer's menu. This was a particular problem for the manufacturer's interface on the Canon, which has many modes in which certain buttons cannot be used and for which there is no feedback. Users must experiment to determine which buttons can be pressed in which situations. The PUC addresses the feedback problem by graying-out controls that are not currently available.

The PUC's screen allows for longer and better labels to be shown for each function. The screen also allows for a two-dimensional layout that can give clues to the organization of the interface. For example, the tab control allows users to see immediately that there are multiple groups of controls and what those groups are. Also, the functions displayed in the main portion of any given screen are grouped by functionality, which decreases the number of functions on any one screen and may make the interface easier to parse.

In order to improve the built-in interfaces to the same usability as the PUC, manufacturers would probably need to invest in larger screens for their appliances. These screens would allow the organization of the interface to be clearer, and hopefully eliminate some of the need for multi-purpose buttons. Any physical buttons that are not always functional should have

indicator lights that show when the button can be pressed. Many problems, such as poor labels, could be addressed with basic user-centered iterative design.

The question remains whether it is economical for manufacturers to make these improvements. Screens and indicator lights for buttons could add substantial manufacturing cost to an appliance that already has a low profit margin. Although usability is becoming more of a marketing point, it is still not clear that consumers value it over price except in a few instances (e.g. the iPod). We believe that the PUC could be an excellent solution for appliance manufacturers that currently find themselves in this situation.



CHAPTER 11

Conclusion

The preceding chapters have shown how the PUC system is able to generate user interfaces that are more usable than the interfaces built into today's appliances. The PUC system has also extended interface generation technology to support the design of interfaces that are personally consistent for each individual user, and to use information about content flow to produce aggregate interfaces for systems of multiple appliances. This chapter discusses high-level issues surrounding the PUC system, describes the PUC systems impact to date, reviews the contributions of this dissertation, and describes some possible directions for future work.

11.1 Discussion

The PUC system was designed specifically to produce interfaces for appliances. It could be used to generate other kinds of interfaces but is limited in the scope of the applications that could be supported. Any application that only needs dialog box-style controls in its interfaces could be generated. The PUC may also be able to produce interfaces for applications that need to display large amounts of data using its support for complex data structures, although this capability has not been rigorously tested by the appliances tested so far. Applications that manipulate long text strings, such as e-mail editors, could theoretically be handled by the

PUC framework, though the current interface generators have limited support for rendering large text boxes in an interface.

Some application interfaces cannot be generated by the PUC. In particular, the PUC cannot support generation for applications that must use direct manipulation, such as painting or circuit design applications. Support for direct manipulation was a limitation of many previous model-based systems as well, and those that were able to support direct manipulation required substantial extra modeling effort to provide enough information to allow the interface generator to produce a reasonable interface. These systems required designer involvement in the process as well, to guide the generation or fix any mistakes.

The limitations of the PUC system illustrate the fundamental trade-off between modeling effort and ability of the generator to produce complex interfaces. A highly detailed model may allow an interface generator to produce highly complex interfaces and perhaps even support direct manipulation, but building a model with the necessary amount of detail may require substantial effort. In the design of the PUC system, I have explicitly tried to optimize this trade-off for appliance interfaces. The preliminary user studies allowed me to design a specification language that contains sufficient information to generate a usable interface, but is still concise and easy to use. I could accomplish this because appliance interfaces are a subset of all possible interfaces.

Optimizing the specification language for conciseness and ease of use resulted in not including some features that have become commonplace in other model-based systems. Most significantly, the PUC specification language does not include a task model. For many appliances, task information is not necessary because most of the tasks have only one step. For example, “play the tape” or “increase the volume” involve pressing only one button or sliding one slider. Even more complex tasks, such as programming a timed recording on a VCR, can be rendered understandably by the PUC because there are few steps and few dependencies between the steps. Although some of today’s appliances have interfaces which may require many complicated inter-dependent steps to complete tasks, most of the complication is not inherent to the appliance’s functionality but instead due to the limited interface capabilities of the appliance. The PUC is much less limited in its ability to provide an interface, and as such tasks can be accomplished in the interface without as many complicated steps.

One of the biggest challenges throughout the PUC’s consistency algorithms is appropriately dealing with the unique functions and organization found in similar appliances. Chapter 8

describes three heuristics to address this problem, but the solutions are often limited because of a lack of useful semantic information about the unique functions. In part, this is due to the PUC's use of "relative semantics" to understand similarity; the PUC only knows that two functions are similar, not *why* they are similar, *what* they do, or *how* they relate to other functions in the appliance. With better information, the PUC could make more informed decisions about where to place functions and when to create new organization. Of course, better information would come at the cost of additional modeling for each appliance because it seems unlikely that detailed information could be automatically extracted from the appliance specifications.

A large part of the multi-appliance interface problem discussed in Chapter 9 arises from multiple appliances being connected together, which requires the user to interact with multiple interfaces to accomplish a single task. An obvious solution here is to integrate the appliances into a single monolithic appliance for which an interaction designer can carefully construct a good user interface. In fact, this solution can be seen for some consumer electronics, such as for shelf stereos which integrate an amplifier with a CD player, radio tuner, and other audio devices. The problem with this approach is that it does not allow for expandability and innovation. If all audio appliances had been integrated ten years ago, then today there would be no place for devices like the iPod. The PUC allows users to easily interact with systems of appliances, which enables appliance manufacturers to pursue the design of new appliances that may be added to these systems. There is also no guarantee that the interface produced by the manufacturer for the integrated appliance will be usable, even if the functionality of multiple appliances is integrated into a single monolithic appliance. In fact, the combined appliance interface would probably be harder to design because of the many complicated functions that the integrated appliance would support beyond that of a single appliance. The examples from current systems suggest that manufacturers might not succeed at this difficult design challenge [Brouwer-Janse 1992, Gomes 2003].

For a system like the PUC to succeed as a product in the real world, it is important to ask the following questions:

- Would consumers find value in such a system?
- Would manufacturers support such a system?

For many years, usability has not been an important criterion for consumers when purchasing a new appliance [Brouwer-Janse 1992]. Price and features have been the most important

factors, as well as appearance and brand. For consumers to purchase and use products with PUC technology, usability will need to become a more important factor in buying decisions. I think there is reason to believe that consumers' buying habits may be changing however. The iPod, for example, dominates its market in large part because it is highly usable. Many reviews of technology products, particularly on web sites such a CNET and Engadget.com, routinely evaluate usability along with the usual factors, like price and performance.

Of course, before customers can decide whether or not they would like to buy products that include the PUC, manufacturers will need to include the technology in their products. There are several issues that manufacturers might have with this technology, most importantly the loss of control over their products' interfaces. Manufacturers will at least want to include branding marks in their interfaces and are likely to want some assurance that customers will not attribute fault to the manufacturer if a poor interface generator produces a bad interface for their product. The current PUC system does not have support for including brand marks, though this feature could be added. Unfortunately, there is no obvious solution to the second problem besides educating consumers. My hope is that manufacturers may find this issue less of a concern if consumers begin to demand more usable interfaces and the PUC can offer these interfaces at less cost than hand-designing such interfaces in-house. After all, most PUC devices can provide far better interaction hardware than is cost-effective to put into each appliance.

I also hope that manufacturers will be more willing to adopt PUC technologies if it can be shown that interface generation can offer benefits not possible in the hand-designed interfaces on the actual appliances. In particular, the ability of the PUC to provide improved interfaces automatically for systems of multiple appliances cannot be implemented with non-automatic generation approaches. There also seems to be considerable value in generating interfaces with personal consistency, as shown by the evaluation.

Generation of consistent interfaces is another issue that likely to concern manufacturers however, particularly where branding is concerned. The current version of the PUC's consistency system is quite likely to make changes that would be inappropriate, such as copying a branded label from one appliance interface into an interface for an appliance from a competing brand. For example, the Mitsubishi DVCR appliance has an automatic rewind function that is labeled as "RentalExpress™." Some appliances also have branded interactions, such as Sony CD players which have different behavior for the next track button than most other manufacturers (pressing the next track button on a Sony immediately after starting the player

advances to track 1 instead of track 2 as most others do). If the difference in these interactions can be described in the specification language, then these differences are also likely to be copied inappropriately to a new interface by the consistency algorithms.

My philosophy with branding is that manufacturers should not use branded interactions where they may be detrimental to the usability of the appliance. These situations are exactly those described in the previous section. It is very appropriate for the manufacturers' logos and brand marks to be maintained in a consistent user interface, but using different names or behaviors for functions that also appear on other appliances just confuses users. Of course, manufacturers are unlikely to agree with me and the PUC system should probably be altered to allow certain branded interactions to be preserved by the consistency algorithms. This may be possible simply by allowing markers to be added in the specification that define labels and configurations of state variables and commands that should not be changed by the consistency algorithms.

Cost is another important factor that will affect whether manufacturers adopt a system like the PUC. The PUC requires features like two-way communication, access to all of the functions of the appliance, and support for a more reliable physical communication layer than IR, such as Bluetooth, Wi-Fi, or Zigbee. Some of these features add development cost to a new appliance, such as changing the appliance's software or writing a functional specification. Other features add to the manufacturing cost for each unit, such as adding new hardware to support a new wireless communication protocol. An increase is manufacturing cost is thus substantially worse than an increase in development cost, because any manufacturing cost increases are multiplied by the number of units produced.

Some increases in development cost to support the PUC may not be as substantial as they might seem. For example, most manufacturers already develop some form of functional specification as part of the product development process. While these specifications may not include all of the same kinds of information as a PUC specification, they should provide a good starting point. My intuition is that writing a PUC specification for an appliance would not add substantially to the development cost of an appliance over current industry practices. Many manufacturers are also working to add support for general communication infrastructures, such as UPnP, to their appliances. These technologies have similar server-side requirements as needed for the PUC, such as supporting access to the state of an appliance and an event mechanism to notify external entities when a state has changed. It is possible

that manufacturers' existing implementations for these infrastructures may be easily adaptable to support technologies such as the PUC.

Some of the extra development and manufacturing costs to support the PUC may also be offset by a reduction in other costs. In particular, manufacturers may choose to reduce the complexity of their on-appliance interfaces in favor of making some functions only available through the PUC interface. Such a choice would reduce both the development cost of the physical interface, since developing a simpler interface should cost less than a complex one, and the manufacturing cost of each unit, as fewer buttons and smaller screens are needed for a simpler interface on the physical appliance. However, this cost trade-off may not occur in appliances that are initially deployed with PUC technology, because manufacturers may be unwilling to make certain functionality available only through an interface technology that has not yet achieved broad acceptance.

Unfortunately, it still seems that development and manufaucturing costs must increase in order to build PUC technologies into appliances. Thus, there will need to be other motivating factors for manufacturers justify the extra cost incurred in adding this functionality. A few possible motivations could be increasing the usability of an appliance, adding accessibility of the appliance interface to users with disabilities (the goal of INCITS/V2), or integrating the appliance interface with other appliances that may be connected to it. It remains to be seen whether these issues will motivate manufacturers to adopt standards, such as INCITS/V2, and add these technologies to their appliances.

11.2 Impact

Although the PUC system has not been adopted by any manufacturers of computerized appliances, it has affected the development of the INCITS/V2 standard, been used by two different research groups, and its specification language was used as a component of a project in an undergraduate class at the University of Alabama.

The INCITS/V2 group (described in section 2.1.2.1) is developing a standard for moving the interfaces from appliances to a remote control device, much like the PUC, for the purpose of providing accessible user interfaces for users with physical and cognitive disabilities. The initial draft of the V2 specification did not sufficiently abstract the functionality of the appliance to allow for generation of interfaces on a wide-variety of platforms and did not meet several of the requirements that I found were needed for a system that provides remote

control of appliances (see section 3.6). I provided an initial analysis of the V2 technology [Nichols 2002a] that, along with other interactions, led the V2 standard to adopt many of the design decisions already included in the PUC system. The current version of the standard is much improved though the PUC and V2 still share some differences. More information is available about the similarities and differences between V2 and the PUC system elsewhere [Nichols 2004a].

The PUC system has also been used by two research projects. A group at the Technical University of Vienna, led by Dieter Schmalstieg, has used the PUC in their research into augmented reality applications. In one project, the PUC was used to provide handheld control of augmented reality characters, resulting in a publication at a leading augmented reality conference [Barakonyi 2004]. Figure 11.1 shows an example of the PUC in action with this project.

A group of Italian researchers have also integrated the PUC system into their middleware system, called DomoNet [Miori 2006]. This system allows communication between several of the common device communication protocols, including UPnP and X10. The PUC provides the user interface capability for this middleware, allowing users to control UPnP, X10, and other devices.

Figure 11.1. The PUC being used to control a character in an augmented reality application as part of work performed with the PUC at the Technical University of Vienna. "Tracked PocketPC as a multi-purpose interaction device: (left) Tangible interface in a screenshot of the AR LEGO application (right) PDA screen capture of the LEGO robot's control GUI" [Barakonyi 2004]. Reproduced with permission.

The PUC specification language was also used as a component of a project conducted in the Spring 2005 semester of CS 491: Generative Programming class at the University of Alabama [Sandridge 2005]. The goal of the project was to provide a framework that would allow a mobile application to run on any device, regardless of the device's operating system or interaction technologies. The students initially tried two languages, XUL and Laszlo, but found that both were not abstract enough to describe a user interface in a form that could be rendered differently across multiple platforms. They discovered the PUC language and found that it fit their requirements for producing interfaces. The students produced a rudimentary code generator for J2ME based on the PUC language and work was continued through a summer REU undergraduate research project.

Collectively, these projects show that the PUC system has some value for researchers outside of the PUC research team. Many of these projects show that there is a need for distributing a user interface across different platforms and that the PUC technologies can be used to accomplish this goal. Furthermore, the existence of these projects also suggests that the PUC technologies are also easy enough to learn and use so that people can adopt them successfully with little support from me.

11.3 Contributions

This dissertation describes a complete system that can improve the usability of today's complex computerized appliances by moving the user interface to a handheld device that the user is already carrying. The interfaces are automatically generated to be customized to the controller device, the user, and the particular configuration of appliances that the user has.

The generation process is supported by a specification language that is capable of describing the complete functionality of a wide-range of appliances. The language was tested in an authoring study that showed that authors with no previous experience could learn the language in about 1.5 hours and write a complete specification for a low-cost VCR in about 6 hours.

The dependency information in the specification language can be used by the interface generators to infer some of the structure of the user interface. When the dependencies for multiple sets of functions are found to be mutual exclusive, the interface generator can take special action by placing the sets of functions on separate overlapping panels.

The Smart Templates technique allows domain-specific design conventions to be referenced in appliance specifications and rendered appropriately in the generated interfaces. Templates

are parameterized, allowing the same templates to be used across specifications for different appliances with different functionality. Templates are also described using the primitive elements of the specification language, which allows interface generators to render a template even if the generator is not pre-programmed to understand that template.

To support consistency, a language was developed for describing semantic similarities between appliance specifications. This language describes the similarities in relative terms, e.g. this is the same as that, which is sufficient for ensuring consistency and may be easier to specify and use than other ontology techniques.

The PUC has rules that achieve consistency between generated interfaces. These rules ensure both functional and structural consistency, and make use of heuristics to ensure that the usability of unique functions is not harmed by changes for consistency. Evaluations of these rules showed that users can be twice as fast when using new interfaces that are generated to be consistent as compared to interfaces generated without the consistency rules.

Content flow is a useful model for describing the tasks that users may wish to perform with a system of multiple connected appliances, such as a home theater or presentation room. The Flow-Based Interface allows users to specify and execute their high-level tasks with a multi-appliance system. Four aggregate interface generators were also created that use content flow and combine functionality from multiple appliances to produce useful task-based interfaces.

The PUC graphical interface generators have been implemented on several different platforms, including the PocketPC, Microsoft's Smartphone, and desktop computers. I collaborated with another research group to produce a speech interface generator based on the PUC framework, demonstrating the PUC specification language is sufficiently abstract to create interfaces in multiple modalities.

Finally, the PUC interface generator was evaluated to show that subjects using interfaces produced by the system can be twice as fast and four times as successful as compared to interfaces currently available on today's computerized appliances, thus proving the thesis underlying this work.

11.4 Future Work

There are many directions for future work that build off of this dissertation.

So far I have built graphical interface generators for three different platforms, but there are many more mobile platforms on which interface generation could be implemented. It would be most interesting to build generators for platforms that are substantially different than those I have already looked at. For example, smart phones from most other manufacturers have similar interface styles to the Microsoft Smartphone platform that my interface generator supports. One interesting platform to explore might be wristwatches, which have even more limited interaction capabilities than smart phones.

To support a platform like wristwatches, it might be necessary to build a user interface that supports only the most common functionality of an appliance rather than creating interfaces for the full functionality of an appliance like my current generators do. The challenge is to reliably determine the functions that users will want to use from the full appliance specification. The group tree and the priority information of the specification language already contain information that should help with making this decision though determining the proper threshold for functions to keep versus functions to omit may be difficult to find. In particular, it would unfortunate if the appliance has a set of items that are used together, such as a list and several commands that operate on the list, and not all of the set is included in the generated interface.

It might also be interesting to create an interface generator for web interfaces. Such a generator might be able to integrate ideas from each of the existing generators to adapt the web page layout for the particular device that the web pages will be rendered upon. New "Web 2.0" technologies could also be used to create appliance interfaces on a web page with similar levels of interactivity as that of the interfaces currently generated by the PUC.

While interface generators can be built for many existing platforms, it would also be interesting to build a new device that is specifically designed to support the PUC system. Most of today's high-end universal remote controls with touchscreens also have some number of physical buttons for controlling common functions, such as volume and channel (see the latest Philips Pronto in Figure 2.1), but the PUC is designed to control a broader range of appliances than just home A/V equipment. What physical buttons would be appropriate for a PUC to have? If soft buttons are used, what is the trade-off between assigning functionality to them versus showing the functions in an organized hierarchy as is done by the PUC's current generated interfaces?

More generation rules could be implemented for any of the PUC interface generators. The current rules are sufficient for generating interfaces for the appliances that have been specified and are likely to work for many others, but there is still room for improvement. Rules to improve the generation of interfaces for lists could be especially helpful, as the PUC currently has only a few rules for dealing with the most common types of list structures. A rule that creates a grid for certain two-dimensional list structures is lacking. List data is not supported at all in the current Smartphone interface generator, so there is also a need to port current rules from the PocketPC and develop appropriate controls to differentiate lists of functions, which are the basis of the Smartphone interface, and lists of data stored on the appliance.

Another interesting approach would be to explore mixing the optimization approach for generating interfaces, such as that used by SUPPLE, with the rule-based approach that the PUC currently uses. Numerical optimization algorithms would be useful for converting the abstract user interface into a concrete layout, for example, but might benefit from information from the PUC's mutual exclusion rules for specifying particular organization that an interface must have. Optimization might also be useful for achieving structural consistency between interfaces, though rules may be needed to ensure that unique functions are handled appropriately and predictably. The key for resource-constrained platforms, such as the handheld devices used by the PUC, is to limit the use of optimization to areas where it can provide the most benefit while still providing reasonable performance.

There are several directions of future work to explore with the specification language. Firstly, the authoring study discussed in Chapters 5 and 6 was designed primarily to explore inconsistencies in specifications and not to understand any usability issues that the language might have. More in-depth authoring studies would be valuable to better understand the difficulties that users have when writing specifications and improve the language. Secondly, the current authoring study suggested that it would valuable to create a tool to assist users in creating new specifications. Existing tools for writing XML are helpful for writing specifications, but the language has some properties that are not conducive for direct XML editing. For example, my typical process for writing a specification is to first enumerate all of the state variables and commands and then organize all of the appliance objects into an appropriate group hierarchy. Reorganizing hierarchies is very difficult in XML and might be better supported by a direct manipulation interaction. Object names also change as the organization changes and a tool should manage names automatically.

A third direction to explore with the specification language would be to analyze whether there is any additional information that should be added. The latest version of the INCITS/V2 interface description language has two features that may be useful to add. The INCITS/V2 design separates labeling information from the functional interface description and places the labels in a separate "resource" file. The particular resource file used during interface generation may be changed to support different locales, for example, or perhaps even users with different cognitive abilities. The INCITS/V2 functional specification also includes a "notification" construct, which represents a quick dialog box-style interaction that is displayed when an event occurs on the appliance. The PUC does support text-only alert dialog boxes that are displayed at the request of the appliance, but these dialogs are not prespecificed in the specification language and appear in response to a special `alert-information` messasge sent through the communication protocol. Specifying these messages explicitly would allow the language of the dialog box to be modified by the interface generator. This might be appropriate, for example, if a part of the interface mentioned in the dialog box has been modified by the consistency rules.

The PUC's consistent interface designs are based on seven basic requirements, as discussed in section 6.3. These requirements are based on current work in consistency, but are very high-level and would likely benefit from some elaboration. In particular, it seems that there are several different dimensions along which consistency might be achieved. Lexical consistency requires interfaces to use the same labels for the same functions. Functional and structural consistency, as defined earlier, mean respectively that the same functions use the same controls and functions are located in the same location. A related, but different, dimension to structural consistency is navigational consistency, which requires that users take the same steps to navigate to a function. Navigational consistency implies structural consistency, but, for example, two interfaces could be structurally consistent but not navigationally consistent if they used different controls to navigate between different areas of the interface. There is also visual consistency, which means that interfaces have roughly the same appearance, and there are likely other dimensions as well. Because of differences between appliances, it is often not possible to achieve consistency along all of these dimensions simultaneously. Work is needed to examine the trade-offs between different dimensions and to understand which dimensions are most important for users' productivity. Lexical consistency, for example, would seem to be a constant requirement, but perhaps structural consistency is less important than

visual consistency and the PUC's consistency rules should be adapted to put more emphasis on the visual aspects of consistent interfaces.

It would also be valuable to explore the value of consistency versus the amount of similarity between appliances. My work in this dissertation focuses on achieving consistency for appliances that share many similar functions but not for appliances that share only a few functions. For appliances with few similarities, it seems that the need for consistency rules changes. Functional consistency rules are nearly always helpful, but it seems that structural consistency rules are just as likely to be harmful as helpful. This is because similar functions across mostly different appliances may have a different context of use, which could be disrupted when the function is moved. Furthermore, in my experience with the home theater appliances, I noticed that different appliances sometimes share the same high-level structure (e.g. Control and Setup groups very high in the group hierarchy) but very little of the same low-level hierarchy. In this situation, a moving rule may move a function from deep in the hierarchy to be a child of a very high-level group. Often this will make the moved function more prominent than is appropriate. More work is needed to understand the situations in which structural consistency rules help and hurt, so that this knowledge can be integrated into the consistency rules. One possibility may be to revisit the ideas of sparse, branch, and significant consistency that I have discussed elsewhere [Nichols 2005], but there is still a need for software to be able to reliably detect these types of consistency. In particular, a threshold must be deteremined between the number of similar and unique functions in two groups of different appliance specifications that is sufficient to specify whether the similarality of those groups is sparse or significant.

My consistency algorithms also rely entirely on the similarity information that is contained within the PUC knowledge base. In this work, I have assumed that the knowledge base does not contain any false mappings, which is unlikely if the similarity information is collected by an automated schema mapping algorithm. Work is needed to understand what the impact of errors in the knowledge base are on the current consistency rules and whether the rules can be changed to mitigate any problems that arise. More work is also needed on the mapping algorithms themselves. In particular, it would be interesting to explore whether mapping can be improved by leveraging the existing content in the knowledge base. Use of the knowledge base would provide more information from which to make a match. It is also possible that the knowledge base could contain information about likely sources of false positives in the mapping algorithm, which could be used to filter the generated mappings and reduce errors.

The PUC's consistency algorithms ensure personal consistency, which has an ordering effect: new interfaces are generated based on previous interfaces. If some aspect of the previous interfaces was "bad," then that low quality element might be copied into the new interface. One of the requirements of my consistency algorithms was to support user choice, so that users could control how consistency is applied and choose the best possible interface. The interface to allow users to make these choices has not been implemented yet, however. It is also unclear how users will know that a better interface is available if they never see it because of the consistency algorithms. Additional work is needed to understand how users can be made aware of their options and to build an interface that allows them to make their choice.

Also because interfaces are generated based on previous interfaces, there is an opportunity to seed the consistency system with high quality interfaces for a variety of different common appliance types. Future interfaces for actual appliances would then be modified by the consistency rules to be more like the high quality interfaces, hopefully producing a higher quality result than otherwise. This concept could even be applied to individual functions, ensuring that certain annoying situations, such as unneeded confirmation buttons, never occur in future interfaces. I have not experimented with this idea, and it would be interesting to see how far this technique can be taken to produce improved interfaces.

The PUC currently does not provide any mechanism for end-user customizations. Although our target users are not trained interface designers, they still may wish to modify the interfaces produced by the system. This mechanism could take the form of an interface builder, or might be designed more around the structure of the underlying specification language. Any customizations should be noted by the consistency system, and used to influence the design of future interfaces. It should also be possible for users to create their own macros that automate certain functions of their appliances.

There are two important problems of multi-appliance systems that the PUC does not currently address: helping with the initial wiring of the system and trouble-shooting problems when they occur. Both features could be added to the PUC using Roadie's [Lieberman 2006] approach, which relies on a planning system similar to the PUC's. Some of the wiring problems could also be addressed in a tool that helps users specify the diagram needed to build the PUC's system-wide content flow model. This tool could also help users determine how to best wire their system to support all the flows that they expect to use. It is worth noting that Roadie takes a different approach to configuration, by including wiring instructions in the plans that it generates for each user task as the user is using the system. This ensures that

users are always able to perform a task if it is possible with some configuration of their system, but it seems better to me to perform this kind of analysis at setup time since, at least in my experience, it is unlikely that users will want to rewire their system on a regular basis.

The PUC currently generates four different kinds of aggregate user interfaces, and it would be interesting to explore both improving the existing set and building new kinds of aggregate interfaces. A promising direction is a usage-based aggregate interface, perhaps based on the ideas of Omojokun et al [Omojokun 2006]. A context-sensitive aggregate interface would also be interesting, especially if it could often provide the right function at the right time. Aggregate interfaces based on usage or context are likely to adapt to the user over time however, which may create problems for users if the interface changes in an unexpected way and previously available functions are either removed or hidden. An area that seems promising is producing interface adaptation methods that are high-level and user-driven. This might mean that users give broad or abstract descriptions about the functionality they desire to have in an interface, the system produces the interface, and then the system and user engage in an iterative design process to produce a product that is similar to what the user desires. This process might be driven by a visualization that allows users to see their actual usage and make design suggestions based on it. For example, the user might perform a task and realize that they are likely to perform that task in the future. The user would then open a usage visualization, make a selection corresponding to the task just performed, and then the system would produce an interface specifically for that selection.

I would also like to conduct a formal evaluation of the interfaces generated for systems of multiple appliances. Some informal evaluations with prototypes of the flow-based interface were performed during its design and those results were incorporated into the final design. However, there has been no evaluation of the aggregate interfaces and I cannot be sure that users will perform well with either type of interface. My goal is to allow an inexperienced user to walk into my home theater room and be able to use the system, rather than wait for me to set things up. It may also be interesting to compare the flow-based/aggregate interface designs with the interfaces produced by today's integrated interface products, such as the Philips Pronto or Logitech Harmony universal remotes.

In the final usability study of the PUC interface it was noticed that when users failed they often failed because a control was not where they expected it. When asked to perform a task, the typical process was for users to navigate to the panel where they believed the function was. If the function was not there, then users would begin a heuristic search of the rest of the

interface looking for the function. Because users were not performing an exhaustive search, it was quite common for them to open the panel that contained the function but not find the function because they did not believe that the function would be on that panel. When users were asked why they missed functions in this case, a common response was, "if it's not there, it's not there." This suggests to me that users might benefit from a searchable user interface, where users can provide one or more keywords and view a list of results from the current interface with controls for using the functions embedded in the search result interface. Some tasks might be performed dramatically faster using this approach, assuming the desired control appeared near the top of the search results.

There are a number of issues with this approach however. The biggest is: what if the search rarely returns the result that the user is looking for? In this case, the user may believe the function does not exist. This concern does not worry me however, because the number of functions in an interface is much lower than, for example, the number of pages on the web. The interface search can be liberal about what it returns in its results and there is a fairly good chance that the desired function will appear. Another worry is whether users can easily specify a search query using a handheld device, where text entry is difficult and may be more time-consuming than just searching the interface. This is a definite concern and I hope that it can be addressed by creating query interfaces that can rely in part on selection from a list of appliance-specific words. Another approach would be to leverage the consistency system. For example, the user might open a previous interface, indicate the function they are looking for, and then have the search engine return results in the current interface that are similar. Finally, another worry is the situation in which the user searches for a function that the appliance does not have. For the consistency-based search, this problem is easy to address because the knowledge base can inform the user that the current interface does not contain any similar functions. For other forms of search, this is an issue that will need to be addressed.

There are also many potential benefits to a searchable user interface. For example, instead of searching for a function the user could instead search for a task. If the system could recognize the task and its steps, then it could perform separate searches for each of the task steps and attempt to assemble an aggregate interface for the task based on these searches. Heuristics based on the location of items in the specification might even allow multi-function appliances, such as a combination VCR/DVD recorder, to produce grouped results for each of the appliance's main functions. For example, a "timed record" task search on the multi-

function recorder might produce two results: one for recording a future show onto a VCR tape and another for recording onto a DVD-R.

My dissertation focuses on generating interfaces, but there may some potential to use some of these ideas to evaluate hand-designed user interfaces. For example, a manufacturer might like to compare the hand-designed interface for a new appliance with previous products. Given an abstract representation for these interfaces and information about their similarities, it might be possible to build an evaluation process that would determine where and how the PUC's existing consistency rules would make changes to the new interface based on the previous products. A report based on this information might help the designers understand how the new interface is inconsistent with the previous products and help them make improvements. Other evaluations might examine how a PUC specification for an appliance differs from its hand-designed interface, or identify areas of the interface structure that are particularly deep or difficult to access because of dependencies.

The hurdle to using these techniques to evaluate a hand-designed interface is providing the evaluation system with an abstract model of the interface that it can understand. It seems less than ideal to ask the interface designer to provide this model, as it would require extra work to produce, much of the knowledge needed is already stored implicitly in the hand-designed interface, and it would require the interface designer to learn an abstract modeling language. The last point is particularly important, as most interface designers are not programmers and may not have the training to produce an adequate abstract model of their hand-designed interface. Thus, it seems an important area of future work is developing techniques to infer useful abstract interface models. Ideally, this model could be inferred from an existing concrete interface, but this seems unlikely to succeed because many design decisions that influenced the final design may not be apparent from the final artifact. For example, imagine a panel containing several controls. During the design process, the designer may have chosen to remove a bounding rectangle that grouped a subset of the controls on the panel. While this grouping may not have been necessary in this particular concrete interface, it might be useful for the abstract model to contain information about this extra grouping for use in other analyses. This suggests that it may be interesting to explore automatically inferring abstract interface models from the interface design process.

An abstract model of a hand-designed interface might also enable an automated system to modify hand-designed interfaces to support features that the PUC explored for automatically generated interfaces, like aggregation and consistency. There are a number of challenges to

building a system that automatically modifies existing interfaces. First, it will be important to understand what information is needed about a hand-designed interface in order to make appropriate modifications and how this differs from the information needed to automatically generate a user interface. It is possible that the information needed to modify an interface is smaller than that needed to automatically generate an interface, because a concrete interface is already available to work from. For example, label information would not be needed in the model because it is already contained in the interface. However, the modification algorithms may also need extra information about the concrete interface, such as links that specify which labels go with which controls. A second challenge is to understand how to modify an interface in a way that is consistent with the original hand-design. For example, the interface layout may be based on a grid which should be taken into account when making changes. Thirdly, such a system needs to work with real interfaces that are implemented using current interface toolkits, such as Java Swing or the .NET Framework. This may present many issues. For example, how should an automatic modification system deal with custom controls that are common in many existing interfaces? It would also be useful to understand if there are any features that toolkits might include to help with automatic modification.

The implications of supporting consistency in hand-designed interfaces could be quite interesting, especially for web applications. Today there are many competing web sites, such as MySpace, Orkut, and Friendster, that support basically the same functionality with slightly different user populations. With automatic modification to aggregate similar functionality and ensure consistency, it might be possible to automatically create a mash-up of several similar web sites that provides access to all of the data from each of these web sites through a unified web interface. Understanding the pros and cons of this approach and finding applications that it would benefit is the subject of future work.

In this dissertation, I have explored two ways that automatically generated interfaces may support features that would be impractical to include in hand-designed interfaces: consistency and aggregation. An important area of future work is to find more areas where automatic generation may provide benefits beyond those of hand-designed interfaces. One area in particular that may be of significant value is the automatic generation of interfaces for users with physical or cognitive disabilities. With a model of the user, the automatic interface generator may be able to produce interfaces that are specifically designed to accommodate the user's disabilities. The high variance in disabilities between different users makes this problem particularly challenging, both because a model describing the users capabilities will

need to be quite broad and the rules necessary to generate an interface will need to take into account a wide range of possibilities. Several researchers have recently started examining this direction, but so far there are few results demonstrating success for a wide range of disabled users.

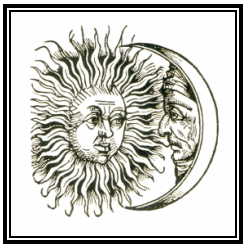
11.5 Final Remarks

This dissertation has attempted to demonstrate the following thesis:

> *A system can automatically generate user interfaces on a wide variety of platforms for remotely controlling appliances where the user's performance is better than with the manufacturer's interfaces for the appliances.*

My evaluations of the PUC system suggest that my interface generators can produce interfaces which are indeed faster than the manufacturers' interfaces for today's computerized appliances.

Going forward, I believe the research described in this dissertation has implications for the future of user interface design and research. For design, it suggests that automatic design should be considered in products where interfaces may be constrained by external factors or individual user customization may have substantial benefits. For research, it suggests that an important direction for future work is developing new techniques that use automatic generation to create interfaces that are customized to each individual.



APPENDIX A

Sample VCR Specification

This appendix contains the full specification for the sample VCR example discussed in Chapter 5.

```
<?xml version="1.0" encoding="utf-8"?>
<spec xmlns="http://www.cs.cmu.edu/~pebbles/puc" name="SimpleVCR" version="PUC/2.3"
      guid="6924EAF4-67E3-431a-BDB9-9FDCE83AC679">
  <labels>
    <label>Simple VCR</label>
  </labels>

  <groupings>
    <group name="Base">
      <state name="Power">
        <apply-type type-name="OnOffType"/>
        <labels>
          <label>Power</label>
        </labels>
      </state>

      <group name="PoweredItems">
        <active-if>
          <equals state="Base.Power">
            <constant value="true"/>
          </equals>
        </active-if>

        <group name="Status" is-a="status-icon-group">
          <state name="TapeIn" access="read-only" is-a="tape-in-status-indicator">
```

```
    <apply-type type-name="YesNoType"/>
    <labels>
      <label>Tape In</label>
    </labels>
  </state>

  <state name="TapeRecordable" access="read-only" is-a="tape-recordable-status-indicator">
    <apply-type type-name="YesNoType"/>
    <labels>
      <label>Tape Recordable</label>
    </labels>
    <active-if>
      <equals state="TapeIn">
        <constant value="true"/>
      </equals>
    </active-if>
  </state>
</group>

<group name="Controls">
  <labels>
    <label>Controls</label>
  </labels>
  <group name="PlayControls" is-a="media-controls">
    <active-if>
      <equals state="TapeIn">
        <constant value="true"/>
      </equals>
    </active-if>
    <labels>
      <label>Play Controls</label>
      <label>Play Mode</label>
      <text-to-speech text="Play Mode" recording="playmode.au"/>
    </labels>

    <state name="Mode">
      <type>
        <enumerated>
          <item-count>6</item-count>
        </enumerated>
        <value-labels>
          <map index="1">
            <labels>
              <label>Stop</label>
            </labels>
          </map>
          <map index="2">
            <labels>
              <label>Play</label>
            </labels>
          </map>
          <map index="3">
            <labels>
              <label>Pause</label>
            </labels>
          </map>
          <map index="4">
            <labels>
              <label>Rewind</label>
```

```
          </labels>
        </map>
        <map index="5">
          <labels>
            <label>Fast-Forward</label>
          </labels>
        </map>
        <map index="6">
          <labels>
            <label>Record</label>
          </labels>
          <active-if>
            <equals state="TapeRecordable">
              <constant value="true"/>
            </equals>
          </active-if>
        </map>
      </value-labels>
    </type>

    <labels>
      <label>Mode</label>
    </labels>
  </state>

  <command name="Eject">
    <labels>
      <label>Eject</label>
    </labels>
  </command>
</group>

<state name="Channel" is-a="channel">
  <type type-name="ChannelType">
    <integer>
      <min>
        <constant value="2"/>
      </min>
      <max>
        <constant value="128"/>
      </max>
    </integer>
  </type>
  <labels>
    <label>Channel</label>
  </labels>
</state>

<state name="TV/VCR">
  <type>
    <boolean/>
    <value-labels>
      <map index="true">
        <labels>
          <label>VCR</label>
        </labels>
      </map>
      <map index="false">
        <labels>
```

```
            <label>TV</label>
          </labels>
        </map>
      </value-labels>
    </type>
    <labels>
      <label>TV/VCR</label>
    </labels>
  </state>
</group>

<group name="TimedRecordings">
  <labels>
    <label>Timed Recordings</label>
  </labels>
  <list-group name="List">
    <labels>
      <label>Timed Recording</label>
    </labels>
    <state name="Channel">
      <apply-type type-name="ChannelType"/>
      <labels>
        <label>Channel</label>
      </labels>
    </state>

    <group name="StartTime" is-a="date-time">
      <labels>
        <label>Start Time</label>
      </labels>
      <state name="Date" is-a="date">
        <type>
          <string/>
        </type>
        <labels>
          <label>Date</label>
        </labels>
      </state>
      <state name="Time" is-a="time">
        <type>
          <string/>
        </type>
        <labels>
          <label>Time</label>
        </labels>
      </state>
    </group>

    <state name="Duration" is-a="time-duration">
      <type>
        <integer/>
      </type>
      <labels>
        <label>Duration</label>
      </labels>
    </state>
  </list-group>

  <group name="Commands" is-a="list-commands">
```

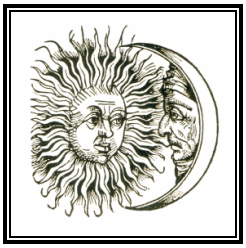
```
          <command name="Add" is-a="list-add">
            <labels>
              <label>Add</label>
            </labels>
          </command>
          <command name="Delete" is-a="list-remove">
            <labels>
              <label>Delete</label>
            </labels>
          </command>
          <command name="Clear" is-a="list-clear">
            <labels>
              <label>Clear All</label>
              <label>Clear</label>
            </labels>
          </command>
        </group>
      </group>
    </group>
  </group>
</groupings>

<ports>
  <inputs>
    <port name="VHF/UHF Antenna" content-type="multi-channel-av" physical-type="coax" />
  </inputs>
  <outputs>
    <port name="VHF/UHF Antenna" content-type="multi-channel-av" physical-type="coax" />
    <port-group name="Output" content-type="av">
      <port name="Video" content-type="video" physical-type="RCA" />
      <port-group name="Audio" content-type="component-audio">
        <port name="Right" content-type="component-audio-right" physical-type="RCA" />
        <port name="Left" content-type="component-audio-left" physical-type="RCA" />
      </port-group>
    </port-group>
  </outputs>
</ports>

<content-flow>
  <pass-through content-type="av">
    <active-if>
      <equals state="Base.Power">
        <constant value="false" />
      </equals>
    </active-if>
    <input-ports>
      <port name="VHF/UHF Antenna" />
    </input-ports>
    <output-ports>
      <port name="VHF/UHF Antenna" />
    </output-ports>
  </pass-through>
  <content-group>
    <active-if>
      <equals state="Base.Power">
        <constant value="true" />
      </equals>
    </active-if>
    <content-group>
```

```
  <active-if>
    <equals state="Base.PoweredItems.Controls.TV/VCR">
      <constant value="true" />
    </equals>
  </active-if>
  <source name="Tape" content-type="av">
    <active-if>
      <equals state="Base.PoweredItems.Status.TapeIn">
        <constant value="true" />
      </equals>
      <not>
        <or>
          <equals state="Base.PoweredItems.Controls.PlayControls.Mode">
            <constant value="10" />
          </equals>
          <equals state="Base.PoweredItems.Controls.PlayControls.Mode">
            <constant value="11" />
          </equals>
        </or>
      </not>
    </active-if>
    <output-ports>
      <port-group name="Output" />
      <port name="VHF/UHF Antenna" channel="3"/>
    </output-ports>
    <objects>
      <group name="Base.PoweredItems.Controls"/>
    </objects>
  </source>
  <pass-through content-type="av">
    <input-ports>
      <port name="VHF/UHF Antenna" />
    </input-ports>
    <processing>
      <block>
        <channel value="3"/>
      </block>
    </processing>
    <output-ports>
      <port name="VHF/UHF Antenna" />
    </output-ports>
  </pass-through>
</content-group>
<pass-through content-type="av">
  <active-if>
    <equals state="Base.PoweredItems.Controls.TV/VCR">
      <constant value="false" />
    </equals>
  </active-if>
  <input-ports>
    <port name="VHF/UHF Antenna" />
  </input-ports>
  <output-ports>
    <port name="VHF/UHF Antenna" />
  </output-ports>
</pass-through>
<recorder name="Tape" content-type="av">
  <active-if>
    <equals state="Base.PoweredItems.Status.TapeIn">
```

```
            <constant value="true"/>
          </equals>
          <equals state="Base.PoweredItems.Status.TapeRecordable">
            <constant value="true"/>
          </equals>
          <equals state="Base.PoweredItems.Controls.PlayControls.Mode">
            <constant value="6"/>
          </equals>
        </active-if>
        <input-ports>
          <port name="VHF/UHF Antenna">
            <channel state="Base.PoweredItems.Controls.Channel"/>
          </port>
        </input-ports>
        <objects>
          <group name="Base.PoweredItems.Controls"/>
        </objects>
      </recorder>
    </content-group>
  </content-flow>
</spec>
```



APPENDIX B

Specification Language Reference

This reference appendix is based on the online documentation for the specification language, which can be found here:

`http://www.pebbles.hcii.cmu.edu/puc/specification.html`

The reference is broken up into three sections. The first is the XML schema that formally defines the language's formatting. The second lists all of the elements with a brief description of their use and the third section describes each element in detail.

B.1 XML Schema

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema targetNamespace="http://www.cs.cmu.edu/~pebbles/puc" elementFormDefault="qualified"
  xmlns="http://www.cs.cmu.edu/~pebbles/puc" xmlns:mstns="http://www.cs.cmu.edu/~pebbles/puc"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!-- Top-Level Element -->
  <xs:element name="spec">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="labels" type="LabelDictionary" />
         <xs:element name="aggregate-spec" type="AggregateSpecTag" minOccurs="0" maxOccurs="1"/>
        <xs:element name="types" type="TypesGroup" minOccurs="0" maxOccurs="1"/>
        <xs:element name="groupings" type="MultipleGroups" />
        <xs:element name="ports" type="PortsGroup" minOccurs="0" maxOccurs="1"/>
        <xs:element name="content-flow" type="ContentFlowGroup" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
```

```
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="version" type="SpecVersionType" use="required"/>
    <xs:attribute name="guid" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>
<!-- Attribute Types -->
<xs:simpleType name="SpecVersionType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="PUC/2.3" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="IgnoreType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="none" />
    <xs:enumeration value="parent" />
    <xs:enumeration value="all" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="AccessType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="read-only" />
    <xs:enumeration value="read-write" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="TypeNameType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="one" />
    <xs:enumeration value="multiple" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="PriorityType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0" />
    <xs:maxInclusive value="10" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="TrueIfType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="any" />
    <xs:enumeration value="all" />
    <xs:enumeration value="none" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="ItemsType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="all" />
    <xs:enumeration value="selected" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="ContentTypeAttrib">
  <xs:restriction base="xs:string">
    <xs:enumeration value="audio" />
    <xs:enumeration value="component-audio"/>
    <xs:enumeration value="video" />
    <xs:enumeration value="component-video"/>
    <xs:enumeration value="av" />
    <xs:enumeration value="multi-channel-audio" />
    <xs:enumeration value="multi-channel-video" />
    <xs:enumeration value="multi-channel-av" />
```

```
  </xs:restriction>
</xs:simpleType>
<!-- Elements -->
<xs:complexType name="LabelDictionary">
  <xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element name="label" type="xs:string" minOccurs="1" />
      <xs:element name="ref-value" type="stateAttribNoContent" />
      <xs:element name="phonetic" type="xs:string" />
      <xs:element name="text-to-speech" type="TextToSpeechType" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="stateAttribNoContent">
  <xs:attribute name="state" type="xs:string" use="required" />
</xs:complexType>
<xs:complexType name="TextToSpeechType">
  <xs:attribute name="text" type="xs:string" use="required" />
  <xs:attribute name="recording" type="xs:string" />
</xs:complexType>
<!-- Aggregate Spec Types -->
<xs:complexType name="AggregateSpecTag">
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="ApplianceReference">
  <xs:attribute name="appliance" type="xs:string" use="required"/>
  <xs:attribute name="object" type="xs:string" use="required"/>
</xs:complexType>
<!-- Types Section -->
<xs:complexType name="TypesGroup">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="group" type="TypesGroupType" />
    <xs:element name="list-group" type="TypesListGroupType" />
    <xs:element name="union-group" type="TypesUnionGroupType" />
    <xs:element name="state" type="TypesStateType" />
    <xs:element name="command" type="TypesCommandType" />
    <xs:element name="explanation" type="TypesCommandType" />
    <xs:element name="type" type="TypesPrimitiveType" />
  </xs:choice>
</xs:complexType>
<xs:attributeGroup name="TypesObjectAttribs">
  <xs:attribute name="type-name" type="xs:string" use="required" />
  <xs:attribute name="is-a" type="xs:string" use="optional" />
  <xs:attribute name="priority" type="PriorityType" use="optional" />
</xs:attributeGroup>
<xs:complexType name="TypesGroupType">
  <xs:sequence>
    <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
    <xs:element name="labels" type="LabelDictionary" minOccurs="0" maxOccurs="1" />
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element name="group" type="GroupType" />
      <xs:element name="list-group" type="ListGroupType" />
      <xs:element name="union-group" type="UnionGroupType" />
      <xs:element name="state" type="StateType" />
      <xs:element name="command" type="CommandType" />
      <xs:element name="explanation" type="CommandType" />
      <xs:element name="apply-type" type="ObjectApplyType" />
    </xs:choice>
  </xs:sequence>
```

```
  <xs:attributeGroup ref="TypesObjectAttribs" />
</xs:complexType>
<xs:complexType name="TypesUnionGroupType">
  <xs:sequence>
    <xs:element name="labels" type="LabelDictionary" minOccurs="0" maxOccurs="1" />
    <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element name="group" type="GroupType" />
      <xs:element name="list-group" type="ListGroupType" />
      <xs:element name="union-group" type="UnionGroupType" />
      <xs:element name="state" type="StateType" />
      <xs:element name="command" type="CommandType" />
      <xs:element name="explanation" type="CommandType" />
      <xs:element name="apply-type" type="ObjectApplyType" />
    </xs:choice>
  </xs:sequence>
  <xs:attributeGroup ref="TypesObjectAttribs" />
  <xs:attribute name="access" type="AccessType" />
</xs:complexType>
<xs:complexType name="TypesListGroupType">
  <xs:sequence>
    <xs:element name="labels" type="LabelDictionary" minOccurs="0" maxOccurs="1" />
    <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:choice minOccurs="0" maxOccurs="1">
        <xs:sequence>
          <xs:element name="min" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
          <xs:element name="max" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
        </xs:sequence>
        <xs:element name="item-count" type="xs:integer" />
      </xs:choice>
      <xs:element name="selections" type="SelectionTypeType" minOccurs="0" maxOccurs="1" />
      <xs:element name="group" type="GroupType" />
      <xs:element name="list-group" type="ListGroupType" />
      <xs:element name="union-group" type="UnionGroupType" />
      <xs:element name="state" type="StateType" />
      <xs:element name="command" type="CommandType" />
      <xs:element name="explanation" type="CommandType" />
      <xs:element name="apply-type" type="ObjectApplyType" />
    </xs:choice>
  </xs:sequence>
  <xs:attributeGroup ref="TypesObjectAttribs" />
</xs:complexType>
<xs:complexType name="TypesCommandType">
  <xs:sequence>
    <xs:element name="labels" type="LabelDictionary" />
    <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attributeGroup ref="TypesObjectAttribs" />
</xs:complexType>
<xs:complexType name="TypesStateType">
  <xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:element name="type" type="PrimitiveType" />
      <xs:element name="apply-type" type="ApplyPrimitiveType" />
    </xs:choice>
    <xs:element name="labels" type="LabelDictionary" minOccurs="0" maxOccurs="1" />
    <xs:element name="active-if" type="DependencyContent" minOccurs="0" maxOccurs="1" />
    <xs:element name="required-if" type="DependencyContent" minOccurs="0" maxOccurs="1"/>
```

```
    <xs:element name="default-value" type="StaticOrReference"/>
  </xs:sequence>
  <xs:attributeGroup ref="TypesObjectAttribs" />
  <xs:attribute name="access" type="AccessType" use="optional" />
</xs:complexType>
<xs:complexType name="TypesPrimitiveType">
  <xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:element name="binary" type="BinaryType" />
      <xs:element name="boolean" />
      <xs:element name="enumerated" type="EnumeratedType" />
      <xs:element name="fixedpt" type="FixedPtType" />
      <xs:element name="floatingpt" type="FloatingPtType" />
      <xs:element name="integer" type="IntegerType" />
      <xs:element name="list-selection" type="ListSelectionType" />
      <xs:element name="string" type="StringType" />
    </xs:choice>
    <xs:element name="value-labels" type="ValueLabelsType" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attribute name="type-name" type="xs:string" use="required" />
</xs:complexType>
<!-- Groupings Section -->
<xs:complexType name="MultipleGroups">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="group" type="GroupType" />
    <xs:element name="list-group" type="ListGroupType" />
    <xs:element name="union-group" type="UnionGroupType" />
    <xs:element name="apply-type" type="ObjectApplyType" />
  </xs:choice>
</xs:complexType>
<xs:complexType name="GroupType">
  <xs:sequence>
    <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
    <xs:element name="labels" type="LabelDictionary" minOccurs="0" maxOccurs="1" />
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element name="group" type="GroupType" />
      <xs:element name="list-group" type="ListGroupType" />
      <xs:element name="union-group" type="UnionGroupType" />
      <xs:element name="state" type="StateType" />
      <xs:element name="command" type="CommandType" />
      <xs:element name="explanation" type="ExplanationType" />
      <xs:element name="apply-type" type="ObjectApplyType" />
    </xs:choice>
  </xs:sequence>
  <xs:attributeGroup ref="ApplianceObjectAttribs" />
</xs:complexType>
<xs:attributeGroup name="ApplianceObjectAttribs">
  <xs:attribute name="name" type="xs:string" use="required" />
  <xs:attribute name="type-name" type="xs:string" use="optional" />
  <xs:attribute name="is-a" type="xs:string" use="optional" />
  <xs:attribute name="priority" type="PriorityType" use="optional" />
</xs:attributeGroup>
<xs:complexType name="UnionGroupType">
  <xs:sequence>
     <xs:element name="appliance-reference" type="ApplianceReference" minOccurs="0" maxOccurs="1"/>
    <xs:element name="labels" type="LabelDictionary" minOccurs="0" maxOccurs="1" />
    <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element name="group" type="GroupType" />
```

```
      <xs:element name="list-group" type="ListGroupType" />
      <xs:element name="union-group" type="UnionGroupType" />
      <xs:element name="state" type="StateType" />
      <xs:element name="command" type="CommandType" />
      <xs:element name="explanation" type="ExplanationType" />
      <xs:element name="apply-type" type="ObjectApplyType" />
    </xs:choice>
  </xs:sequence>
  <xs:attributeGroup ref="ApplianceObjectAttribs" />
  <xs:attribute name="access" type="AccessType" />
</xs:complexType>
<xs:complexType name="ListGroupType">
  <xs:sequence>
    <xs:element name="appliance-reference" type="ApplianceReference" minOccurs="0" maxOccurs="1"/>
    <xs:element name="labels" type="LabelDictionary" minOccurs="0" maxOccurs="1" />
   <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
   <xs:choice minOccurs="1" maxOccurs="unbounded">
     <xs:choice minOccurs="0" maxOccurs="1">
       <xs:sequence>
         <xs:element name="min" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
         <xs:element name="max" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
       </xs:sequence>
       <xs:element name="item-count" type="xs:integer" />
     </xs:choice>
     <xs:element name="selections" type="SelectionTypeType" minOccurs="0" maxOccurs="1" />
     <xs:element name="sortable" minOccurs="0" maxOccurs="1" />
     <xs:element name="group" type="GroupType" />
     <xs:element name="list-group" type="ListGroupType" />
     <xs:element name="union-group" type="UnionGroupType" />
     <xs:element name="state" type="StateType" />
     <xs:element name="command" type="CommandType" />
     <xs:element name="explanation" type="ExplanationType" />
     <xs:element name="apply-type" type="ObjectApplyType" />
   </xs:choice>
  </xs:sequence>
  <xs:attributeGroup ref="ApplianceObjectAttribs" />
</xs:complexType>
<xs:complexType name="StaticOrReference">
  <xs:choice>
    <xs:element name="constant" type="valueAttribNoContent" />
    <xs:element name="ref-value" type="stateAttribNoContent" />
  </xs:choice>
</xs:complexType>
<xs:complexType name="valueAttribNoContent">
  <xs:attribute name="value" type="xs:string" use="required" />
</xs:complexType>
<xs:complexType name="SelectionTypeType">
  <xs:attribute name="number" type="TypeNameType" use="required" />
  <xs:attribute name="access" type="AccessType" use="optional" default="read-write" />
</xs:complexType>
<xs:complexType name="ObjectApplyType">
  <xs:attribute name="type-name" type="xs:string" use="required"/>
  <xs:attribute name="name" type="xs:string" use="optional" />
  <xs:attribute name="priority" type="PriorityType" use="optional" />
  <xs:attribute name="access" type="AccessType" use="optional" />
</xs:complexType>
<xs:complexType name="ModifiesStateType">
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
```

```
<xs:complexType name="CommandType">
  <xs:sequence>
    <xs:element name="appliance-reference" type="ApplianceReference" minOccurs="0" maxOccurs="1"/>
    <xs:element name="modifies-state" type="ModifiesStateType" minOccurs="0" maxOccurs="1"/>
    <xs:element name="labels" type="LabelDictionary" />
   <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attributeGroup ref="ApplianceObjectAttribs" />
</xs:complexType>
<xs:complexType name="ExplanationType">
  <xs:sequence>
    <xs:element name="appliance-reference" type="ApplianceReference" minOccurs="0" maxOccurs="1"/>
    <xs:element name="labels" type="LabelDictionary" />
    <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attributeGroup ref="ApplianceObjectAttribs" />
</xs:complexType>
<xs:complexType name="StateType">
  <xs:sequence>
    <xs:element name="appliance-reference" type="ApplianceReference" minOccurs="0" maxOccurs="1"/>
    <xs:choice minOccurs="1" maxOccurs="1">
     <xs:element name="type" type="PrimitiveType" />
     <xs:element name="apply-type" type="ApplyPrimitiveType" />
   </xs:choice>
   <xs:element name="labels" type="LabelDictionary" minOccurs="0" maxOccurs="1" />
   <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
   <xs:element name="required-if" type="DependencyContent" minOccurs="0" maxOccurs="1"/>
   <xs:element name="default-value" type="StaticOrReference" minOccurs="0" maxOccurs="1"/>
   <xs:element name="completions-available" minOccurs="0" maxOccurs="1" />
   <xs:element name="server-side-error-correction" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attributeGroup ref="ApplianceObjectAttribs" />
  <xs:attribute name="access" type="AccessType" use="optional" />
</xs:complexType>
<xs:complexType name="ApplyPrimitiveType">
  <xs:attribute name="type-name" type="xs:string" use="required" />
</xs:complexType>
<xs:complexType name="PrimitiveType">
  <xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:element name="binary" type="BinaryType" />
      <xs:element name="boolean" />
      <xs:element name="enumerated" type="EnumeratedType" />
      <xs:element name="fixedpt" type="FixedPtType" />
      <xs:element name="floatingpt" type="FloatingPtType" />
      <xs:element name="integer" type="IntegerType" />
      <xs:element name="list-selection" type="ListSelectionType" />
      <xs:element name="string" type="StringType" />
    </xs:choice>
    <xs:element name="value-labels" type="ValueLabelsType" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attribute name="type-name" type="xs:string" use="optional" />
</xs:complexType>
<xs:complexType name="ValueLabelsType">
  <xs:sequence>
    <xs:element name="map" type="MapType" minOccurs="1" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="MapType">
```

```
  <xs:all>
    <xs:element name="labels" type="LabelDictionary" minOccurs="1" maxOccurs="1" />
    <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
  </xs:all>
  <xs:attribute name="index" type="xs:string" />
</xs:complexType>
<xs:complexType name="StringType">
  <xs:all>
    <xs:element name="min" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
    <xs:element name="max" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
    <xs:element name="average" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
  </xs:all>
</xs:complexType>
<xs:complexType name="ListSelectionType">
  <xs:sequence>
    <xs:element name="active-if" type="ActiveIfType" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attribute name="list" type="xs:string" />
</xs:complexType>
<xs:complexType name="FloatingPtType">
  <xs:sequence>
    <xs:element name="min" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
    <xs:element name="max" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
     <xs:element name="specific-values-important" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="IntegerType">
  <xs:sequence>
    <xs:element name="min" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
    <xs:element name="max" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
    <xs:element name="incr" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
     <xs:element name="specific-values-important" minOccurs="0" maxOccurs="1" />
   </xs:sequence>
</xs:complexType>
<xs:complexType name="FixedPtType">
  <xs:sequence>
    <xs:element name="pointpos" type="xs:integer" minOccurs="1" maxOccurs="1" />
    <xs:element name="min" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
    <xs:element name="max" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
    <xs:element name="incr" type="StaticOrReference" minOccurs="0" maxOccurs="1" />
     <xs:element name="specific-values-important" minOccurs="0" maxOccurs="1" />
   </xs:sequence>
</xs:complexType>
<xs:complexType name="EnumeratedType">
  <xs:sequence>
    <xs:element name="item-count" type="xs:integer" minOccurs="1" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="BinaryType">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ActiveIfType">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="or" type="DependencyContent" />
    <xs:element name="and" type="DependencyContent" />
    <xs:element name="not" type="NotType" />
    <xs:element name="apply-over" type="ApplyOverType" />
```

```
    <xs:element name="defined" type="stateAttribNoContent" />
    <xs:element name="undefined" type="stateAttribNoContent" />
    <xs:element name="equals" type="ValueDependencyType" />
    <xs:element name="greaterthan" type="ValueDependencyType" />
    <xs:element name="lessthan" type="ValueDependencyType" />
    <xs:element name="true" />
    <xs:element name="false" />
  </xs:choice>
  <xs:attribute name="ignore" type="IgnoreType" use="optional" />
</xs:complexType>
<xs:complexType name="DependencyContent">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="or" type="DependencyContent" />
    <xs:element name="and" type="DependencyContent" />
    <xs:element name="not" type="NotType" />
    <xs:element name="apply-over" type="ApplyOverType" />
    <xs:element name="defined" type="stateAttribNoContent" />
    <xs:element name="undefined" type="stateAttribNoContent" />
    <xs:element name="equals" type="ValueDependencyType" />
    <xs:element name="greaterthan" type="ValueDependencyType" />
    <xs:element name="lessthan" type="ValueDependencyType" />
    <xs:element name="true" />
    <xs:element name="false" />
  </xs:choice>
</xs:complexType>
<xs:complexType name="ValueDependencyType">
  <xs:choice>
    <xs:element name="constant" type="valueAttribNoContent" />
    <xs:element name="ref-value" type="stateAttribNoContent" />
  </xs:choice>
  <xs:attribute name="state" type="xs:string" use="required" />
</xs:complexType>
<xs:complexType name="NotType">
  <xs:choice minOccurs="1" maxOccurs="1">
    <xs:element name="or" type="DependencyContent" />
    <xs:element name="and" type="DependencyContent" />
    <xs:element name="apply-over" type="ApplyOverType" />
    <xs:element name="defined" type="stateAttribNoContent" />
    <xs:element name="undefined" type="stateAttribNoContent" />
    <xs:element name="equals" type="ValueDependencyType" />
    <xs:element name="greaterthan" type="ValueDependencyType" />
    <xs:element name="lessthan" type="ValueDependencyType" />
  </xs:choice>
</xs:complexType>
<xs:complexType name="ApplyOverType">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="or" type="DependencyContent" />
    <xs:element name="and" type="DependencyContent" />
    <xs:element name="not" type="NotType" />
    <xs:element name="apply-over" type="ApplyOverType" />
    <xs:element name="defined" type="stateAttribNoContent" />
    <xs:element name="undefined" type="stateAttribNoContent" />
    <xs:element name="equals" type="ValueDependencyType" />
    <xs:element name="greaterthan" type="ValueDependencyType" />
    <xs:element name="lessthan" type="ValueDependencyType" />
  </xs:choice>
  <xs:attribute name="list" type="xs:string" use="required" />
  <xs:attribute name="items" type="ItemsType" use="optional" />
  <xs:attribute name="true-if" type="TrueIfType" use="optional" />
```

```
</xs:complexType>
<xs:complexType name="PortsGroup">
  <xs:sequence>
    <xs:element name="inputs" type="PortsType" minOccurs="0" maxOccurs="1"/>
    <xs:element name="outputs" type="PortsType" minOccurs="0" maxOccurs="1"/>
    <xs:element name="input-output" type="PortsType" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="PortsType">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="port" type="PortType"/>
    <xs:element name="port-group" type="PortGroupType"/>
  </xs:choice>
</xs:complexType>
<xs:complexType name="PortType">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="content-type" type="ContentTypeAttrib" use="required"/>
  <xs:attribute name="physical-type" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="PortGroupType">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="port" type="PortType"/>
    <xs:element name="port-group" type="PortGroupType"/>
  </xs:choice>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="content-type" type="ContentTypeAttrib" use="required"/>
</xs:complexType>
<xs:complexType name="ContentFlowGroup">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="content-group" type="ContentGroupType"/>
    <xs:element name="source" type="SourceType"/>
    <xs:element name="renderer" type="SinkType"/>
    <xs:element name="recorder" type="SinkType"/>
    <xs:element name="pass-through" type="PassThroughType"/>
  </xs:choice>
</xs:complexType>
<xs:complexType name="ContentGroupType">
  <xs:sequence>
    <xs:element name="active-if" type="ActiveIfType" minOccurs="1" maxOccurs="1"/>
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element name="content-group" type="ContentGroupType"/>
      <xs:element name="source" type="SourceType"/>
      <xs:element name="renderer" type="SinkType"/>
      <xs:element name="recorder" type="SinkType"/>
      <xs:element name="pass-through" type="PassThroughType"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="SourceType">
  <xs:sequence>
    <xs:element name="active-if" type="DependencyContent" minOccurs="0" maxOccurs="1"/>
    <xs:element name="output-ports" type="ContentPortType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="objects" type="ContentStatesType" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="content-type" type="ContentTypeAttrib" use="required"/>
</xs:complexType>
<xs:complexType name="SinkType">
  <xs:sequence>
```

```
    <xs:element name="active-if" type="DependencyContent" minOccurs="0" maxOccurs="1"/>
    <xs:element name="input-ports" type="ContentPortType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="objects" type="ContentStatesType" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="content-type" type="ContentTypeAttrib" use="required"/>
</xs:complexType>
<xs:complexType name="PassThroughType">
  <xs:sequence>
    <xs:element name="active-if" type="DependencyContent" minOccurs="0" maxOccurs="1"/>
    <xs:element name="input-ports" type="ContentPortType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="processing" type="ProcessingType" minOccurs="0" maxOccurs="1"/>
    <xs:element name="output-ports" type="ContentPortType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="objects" type="ContentStatesType" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="optional"/>
  <xs:attribute name="content-type" type="ContentTypeAttrib" use="required"/>
</xs:complexType>
<xs:complexType name="ContentStatesType">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="object" type="ContentObjectType"/>
     <xs:element name="group" type="ContentObjectType"/>
  </xs:choice>
</xs:complexType>
 <xs:complexType name="ContentObjectType">
   <xs:attribute name="name" type="xs:string"/>
</xs:complexType>
<xs:complexType name="ContentPortType">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="port" type="ContentPortParamType"/>
    <xs:element name="port-group" type="ContentPortParamType"/>
  </xs:choice>
</xs:complexType>
<xs:complexType name="ContentPortParamType">
  <xs:sequence>
    <xs:element name="active-if" type="DependencyContent" minOccurs="0" maxOccurs="1"/>
    <xs:element name="channel" type="ChannelType" minOccurs="0" maxOccurs="1"/>
     <xs:element name="objects" type="ContentStatesType" minOccurs="0" maxOccurs="1"/>
   </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="channel" type="xs:string" use="optional"/>
</xs:complexType>
<xs:complexType name="ChannelType">
  <xs:attribute name="state" type="xs:string" use="optional"/>
  <xs:attribute name="value" type="xs:string" use="optional"/>
</xs:complexType>
<xs:complexType name="ProcessingType">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="block" type="BlockProcessingType"/>
  </xs:choice>
</xs:complexType>
<xs:complexType name="BlockProcessingType">
  <xs:sequence>
    <xs:element name="active-if" type="DependencyContent" minOccurs="0" maxOccurs="1"/>
    <xs:element name="channel" type="ChannelType" minOccurs="0" maxOccurs="1"/>
     <xs:element name="port" type="BlockPortType" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="channel" type="xs:string" use="optional"/>
   <xs:attribute name="port" type="xs:string" use="optional"/>
```

```
  </xs:complexType>
  <xs:complexType name="BlockPortType">
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

B.2 Element Index

Element Name	Description
<active-if>	Contains dependency information for an appliance object or a group of objects. Defines an AND relation with all dependencies that are contained within, unless they are grouped within a logical operation block, such as <or>.
<and>	Defines an AND relation with the dependencies that are contained within.
<apply-over>	Applies dependency relations to a list of data.
<apply-type>	Allows the re-use of an existing type block within a specification.
<binary>	Binary type - encompasses any kind of binary data, including images and sounds. A Smart Template must be used to appropriately interpret and render the binary data.
<block>	Specifies that a channel should be blocked by an appliance that is passing through a multi-channel content stream.
<boolean>	Boolean type - takes on true or false values.
<channel>	Used in the content flow descriptions to define a particular channel that content may flow over.
<command>	Defines a command appliance object.
<completions-available>	Specifies that this state variable has completions available from the server.
<constant>	Defines a constant value in any place where a reference would also be accepted.
<content-flow>	Defines the section that describes the internal content flows of an appliance.

Element Name	Description
<content-group>	Groups two or more content flows for the purpose of specifying dependencies that apply to both.
<default-value>	Specifies a default value for a state variable. In an interface generator, these values would be used when the UI needs to prompt the user for a new value and could also be used to demo an offline interface.
<defined>	Used in conjunction with the <active-if> element to define a dependency that a state variable must have some value.
<enumerated>	Enumerated type - Define the number of items in the enumeration using the <item-count> tag. Labels can be defined within the <value-labels> tag.
<equals>	Used in conjunction with the <active-if> tag to define equals dependency information for this state variable.
<explanation>	Defines an explanation appliance object.
<false>	False - for use in dependency expressions.
<fixedpt>	Fixed Point type - for variables that take the form of decimal values with a fixed decimal point location. Minimum, maximum and increment values can be defined using the <min>, <max>, and <incr> tags.
<floatingpt>	Floating Point type - for variables that take the form of decimal values. Minimum and maximum values can be defined using the <min> and <max> tags.
<greaterthan>	Used in conjunction with the <active-if> tag to define greaterthan dependency information for this state variable.
<group>	Used to define the nodes of the group tree.
<groupings>	Defines the section that includes the group tree.
<incr>	Defines the increment that an integer or fixed point variable must use. This restriction means that the variable must have a value equals to its minimum + n * increment, where n is an integer.
<inputs>	Defines the section that describes the input ports of an appliance.

Element Name	Description
<input-output>	Defines the section that describes the input-output ports on an appliance.
<input-ports>	Defines the input ports that are used by a pass-through, recorder or renderer content flow.
<item-count>	Used to denote how many values there are in an <enumerated> type or a <list-group>. Labels for the items can be defined within the <value-labels> tag.
<integer>	Integer type - for variables that take the form of integers. Minimum, maximum and increment values can be defined using the <min>, <max>, and <incr> tags.
<label>	Defines a label to be included within a label dictionary.
<labels>	Defines a label dictionary for an appliance object or group. The <map> tag specifies a dictionary to be associated with the specific value of a variable.
<lessthan>	Used in conjunction with the <active-if> tag to define lessthan dependency information for this state variable.
<list-group>	Specifies a special group that represents a list of data. The variables contained in this group have multiple values for every item in the list.
<list-selection>	List selection type - for variables that specify a selection in a list in a different location within the spec than where that list is defined.
<map>	Specifies a label dictionary for the specific value of a variable.
<min>	Defines the minimum value a numeric variable may take.
<max>	Defines the maximum value a numeric variable may take.
<modifies-state>	Allows a command to specify that its invocation will modify a state elsewhere in the specification.
<not>	Negates the value of a dependency equation.

Element Name	Description
<or>	Defines an OR relation with the dependencies that are contained within.
<object>	Defines an object that is related to a content flow.
<objects>	Defines the section in which groups and objects related to a content flow may be specified.
<outputs>	Defines the section that describes the output ports of an appliance.
<output-ports>	Defines the output ports that are used by a pass-through or source content flow.
<pass-through>	Defines a pass-through content flow.
<phonetic>	Provides pronunciation information for speech interfaces that are based upon this specification language. Many pronunciations may be included in a label dictionary.
<pointpos>	Defines the position of the decimal point for a fixed point type.
<port>	Defines a port of an appliance.
<port-group>	Defines a group of ports of an appliance.
<ports>	Defines the section in which all ports of an appliance are described.
<processing>	Defines any processing that an appliance does to a content stream in a pass-through content flow, such as if a channel is blocked.
<recorder>	Defines a content flow sink that records the stream sent to it.
<ref-value>	Used to define a dynamic value for any parameter element of a state variable type, except the <pointpos> tag. E.g. a ref-value can be used to set the maximum of a numeric state variable to be the value of another state variable. ref-value elements may also be used with dependencies and in several other locations.
<renderer>	Defines a content flow sink that renders the stream sent to it.

Element Name	Description
<required-if>	Specifies the circumstances when a state variable's value is required. If this element is omitted from a state variable's definition, then a value is not required whenever the state variable is active. Otherwise, a value is required whenever the contents of this element are satisfied.
<selections>	Defines the number of selections allowed in a list and whether the user is allowed to modify the selection.
<server-side-error-correction>	Specifies that this state variable will error corrected on the server-side.
<sortable>	Specifies that this list group has not natural order and may be arbitrarily sorted by the user interface.
<source>	Defines a source of content.
<spec>	Every specification begins with this tag.
<specific-values-important>	Defines that the specific values of a numeric type will be interesting to the user, as opposed to the position of the value in the range.
<state>	Defines a state variable appliance object.
<string>	String type - for variables that take the form of strings.
<text-to-speech>	Defines a text-to-speech entry to be included within a label dictionary.
<true>	True - for use in dependency expressions.
<type>	Describes the value space of a state variable (ex: Boolean, Integer, etc...) and the labels that its values take.
<types>	The first section of the specification in which authors may define types that the re-use throughout a specification.
<undefined>	Used in conjunction with the <active-if> element to define a dependency that a state variable must not have any value.
<union-group>	Specifies a special group in which only one of the children may have a valid value.
<value-labels>	Contains one or more <map> tags that provide label dictionaries for specific values that the variable might have.

B.3 Element Descriptions

<spec>

```
<spec name="Sample Specification" version="PUC/2.1"></spec>
```

Every specification begins with this element. The name specified in the *name* attribute is a machine-readable name for the parser. The contained <labels> element specifies human-readable names for the appliance.

Placement:

First element of the spec after the required XML header (E.g. <?xml version="1.0" encoding="UTF-8"?>)

Parameters:

- *name* - **required** The name of the appliance defined in this specification
- *version* - **required** The version of the specification language being used. Current valid values are "PUC/2.0", "PUC/2.1", "PUC/2.2", and "PUC/2.3". (This document describes PUC/2.3)
- *guid* – **required** A global identifier for this specification.

Must Contain:

<groupings>, <labels>, <types>, <ports>, <content-flow>

<types>

```
<types></types>
```

Contains snippets of specification that the author will reuse in the groupings section.

Placement:

Inside the <spec> element

May Contain:

<group>, <list-group>, <union-group>, <state>, <command>, <explanation>, <type>

<groupings>

```
<groupings></groupings>
```

Contains the entire group tree.

Placement:

Inside the <spec> element

May Contain:

<group>, <list-group>, <union-group>

<group>

```
<group name="GroupA" priority="10"></group>
```

Defines the nodes of the group tree.

Group nodes may be assigned a label dictionary with the <labels> tag. Group nodes may also specify dependencies for all their members using the <active-if> tag. These dependencies are applied to the rest of the member's dependencies with an AND logical operation.

Each group has a unique name that is its local name concatenated with the names of all its parent groups.

Placement:

Inside the <groupings>, <group>, <list-group>, <types>, or <union-group> elements

Parameters:

- *name* - **required** The local name of this group.
- *type-name* - required if within types section, optional otherwise, defines a type with the given name
- *is-a* - The Smart Template that represents this group and its children.
- *priority* - The priority this group should be assigned relative to other objects in its parent group.

May Contain:

<labels>, <active-if>, <state>, <command>, <explanation>, <group>, <list-group>, <union-group>

<list-group>

```
<list-group name="ListGroupA" priority="10"></list-group>
```

Defines a special node of the group tree that represents a list.

A list group has the all the same qualities of a regular group, but also may contain some extra elements for describing the features of the list.

A list group automatically creates two states within itself. The "Length" state stores the current length of the list. If this state has an undefined value, then there are no items in the list. The "Selection" state stores the current selection(s). If multiple selections are allowed, then "Selection" is treated like another list-group, allowing list operators like <apply-over> to be applied to it.

Three elements are provided, <item-count>, <min>, and <max>, to allow the specification writer to pre-specify constraints on the size of the list.

Placement:

Inside the <groupings>, <group>, <list-group>, <types> or <union-group> elements

Parameters:

- *name* - **required** The local name of this group.
- *type-name* - required if within types section, optional otherwise, defines a type with the given name
- *is-a* - The Smart Template that represents this group and its children.
- *priority* - The priority this group should be assigned relative to other objects in its parent group.

May Contain:

<labels>, <active-if>, <state>, <command>, <explanation>, <group>, <list-group>, <union-group>, <selections>, <sortable>, <item-count>, <min>, <max>

<union-group>

```
<union-group name="UnionGroupA" access="read-only"></union-group>
```

Defines a special node of the group tree that represents a union.

A union group has the all the same qualities of a regular group. It does not contain any extra elements for describing the union.

The union group automatically creates one state named "ChildUsed", which defines the active child variable/group. The access parameter of this state is defined by the *access* attribute of the union group element.

Placement:

Inside the <groupings>, <group>, <list-group>, <types> or <union-group> elements

Parameters:

- *name* - **required** The local name of this group.
- *type-name* - required if within types section, optional otherwise, defines a type with the given name
- *access* - Defines how users interact with the ChildUsed variable. Possible values are *read-only* and *read-write*.
- *is-a* - The Smart Template that represents this group and its children.
- *priority* - The priority this group should be assigned relative to other objects in its parent group.

May Contain:

<labels>, <active-if>, <state>, <command>, <explanation>, <group>, <list-group>, <union-group>

<selections>

```
<selections number="one" access="read-only" />
```

Defines the number of selections available in a list and whether or not a user may change the current selection(s).

Placement:

Inside the <list-group> element

Parameters:

- *number* - **required** The number of allowed selections. Possible values are *one* and *multiple*.
- *access* - Defines how users interact with the ChildUsed variable. Possible values are *read-only* and *read-write*.

<sortable>

```
<sortable/>
```

Specifies that this list-group has not natural order and can be arbitrarily sorted by the user interface.

Placement:

Inside the <list-group> element

<state>

```
<state name="StateName" access="read-write"
       priority="10"></state>
```

Defines a state variable appliance object.

Placement:

Inside <group>, <list-group>, <types> or <union-group>

Parameters:

- *name* - **required** The name of the state variable.

- *type-name* - required if within types section, optional otherwise, defines a type with the given name
- *priority* - The priority this state should be assigned relative to other objects in the same group.
- *is-a* - The Smart Template that represents this group and its children.
- *access* - Defines how users interact with the ChildUsed variable. Possible values are *read-only* and *read-write*.

May Contain:

<active-if>, <apply-type>, <completions-available>, <default-value>, <labels>, <required-if>, <server-side-error-correction>, <type>

<command>

```
<command name="CommandName" priority="10"></command>
```

Defines a command appliance object.

Placement:

Inside <group>, <list-group>, <types>, or <union-group>

Parameters:

- *name* - **required** The name of the command.
- *type-name* - required if within types section, optional otherwise, defines a type with the given name
- *priority* - The priority this state should be assigned relative to other objects in the same group.

May Contain:

<active-if>, <labels>, <modifies-state>

<modifies-state>

```
<modifies-state state=”Base.Power”/>
```

Specifies that this command will modify a state in this specification. This may cause the interface generator to appropriately display changes to the specified state variable following an invocation of this command.

Placement:

Inside the the <command> element.

Parameters:

- *state* – **required** The name of the state that will be modified.

<explanation>

```
<explanation name="ExplanationName" priority="10"></explanation>
```

Defines an explanation appliance object. The labels block, which is required for an explanation object, defines the text that will be used for the explanation.

Placement:

Inside <group>, <list-group>, <types>, or <union-group>

Parameters:

- *name* - **required** The name of the explanation.
- *type-name* - required if within types section, optional otherwise, defines a type with the given name
- *priority* - The priority this state should be assigned relative to other objects in the same group.

May Contain:

<active-if>, <labels>

<type>

```
<type name="TypeName"></type>
```

Describes the value space of a state variable (ex: Boolean, Integer, etc...) and the labels that its values take.

Placement:

Inside the <state> and <types> tag

Parameters:

- *type-name* - The name of the type object. Required if this element is contained in a types block.

May Contain:

<binary>, <boolean>, <enumerated>, <fixedpt>, <floatingpt>, <integer>, <list-selection>, <string>, <value-labels>

<apply-type>

```
<apply-type name="TypeName"/>
```

Allows the re-use of an existing type block within a specification. Using this element is exactly the same as cutting and pasting the type block. No data will be shared with other states or groups that apply the same type. The type must have been declared earlier in the specification document.

Placement:

Inside a <state>, <group>, <list-group>, or <union-group> element

Parameters:

- *type-name* - **required** The name of the appliance object to reference.
- *name* - optional if this element is not inside a state element.
- *priority* - optional if this element is not inside a state element.

- *access* - optional if this element is not inside a state element and the type being applied is a state variable or a union-group.

<default-value>

```
<default-value></default-value>
```

Specifies a default value for a state variable. In an interface generator, these values would be used when the UI needs to prompt the user for a new value and could also be used for demonstrational purposes.

Placement:

Inside the <state> element

May Contain:

<constant>, <ref-value>

<required-if>

```
<required-if></required-if>
```

Contains dependency information that defines when the value of a state variable is required for successful operation of the appliance. The content of this element is the same as the <active-if> element. If this element is not specified, then the object will not be required. To ensure that an object is required, create this element containing a <true> element.

Placement:

Inside the <state> element

May Contain:

<and>, <apply-over>, <defined>, <equals>, <false>, <greaterthan>, <lessthan>, <not>, <or>, <true>, <undefined>

<binary/>

```
<binary/>
```

Binary type - contains any kind of a binary data, such as sounds or images. A Smart Template must be used with this type to ensure proper interpretation and rendering.

This type may contain arbitrary tags that act are parameters that particular Smart Templates will recognize.

Placement:

Inside the <type> element

May Contain:

Any arbitrary tag.

<boolean/>

<boolean/>

Boolean type - takes on true or false values.

Placement:

Inside the <type> element

<enumerated>

<enumerated></enumerated>

Enumerated type - Define the number of items this composite type contains using the <item-count> element. Note: the <value-labels> element must contain <map> elements that map each of the enumerated values with a label. So if there are 5 enumerated values in a particular enumerated type (denoted by <item-count>5<item-count/>) the value-labels section must contain 5 different mappings of values to labels.

Enumerated type values are treated as integers that range between 1 and the number of items in the type. Zero is not a valid value for an enumerated type. This is important when you specify an index to the <map> tag;

Placement:

Inside the <type> element

May Contain:

<item-count>

<item-count>

<item-count></item-count>

Used to denote how many enumerated values there are in an <enumerated> type or a fixed number of items in a <list-group>. Labels for the items can be defined within the <value-labels> tag.

Placement:

Inside the <enumerated> and <list-group> elements

<fixedpt>

<fixedpt></fixedpt>

Fixed Point type - for variables that take the form of decimal values with a fixed decimal point. The location of the fixed decimal point is specified with the <pointpos> element. Minimum, maximum, and increment values can be defined using the <min>, <max>, and <incr> tags.

Placement:

Inside the <type> element

May Contain:

<pointpos>, <incr>, <max>, <min>

<pointpos>

<pointpos></pointpos>

Defines the position of the decimal point for a fixed point type.

Placement:

Inside the <fixedpt> tag

<floatingpt>

```
<floatingpt></floatingpt>
```

Floating Point type - for variables that take the form of decimal values. Minimum and maximum values can be defined using the <min> and <max> tags.

Placement:

Inside the <type> element

May Contain:

<max>, <min>

<integer>

```
<integer></integer>
```

Integer type - for variables that take the form of integers. Minimum, maximum, and increment values can be defined using the <min>, <max>, and <incr> tags.

Placement:

Inside the <type> element

May Contain:

<incr>, <max>, <min>, <specific-values-important>

<specific-values-important/>

```
<specific-values-important/>
```

Defines that the user will want to modify this integer type to any of the specific values that is supports. Thus, a slider or other control that enables only imprecise changes of the value should not be used.

Placement:

Inside the <integer> element.

<list-selection>

```
<list-selection></list-selection>
```

List selection type - for variables that represent and independent selection within a list elsewhere in the specification. This is particular useful in situations where the some aspect of the appliance is configured as a list, and then an item from this list must be selected often during normal operation (e.g. the channels on a VCR or TV).

Placement:

Inside the <type> element

May Contain:

<active-if>

<incr> ... <max> ... <min>

```
<incr></incr>
<max></max>
<min></min>
```

Describe minimum, maximum and increment values that the variable may take.

The increment may not be defined for the floating point type.

Placement:

Inside the <fixedpt>, <floatingpt> (except for incr), <integer>, <list-group> (except for incr) and <string> elements.

May Contain:

<constant>, <ref-value>

<string>

```
<string></string>
```

String type - for variables that take the form of strings. You can specify parameters about the length of the string to help out the interface generators.

Placement:

Inside the <type> element

May Contain:

<min>, <max>, <average>

<average>

```
<average></average>
```

Describe the average length of a string.

Placement:

Inside the <string> element

May Contain:

<constant>, <ref-value>

<value-labels>

```
<value-labels></value-labels>
```

Contains one or more <map> tags that provide label dictionaries for specific values that the variable might have.

Placement:

Inside the <type> element

May Contain:

<map>

<map>

```
<map index="value"></map>
```

Specifies a label dictionary for the specific value of a variable (specified by the index parameter).

Certain values of a variable can also be enabled based on dependency information which is specified in the enclosed <active-if> element.

Placement:

Inside the <value-labels> element.

Parameters:

- *index* - **required** The value to associate this dictionary with

May Contain:

<labels>, <active-if>

<labels>

```
<labels></labels>
```

Defines a label dictionary for an appliance object or group.

Placement:

Inside the <command>, <explanation>, <group>, <list-group>, <union-group>, <state>, <spec>, or <map> elements.

May Contain:

<label>, <ref-value>, <phonetic>, <text-to-speech>

<text-to-speech>

```
<text-to-speech text="<whisper>mute</whisper>"
                recording="mute.au"/>
```

Defines a text-to-speech entry to be included within a label dictionary. The text parameter may contain embedded SABLE markup tags.

Placement:

Inside the <labels> element

Parameters:

- *text* - **required** the text to be spoken. May contain embedded SABLE tags.
- *recording* - a recording of the text, if available

<label>

```
<label></label>
```

Defines a label to be included within a label dictionary.

Placement:

Inside the <labels> element

May Contain:

string

<phonetic>

```
<phonetic></phonetic>
```

Defines a pronunciation to be included within a label dictionary.

Placement:

Inside the <labels> element

May Contain:

string of phonemes using the arpabet representation

<active-if>

```
<active-if ignore="all" | "parent"></active-if>
```

Contains dependency information for an appliance object or group of objects. Defines an *and* relation with all dependencies that are contained within, unless they are grouped within a logical operation block, such as <and> or <or> tags.

Placement:

Inside the <command>, <explanation>, <group>, <list-group>, <union-group>, <map>, <list-selection>, or <state> elements.

Parameters:

- *ignore* - stop dependency inheritance through the group tree. The possible options are to omit the option, *parent*, and *all*

May Contain:

<and>, <apply-over>, <defined>, <equals>, <false>, <greaterthan>, <lessthan>, <not>, <or>, <true>, <undefined>

<apply-over>

```
<apply-over list="SomeList" items="all" true-if="any">
</apply-over>
```

Used to apply dependency information to lists of information.

There are three different ways that dependencies can be applied. The apply-over block can be true if the dependencies are true for any item in the list, if they are true for all items in the list, or if they are true for no items in the list. The particular choice is chosen with the true-if attribute.

Placement:

Inside the <active-if>, <and>, <apply-over>, <not>, or <or> elements.

Parameters:

- *list* - **required** The name of the list group that the dependencies are applied to it.
- *true-if* - Sets when the apply-over returns true. The possible options are *any*, *all*, or *none*.

May Contain:

<and>, <apply-over>, <defined>, <equals>, <false>, <greaterthan>, <lessthan>, <not>, <or>, <true>, <undefined>

<defined>

```
<defined state="SomeState"/>
```

Used in conjunction with the <active-if> tag to define that some appliance object depends on a state variable having a defined value.

Placement:

Inside the <active-if>, <and>, <apply-over>, <not>, or <or> elements.

Parameters:

- *state* - **required** The name of the state that is depended upon

<undefined>

```
<undefined state="SomeState"/>
```

Used in conjunction with the <active-if> tag to define that some appliance object depends on a state variable not having a defined value.

Placement:

Inside the <active-if>, <and>, <apply-over>, <not>, or <or> elements.

Parameters:

- *state* - **required** The name of the state that is depended upon

<equals>

```
<equals state="SomeState">value</equals>
```

Used in conjunction with the <active-if> tag to define equals dependency information for this state variable.

Placement:

Inside the <active-if>, <and>, <apply-over>, <not>, or <or> elements.

Parameters:

- *state* - **required** The name of the state that is depended upon

May Contain:

<ref-value>, <constant>

<greaterthan>

```
<greaterthan state="SomeState">value</greaterthan>
```

Used in conjunction with the <active-if> tag to define greater-than dependency information for this state variable.

Placement:

Inside the <active-if>, <and>, <apply-over>, <not>, or <or> elements.

Parameters:

- *state* - **required** The name of the state that is depended upon

May Contain:

<ref-value>, <constant>

<lessthan>

```
<lessthan state="SomeState">value</lessthan>
```

Used in conjunction with the <active-if> tag to define less-than dependency information for this state variable.

Placement:

Inside the <active-if>, <and>, <apply-over>, <not>, or <or> elements.

Parameters:

- *state* - **required** The name of the state that is depended upon

May Contain:

<ref-value>, <constant>

<and>

```
<and></and>
```

Defines an *and* relation with the dependencies that are contained within.

Placement:

Inside the <active-if>, <and>, <apply-over>, <not>, or <or> elements.

May Contain:

<and>, <apply-over>, <defined>, <equals>, <false>, <greaterthan>, <lessthan>, <not>, <or>, <true>, <undefined>

<or>

```
<or></or>
```

Defines an *or* relation with the dependencies that are contained within.

Placement:

Inside the <active-if>, <and>, <apply-over>, <not>, or <or> elements.

May Contain:

<and>, <apply-over>, <defined>, <equals>, <false>, <greaterthan>, <lessthan>, <not>, <or>, <true>, <undefined>

<not>

```
<not></not>
```

Defines a *not* relation with the dependencies that are contained within.

Placement:

Inside the <active-if>, <and>, <apply-over>, or <or> elements.

May Contain:

<and>, <apply-over>, <defined>, <equals>, <false>, <greaterthan>, <lessthan>, <not>, <or>, <true>, <undefined>

<true>

```
<true/>
```

Specifies a true value for a dependency formula.

Placement:

Inside the <active-if>, <and>, <apply-over>, <not>, or <or> elements.

<false>

```
<false/>
```

Specifies a false value for a dependency formula.

Placement:

Inside the <active-if>, <and>, <apply-over>, <not>, or <or> elements.

<ref-value>

```
<ref-value state="StateName"/>
```

Used to define a numeric value for any of the parameter elements, except the <pointpos> tag, that depends on the value of a numeric state variable. E.g. a ref-value can be used to set the maximum of a numeric state variable to be the value of another state variable.

Placement:

Inside the <max>, <min>, <average>, and <incr> elements.

Parameters:

- *state* - **required** The name of a state variable with a numeric type

<constant>

```
<constant value="12">
```

Used to define a constant value in any location where a reference would also be accepted (i.e. the <ref-value> element).

Placement:

Inside the <max>, <min>, <average>, and <incr> elements.

Parameters:

- *value* - **required** The constant value.

<completions-available>

```
<completions-available>
```

Used to specify that completions are available for this state variable.

Placement:

Inside <state>

<server-side-error-corrections>

```
<server-side-error-corrections>
```

Used to specify that this variable will be automatically error corrected on the server-side.

Placement:

Inside <state>

<ports>

```
<ports></ports>
```

Defines the section in which the physical ports of the appliance are described.

Placement:

Inside the <spec> element.

May Contain:

<inputs>, <input-output>, <outputs>

<inputs>

```
<inputs></inputs>
```

Defines the physical input ports of an appliance.

Placement:

Inside the <ports> element.

May Contain:

<port>, <port-group>

<outputs>

```
<outputs></outputs>
```

Defines the physical output ports of an appliance.

Placement:

Inside the <ports> element.

May Contain:

<port>, <port-group>

<input-output>

```
<input-output></input-output>
```

Defines any physical ports of an appliance which can simultaneously be both inputs and outputs.

Placement:

Inside the <ports> element.

May Contain:

<port>, <port-group>

<port-group>

```
<port-group name="Output" content-type="av"></port-group>
```

Defines a group of physical ports. This construct is useful for grouping ports that are activated simultaneously. Referencing the name given to a port group in a content flow is shorthand for referencing all of its contained ports.

Placement:

Inside the <inputs>, <input-output>, <outputs>, or <port-group> elements.

Parameters:

- *name* – **required** The name for this port group. Must be unique among other names for ports and port groups at this level.
- *content-type* – **required** The content type that is carried over the ports contained in this group.

May Contain:

<port>, <port-group>

<port>

```
<port name="Right" content-type="component-audio-right"
      physical-type="RCA"/>

<port name="Input 1.Video" channel="4"></port>
```

This element has two different uses. The first use, when within the <ports> block, defines a physical port of the appliance. The full name of this port is similar to the naming of appliance objects and has the form <groupname>.<groupname>.<name> where the port group names are from port groups that contain this port. In this use, the port element may not contain any other elements or content.

The second use, when within the <content-flow> block, defines a port or port group across which content is being accepted as an input or produced as an output. In this use, the port element may contain several elements. In this use, the name must reference a complete, unique port name defined in the ports block.

Placement:

First use: inside the <inputs>, <input-output>, <outputs>, or <port-group> elements.

Second use: inside the <input-ports> or <output-ports> elements.

Parameters:

- *name* – **required** The name for this port group. Must be unique among other names for ports and port groups at this level.
- *content-type* – **required** The content type that is carried over the ports contained in this group.
- *physical-type* – **required** The type of the physical port on the appliance. E.g. RCA, HDMI, VGA, etc.
- *channel* – Only available in the second use of this element. Defines the channel to be used from a multi-channel stream.

May Contain:

Second use only: <active-if>, <channel>, <objects>

<content-flow>

```
<content-flow></content-flow>
```

Defines the section in which the internal content flows of an appliance are described.

Placement:

Inside the <spec> element.

May Contain:

<content-group>, <source>, <pass-through>, <recorder>, <renderer>

<content-group>

```
<content-group></content-group>
```

Defines a group of content flows, for the purpose of defining some dependencies that apply to all of the content flows in the group.

Placement:

Inside the <content-flow> or <content-group> elements.

May Contain:

<active-if>, <content-group>, <source>, <pass-through>, <recorder>, <renderer>

<source>

```
<source name="Tape" content-type="av"></source>
```

Describes a source of content within an appliance.

Placement:

Inside the <content-flow> or <content-group> elements.

Parameters:

- *name* – **required** A name for the source.
- *content-type* – **required** The type of content produced by this source.

May Contain:

<active-if>, <output-ports>, <objects>

<pass-through>

```
<pass-through content-type="av"></pass-through>
```

Describes a pass through for content on this appliance.

Placement:

Inside the <content-flow> or <content-group> elements.

Parameters:

- *content-type* – **required** The type of content produced by this source.

May Contain:

<active-if>, <input-ports>, <processing>, <output-ports>, <objects>

<recorder>

```
<recorder name="Tape" content-type="av"></recorder>
```

Describes a content sink that records a content stream received as an input.

Placement:

Inside the <content-flow> or <content-group> elements.

Parameters:

- *name* – **required** A name for the source.
- *content-type* – **required** The type of content produced by this source.

May Contain:

<active-if>, <input-ports>, <objects>

<renderer>

```
<renderer name="Screen" content-type="video"></renderer>
```

Describes a content sink that renders a content stream received as an input for the user.

Placement:

Inside the <content-flow> or <content-group> elements.

Parameters:

- *name* – **required** A name for the source.
- *content-type* – **required** The type of content produced by this source.

May Contain:

<active-if>, <input-ports>, <objects>

<input-ports>

```
<input-ports></input-ports>
```

Describes the input ports that may be used by a content flow.

Placement:

Inside the <pass-through>, <recorder>, and <renderer> elements.

May Contain:

<port>

<output-ports>

```
<output-ports></output-ports>
```

Describes the output ports that may be used by a content flow.

Placement:

Inside the <pass-through> or <source> elements.

May Contain:

<port>

<processing>

```
<processing></processing>
```

Defines processing that may be done to a content stream as it is being passed through an appliance.

Placement:

Inside the <pass-through> element.

May Contain:

<block>

<channel>

```
<channel value="4"/>
```

```
<channel state="Base.Controls.Channel"/>
```

Defines the channel of a multi-channel content stream to be modified by the appliance. Must have one of the two parameter options.

Placement:

Inside the <port> element.

Parameters:

- *value* – Defines a state value for the channel.
- *state* – Defines a state variable or group that will contain the value of the channel to be operated on.

<block>

```
<block channel="4"></block>
```

Defines the block operation that may be applied by an appliance to one or more channels of a multi-channel content stream. When this operation is active, the channel(s) received from the input are not passed through the output.

Placement:

In the <processing> element.

Parameters:

- *channel* – Defines the channel to block

May Contain:

<active-if>, <channel>

<objects>

```
<objects></objects>
```

Defines a set of groups and appliance objects that are related to the control of this content flow.

Placement:

Inside the <pass-through>, <recorder>, <renderer>, and <port> elements.

May Contain:

<group>, <object>

<object/>

```
<object name="Base.Power"/>
```

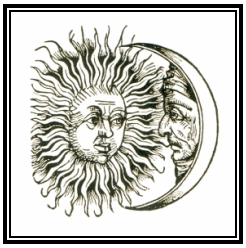
Defines an object that is related to the control of a content flow. This name must resolve to an appliance object within the specification (and not a group).

Placement:

Inside the <objects> element.

Parameters:

- *name* – **required** The name of an appliance object within the specification.



APPENDIX C

Other PUC XML Language Schemas

The PUC also uses XML as its communication protocol, to store mappings in its knowledge base, and to describe systems of multiple appliances. Each of these XML languages is described in this appendix.

C.1 Communication Protocol Schema

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema targetNamespace="http://www.cs.cmu.edu/~pebbles/puc/puc-protocol"
elementFormDefault="qualified"
  xmlns="http://www.cs.cmu.edu/~pebbles/puc/puc-protocol"
xmlns:mstns="http://www.cs.cmu.edu/~pebbles/puc/puc-protocol"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!--
    The PUC protocol consists of an eight-byte header, XML content, and
    optional format-independent binary content.

    The header is divided into two four-byte chunks.  The first chunk is an
    integer giving the full length of the message (not including the header).
    The second chunk is an integer giving the length of the XML portion of
    the message.  Both of these integers are sent using the network byte
    order.

    The XML content is described by this Schema.

    The binary content may be in any arbitrary format.  The format is usually
    defined within an element of the accompanying XML message.
    -->
```

```
<xs:element name="message">
  <xs:complexType>
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:element name="state-change-notification" type="PUCDataType" />
      <xs:element name="state-change-request" type="PUCDataType" />
      <xs:element name="binary-state-change-notification" type="BinaryStateChangeNotifyType" />
      <xs:element name="state-value-request" type="StateValueRequestType" />
      <xs:element name="command-invoke-request" type="CommandInvokeRequestType" />
      <xs:element name="spec-request" />
      <xs:element name="device-spec" type="DeviceSpecType" />
      <xs:element name="full-state-request" />
      <xs:element name="server-information-request" />
      <xs:element name="server-information" type="ServerInformationType" />
      <xs:element name="alert-information" type="xs:string" />
      <xs:element name="register-device" type="DeviceType" />
      <xs:element name="unregister-device" type="UnregisterDeviceType" />
    </xs:choice>
  </xs:complexType>
</xs:element>
<xs:complexType name="BinaryStateChangeNotifyType">
  <xs:sequence>
    <xs:element name="state" type="StateWithContentAttribType" minOccurs="1" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="StateWithContentAttribType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="content-type" type="xs:string" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="StateValueRequestType">
  <xs:sequence>
    <xs:element name="state" type="xs:string" minOccurs="1" maxOccurs="1" />
    <xs:any minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="CommandInvokeRequestType">
  <xs:sequence>
    <xs:element name="command" type="xs:string" minOccurs="1" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="DeviceSpecType">
  <xs:sequence>
    <xs:element name="spec" type="xs:string" minOccurs="1" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ServerInformationType">
  <xs:sequence>
    <xs:element name="server-name" type="xs:string" minOccurs="1" maxOccurs="1" />
    <xs:element name="device" type="DeviceType" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="DeviceType">
  <xs:sequence>
    <xs:element name="name" type="xs:string" minOccurs="1" maxOccurs="1" />
    <xs:element name="port" type="xs:integer" minOccurs="1" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="UnregisterDeviceType">
  <xs:sequence>
    <xs:element name="port" type="xs:integer" minOccurs="1" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="PUCDataType">
  <xs:choice minOccurs="1" maxOccurs="1">
    <xs:sequence>
      <xs:element name="state" type="xs:string" />
      <xs:element name="value" type="OldValueType" />
    </xs:sequence>
    <xs:element name="data" type="ListDataType" />
    <xs:element name="change" type="ChangeDataType" />
    <xs:element name="insert" type="InsertType" />
    <xs:element name="delete" type="DeleteType" />
    <xs:element name="replace" type="ReplaceType" />
  </xs:choice>
</xs:complexType>
<xs:complexType name="OldValueType" mixed="true">
  <xs:sequence>
    <xs:element name="undefined" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="NewValueType" mixed="true">
  <xs:sequence>
    <xs:element name="undefined" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="state" type="xs:string"/>
</xs:complexType>
<xs:complexType name="AnyListContentType">
  <xs:choice>
    <xs:element name="data" type="ListDataType" />
    <xs:element name="change" type="ChangeDataType" />
    <xs:element name="insert" type="InsertType" />
    <xs:element name="delete" type="DeleteType" />
    <xs:element name="replace" type="ReplaceType" />
    <xs:sequence>
      <xs:element name="value" type="NewValueType" minOccurs="1" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:choice>
</xs:complexType>
<xs:complexType name="AfterChangeOpListType">
  <xs:choice>
    <xs:element name="data" type="ListDataType" />
    <xs:sequence>
      <xs:element name="value" type="NewValueType" minOccurs="1" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:choice>
</xs:complexType>
<xs:complexType name="ListDataType">
  <xs:sequence>
    <xs:element name="el" type="AfterChangeOpListType" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="state" type="xs:string" use="required" />
</xs:complexType>
<xs:complexType name="ChangeDataType">
  <xs:sequence>
    <xs:element name="el" type="AnyListContentType" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
```

```
    <xs:attribute name="state" type="xs:string" use="required" />
    <xs:attribute name="index" type="xs:integer" use="required" />
  </xs:complexType>
  <xs:complexType name="DeleteType">
    <xs:attribute name="state" type="xs:string" use="required" />
    <xs:attribute name="begin" type="xs:integer" use="required" />
    <xs:attribute name="length" type="xs:integer" use="required" />
  </xs:complexType>
  <xs:complexType name="InsertType">
    <xs:sequence>
      <xs:element name="el" type="AfterChangeOpListType" minOccurs="1" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="state" type="xs:string" use="required" />
    <xs:attribute name="after" type="xs:integer" use="required" />
  </xs:complexType>
  <xs:complexType name="ReplaceType">
    <xs:sequence>
      <xs:element name="el" type="AfterChangeOpListType" minOccurs="1" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="state" type="xs:string" use="required" />
    <xs:attribute name="begin" type="xs:integer" use="required" />
    <xs:attribute name="length" type="xs:integer" use="required" />
  </xs:complexType>
</xs:schema>
```

C.2 Knowledge Base Schema

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema targetNamespace="http://www.cs.cmu.edu/~pebbles/puc-kb" elementFormDefault="qualified"
  xmlns="http://www.cs.cmu.edu/~pebbles/puc-kb" xmlns:mstns="http://www.cs.cmu.edu/~pebbles/puc-kb"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!-- Top-Level Element -->
  <xs:element name="puc-knowledgebase">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="spec-store-table" type="SpecStoreTableType" minOccurs="1" maxOccurs="1"/>
        <xs:element name="spec-map" type="SpecMapType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- Attribute Types -->
  <xs:simpleType name="WrapAttributeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="allowed"/>
      <xs:enumeration value="not-allowed"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="IndexAttributeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="1-indexed"/>
      <xs:enumeration value="0-indexed"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="SourceAttributeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="manual"/>
      <xs:enumeration value="regular-automatch"/>
      <xs:enumeration value="transitive-automatch"/>
```

```
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="ConfidenceType">
  <xs:restriction base="xs:float">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="1"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="SimilarityAttributeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="sparse"/>
    <xs:enumeration value="branch"/>
    <xs:enumeration value="significant"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CostAttribType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="equivalent"/>
    <xs:enumeration value="fully-contained"/>
    <xs:enumeration value="overlapping"/>
    <xs:enumeration value="no-overlap"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="MapTypeAttributeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="function"/>
    <xs:enumeration value="organization"/>
  </xs:restriction>
</xs:simpleType>
<!-- Spec Store Table Types -->
<xs:complexType name="SpecStoreTableType">
  <xs:sequence>
    <xs:element name="spec" type="SpecType" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="SpecType">
  <xs:sequence>
    <xs:element name="location" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="generated-interface-location" type="xs:string" minOccurs="0" maxOccurs="1"/>
    <xs:element name="object-count" type="xs:integer" minOccurs="1" maxOccurs="1"/>
    <xs:element name="similar-specs-table" type="SimilarSpecsTableType" minOccurs="0" maxOccurs="1"/>
    <xs:element name="mappings-table" type="MappingsTableType" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="optional"/>
  <xs:attribute name="guid" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="MappingsTableType">
  <xs:sequence>
    <xs:element name="mapped" type="MappedType" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="MappedType">
  <xs:attribute name="path" type="xs:string" use="required"/>
  <xs:attribute name="type" type="MapTypeAttributeType" use="optional"/>
  <xs:attribute name="spec-name" type="xs:string" use="optional"/>
  <xs:attribute name="spec-guid" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="SimilarSpecsTableType">
  <xs:sequence>
```

```
    <xs:element name="spec-entry" type="SpecEntryType" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="SpecEntryType">
  <xs:sequence>
    <xs:element name="similar-objects" type="xs:integer" minOccurs="1" maxOccurs="unbounded"/>
    <xs:element name="branch" type="BranchType" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="optional"/>
  <xs:attribute name="guid" type="xs:string" use="required"/>
  <xs:attribute name="similarity" type="SimilarityAttributeType" use="required"/>
</xs:complexType>
<xs:complexType name="BranchType">
  <xs:attribute name="spec-group" type="xs:string" use="required"/>
  <xs:attribute name="spec-entry-group" type="xs:string" use="required"/>
</xs:complexType>
<!-- Spec Map Types -->
<xs:complexType name="SpecMapType">
  <xs:sequence>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="group" type="GroupType"/>
      <xs:element name="mapping" type="MappingType"/>
      <xs:element name="state-mapping" type="StateMappingType"/>
      <xs:element name="list-mapping" type="ListMappingType"/>
      <xs:element name="template-mapping" type="TemplateMappingType"/>
      <xs:element name="location-mapping" type="LocationMappingType"/>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="spec1-guid" type="xs:string" use="required"/>
  <xs:attribute name="spec1-name" type="xs:string" use="optional"/>
  <xs:attribute name="spec2-guid" type="xs:string" use="required"/>
  <xs:attribute name="spec2-name" type="xs:string" use="optional"/>
</xs:complexType>
<xs:complexType name="GroupType">
  <xs:sequence>
    <xs:element name="label" type="xs:string" minOccurs="0" maxOccurs="1"/>
    <xs:element name="source" type="SourceType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="cost" type="MappingCostType" minOccurs="1" maxOccurs="1"/>
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element name="mapping" type="MappingType"/>
      <xs:element name="state-mapping" type="StateMappingType"/>
      <xs:element name="template-mapping" type="TemplateMappingType"/>
      <xs:element name="location-mapping" type="LocationMappingType"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="MappingCostType">
  <xs:attribute name="one-to-two" type="CostAttribType" use="required"/>
  <xs:attribute name="two-to-one" type="CostAttribType" use="required"/>
</xs:complexType>
<xs:complexType name="SourceType">
  <xs:attribute name="type" type="SourceAttributeType" use="required"/>
  <xs:attribute name="confidence" type="ConfidenceType" use="optional"/>
</xs:complexType>
<xs:complexType name="LocationMappingType">
  <xs:sequence>
    <xs:element name="label" type="xs:string" minOccurs="0" maxOccurs="1"/>
    <xs:element name="source" type="SourceType" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
```

```
  <xs:attribute name="spec1" type="xs:string" use="required"/>
  <xs:attribute name="spec2" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="TemplateMappingType">
  <xs:sequence>
    <xs:element name="label" type="xs:string" minOccurs="0" maxOccurs="1"/>
    <xs:element name="source" type="SourceType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="cost" type="MappingCostType" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="spec1" type="xs:string" use="required"/>
  <xs:attribute name="spec2" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="StateMappingType">
  <xs:sequence>
    <xs:element name="label" type="xs:string" minOccurs="0" maxOccurs="1"/>
    <xs:element name="source" type="SourceType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="cost" type="MappingCostType" minOccurs="1" maxOccurs="1"/>
    <xs:choice minOccurs="0" maxOccurs="1">
      <xs:element name="all-values-equivalent"/>
      <xs:sequence>
        <xs:element name="value" type="ValueType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="spec1-state" type="xs:string" use="required"/>
  <xs:attribute name="spec2-state" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="ValueType">
  <xs:sequence>
    <xs:element name="spec1" type="SpecValueType" minOccurs="0" maxOccurs="1"/>
    <xs:element name="spec2" type="SpecValueType" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="spec1" type="xs:string" use="optional"/>
  <xs:attribute name="spec2" type="xs:string" use="optional"/>
</xs:complexType>
<xs:complexType name="SpecValueType">
  <xs:choice minOccurs="1" maxOccurs="1">
    <xs:element name="constant" type="ConstantValueType"/>
    <xs:element name="range" type="RangeValueType"/>
    <xs:element name="defined"/>
    <xs:element name="undefined"/>
  </xs:choice>
</xs:complexType>
<xs:complexType name="RangeValueType">
  <xs:attribute name="start" type="xs:string" use="required"/>
  <xs:attribute name="end" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="ConstantValueType">
  <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="MappingType">
  <xs:sequence>
    <xs:element name="label" type="xs:string" minOccurs="0" maxOccurs="1"/>
    <xs:element name="source" type="SourceType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="cost" type="MappingCostType" minOccurs="1" maxOccurs="1"/>
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:sequence>
        <xs:element name="initial" type="SpecStepsType" minOccurs="1" maxOccurs="1"/>
        <xs:element name="step" type="SpecStepsType" minOccurs="1" maxOccurs="1"/>
```

```
      </xs:sequence>
      <xs:sequence>
        <xs:element name="spec1" type="OperationType" minOccurs="1" maxOccurs="1"/>
        <xs:element name="spec2" type="OperationType" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="SpecStepsType">
  <xs:sequence>
    <xs:element name="spec1" type="OperationType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="spec2" type="OperationType" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="OperationType">
  <xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element name="change" type="ChangeOperationType"/>
      <xs:element name="invoke" type="InvokeOperationType"/>
      <xs:element name="repeat" type="RepeatOperationType"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RepeatOperationType">
  <xs:sequence>
    <xs:element name="count" type="CountType" minOccurs="1" maxOccurs="1"/>
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element name="change" type="ChangeOperationType"/>
      <xs:element name="invoke" type="InvokeOperationType"/>
      <xs:element name="repeat" type="RepeatOperationType"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="CountType">
  <xs:choice>
    <xs:element name="variable" type="VariableType"/>
    <xs:element name="constant" type="ConstantValueType"/>
  </xs:choice>
</xs:complexType>
<xs:complexType name="InvokeOperationType">
  <xs:attribute name="command" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="ChangeOperationType">
  <xs:choice minOccurs="0" maxOccurs="1">
    <xs:element name="constant" type="ConstantValueType"/>
    <xs:element name="variable" type="VariableType"/>
    <xs:element name="increase" type="IncreaseValueType"/>
    <xs:element name="decrease" type="DecreaseValueType"/>
  </xs:choice>
  <xs:attribute name="state" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="VariableType">
  <xs:attribute name="name" type="xs:string"/>
</xs:complexType>
<xs:complexType name="IncreaseValueType">
  <xs:attribute name="value" type="xs:string" use="optional"/>
  <xs:attribute name="wrap" type="WrapAttributeType" use="optional"/>
</xs:complexType>
<xs:complexType name="DecreaseValueType">
```

```
    <xs:attribute name="value" type="xs:string" use="optional"/>
    <xs:attribute name="wrap" type="WrapAttributeType" use="optional"/>
  </xs:complexType>
  <xs:complexType name="ListMappingType">
    <xs:sequence>
      <xs:element name="label" type="xs:string" minOccurs="0" maxOccurs="1"/>
      <xs:element name="source" type="SourceType" minOccurs="1" maxOccurs="1"/>
      <xs:sequence minOccurs="0" maxOccurs="1">
        <xs:element name="spec1" type="ListIndexStateMapType" minOccurs="1" maxOccurs="1"/>
        <xs:element name="spec2" type="ListIndexStateMapType" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:sequence>
    <xs:attribute name="spec1-list" type="xs:string" use="required"/>
    <xs:attribute name="spec2-list" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="ListIndexStateMapType">
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:element name="list-index" type="ListIndexMapType"/>
      <xs:element name="state" type="StateMapType"/>
    </xs:choice>
  </xs:complexType>
  <xs:complexType name="ListIndexMapType">
    <xs:attribute name="type" type="IndexAttributeType" use="optional"/>
  </xs:complexType>
  <xs:complexType name="StateMapType">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="type" type="IndexAttributeType" use="optional"/>
  </xs:complexType>
</xs:schema>
```

C.3 Multi-Appliance Wiring Diagram Schema

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema targetNamespace="http://www.cs.cmu.edu/~pebbles/puc-wiring"
                  elementFormDefault="qualified"
                  xmlns="http://www.cs.cmu.edu/~pebbles/puc-wiring"
                  xmlns:mstns="http://www.cs.cmu.edu/~pebbles/puc-wiring"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="puc-wiring-description">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="appliances" type="AppliancesType" minOccurs="1" maxOccurs="1"/>
        <xs:element name="wires" type="WiresType" minOccurs="1" maxOccurs="1"/>
         <xs:element name="preferred-flows" type="PreferredFlowsType" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="AppliancesType">
    <xs:sequence>
      <xs:element name="appliance" type="ApplianceType" minOccurs="1" maxOccurs="unbounded"/>
       <xs:element name="external-source" type="ExternalSourceType" minOccurs="1" maxOc-
curs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ApplianceType">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="spec-guid" type="xs:string" use="required"/>
```

```
    <xs:attribute name="server" type="xs:string" use="required"/>
    <xs:attribute name="port" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="ExternalSourceType">
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="WiresType">
    <xs:sequence>
      <xs:element name="connect" type="ConnectType" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ConnectType">
    <xs:sequence>
      <xs:element name="start" type="ApplianceConnectType" minOccurs="1" maxOccurs="1"/>
      <xs:element name="end" type="ApplianceConnectType" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ApplianceConnectType">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="port" type="xs:string" use="optional"/>
  </xs:complexType>
  <xs:complexType name="PreferredFlowsType">
    <xs:sequence>
      <xs:element name="flow" type="PreferredFlowType" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="PreferredFlowType">
    <xs:attribute name="appliance" type="xs:string" use="required"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:schema>
```

APPENDIX D

Specification Authoring Study Instructions

This appendix contains the tutorial document created for teaching the PUC specification language to the subjects of the specification authoring study. The document starts on the next page to preserve formatting.

Specification Authoring Study

We are developing the Personal Universal Controller (PUC) system, which will improve the user interfaces for common home and office appliances by moving the interface to a handheld computer. The system relies on a specification language that is capable of describing the complete functionality of any appliance that the user may encounter. The handheld computer uses the specification of an appliance to automatically generate a user interface that remotely controls that appliance. One of the benefits of this system is that a user interface generated for a new appliance can be made consistent with interfaces that the user has already used for similar appliances. For example, the interface generated by my handheld for controlling a VCR in a conference center would be made consistent with the interface that I use for controlling my VCR at home.

In this study you will create functional specifications that are suitable for interface generation by our PUC system. You will start by learning how to specify an appliance through reading and working a small specification for a to-do list application. Then you will write a specification for an appliance that we have provided you with (Mitsubishi DVCR). Finally, you may optionally write an additional specification for an appliance that you own and have an owner's manual for.

You will author these specifications in an XML-based language that we have developed.

1. General Concepts for Describing Appliance Functionality

This section describes the general concepts about our method for describing appliance functionality. These concepts will be applicable regardless of which authoring method you choose to use for this study. The four most important concepts in our language are:

- The functions of an appliance can be represented by either state variables or state-less commands. State variables have specific type information that describes how they can be manipulated by the interface. Commands and states are collectively called *appliance objects*.
- The structure of the prototype interfaces were often based upon *dependency* information. For example, suppose that an interface was being created for a shelf stereo

system with a tape and CD player. When the power is off, a screen with only a power button widget would be shown, because none of the other objects would be enabled. When the power is on, a screen is shown with many widgets, because most of the objects are active when the power is on. We might also expect this interface to have a panel whose widgets change based upon whether the tape or CD player is active.

- The final representation of any interface can be described using a tree format. It is not reasonable to include the tree representation of one interface in the specification of an appliance however, because the tree may differ for different form factors. For example, the tree will be very deeply branched on a small screen WAP cellular phone interface, whereas the tree will be broader for a desktop PC interface. We prefer specifications that define a *group tree* that is deeply branched. This information can be used for small screen and large screen interfaces alike, because some of the branches can be ignored in a large interface.
- Domain-specific conventions are often used in appliance interfaces, such as the standard number pad on a telephone or the standard play and stop icons used on media players. Interfaces generated by our PUC system need to include these conventions in their generated interfaces, and we have developed "Smart Templates" to help identify pieces of a specification where conventions should be applied.

Each of these items is described in more detail below.

Appliance Objects

Three types of appliance objects are supported in the specification language.

- **States** - Variables that represent data that is stored within the appliance. Examples might be the radio station on a stereo, the number of rings until an answering machine picks up, and the time that an alarm clock is set for. Each variable has a type, and the UI generator assumes that the value of a state may be changed to any value within that type, at any time that the state is enabled. It is possible for state variables to be undefined, i.e. without any value. This commonly happens just after an interface is generated before any values have been assigned, but could occur for other reasons.
- **Commands** - Any function of an appliance that cannot be described by variables alone. They may be used in situations where invoking the command caused an un-

known change to a known state variable (such as the "seek" function on a radio), or in situations where the state variable is not known (manufacturer choice or other reason, the dialing buttons on a standard phone would all be commands).

- **Explanations** - Descriptive information that is not appropriate to include as the label of a state or command, and is more important than a simple group label. This represents an early attempt at including help information within our specifications. Explanations are used rarely in specifications.

Although there are differences between states, commands and explanations, they also share a common property of being enabled. When an object is enabled (or active), the user interface widgets that correspond to that object can be manipulated by the user. Knowing the circumstances in which an object will be enabled or disabled can provide a helpful hint for structuring the interface, because items that are active in similar situations can be grouped, and items can be placed on panels such that the widgets are not visible when the object would not be active. Specifying the prior knowledge of the enabled property is discussed in more detail later in the Dependency Information sub-section.

Label Information

Another common property of appliance objects is the need to specify rich labeling information for flexibility when generating interfaces in different form factors.

To support specifying labeling information, we use the concept of a *label dictionary*. At any place in the specification where a label can be entered, more than one label may be provided. It is expected that all these labels contain the same general information, but vary in terms of length and detail. The interface generator would choose the longest label that fits within the space allocated on the screen.

Labels can be specified for any appliance object, and also be linked with particular values of an appliance state's type.

State Variable Types

Every appliance state has a type object associated with it. The type information is used to determine what kinds of widgets can be used to manipulate the state, and is one of the parameters that are used to recognize Smart Templates.

There are seven different kinds of types that can be used in the specification:

- Boolean
- enumerated
- fixed point
- floating point
- integer
- list-selection
- string

Each of these types has a different set of parameters that can be specified for it.
The Boolean type is for variables that have a value of true or false.

The enumerated type is for small collections of values that all have some string label. Internally, these values are represented by numbers starting with 1. For example, an enumerated type with 4 items can have a value of 1 through 4. Enumerated types must have labels defined for each of their values.
The types of fixed point, floating point, and integer all contain numeric values. Integers do not have a decimal component, while fixed point and floating point both do. Fixed point values have a fixed number of digits to the right of the decimal point, as defined by the required "Decimal Places" field. Floating point values have an arbitrary number of digits on either side of the decimal point. The fixed point and integer types also have an optional increment field that can be used to further restrict the values that the state variable may contain. If an increment is specified, then a minimum value must also be specified. When these parameters are specified, the value of the state variable must be equal to the minimum + n * increment, where n is some integer.

The list selection type is a special type for linking the value of a state variable to a selection within a list that is somewhere in the appliance description. Usually this will be used be when the user has configured a set of common values in a setup portion of the interface, and then wants to select one of those values elsewhere in the interface. You must specify the name of the list that this variable will select from. Optionally, you may also specify some dependencies on the values of that list in order to restrict the list items that may be selected.
The string type contains a string value. Currently there are no parameters for this type.

The Group Tree

Proper structure is a very important part of any user interface. In our language we use a hierarchical "group tree" to specify structure. The leaf nodes in the tree are appliance objects, and the branch nodes are groups. Each node in the tree has a name which must not be the same as any other child of its parent. Thus each node has a locally unique name, and a globally unique name can be constructed by pre-pending the names of all a nodes parents. We use a "." character to separate each name. For example, the locally unique name of a state variable might be "PlayState" but the globally unique name would be "Stereo.CD.PlayState". That same specification could also contain other states like "Stereo.Tape.PlayState". Note that the root name "Stereo" cannot be re-used as a name anywhere in the specification.

It is often necessary to explicitly refer to state variables in a specification. This may be done using any name that starts with a globally unique name. Since the root name is unique, the full name will always work. For the above examples, "Tape.PlayState" and "CD.PlayState" would also work, assuming that there are no other groups or states named "CD" or "Tape".

There are three types of groups: normal, list, and union. A normal group is used for putting related data together, similar to a record in a programming language. List and union groups have special behavior beyond that of normal groups, which are discussed in the next section.

Lists

Specifying a list group is similar to specifying an array of records in a programming language, and multiple list groups can be nested to create multi-dimensional lists. Each list group has an implicit length state variable (named "Length") that always contains the current length of the list. If this variable is undefined, then the list currently has no members. The specification may define bounds on the length of the list in order to help the interface generator create a better rendering. An exact size may be specified, or a minimum and/or maximum size may be specified.

List groups also maintain an implicit structure to keep track of one or more list selections. The number of selections allowed may be defined in the specification ("one" and "many" are the only options currently), and the default is one if nothing is specified. If a list allows only one selection, then an implicit "Selection" variable is created which contains the index of the current selection (undefined means no selection). If multiple selections are created, then an

implicit list group named "Selections" is created. This group contains a "Length" state (as all list groups do) and a "Selection" state which contains all of the selected indices.

Unions

A union group is similar to specifying a union in a programming language; of the children within a union (either groups or appliance objects), only one may be active at a time. An implicit state variable named "ChildUsed" is automatically created within a union group that contains the name of the currently active child.

Dependency Information

Dependency information is specified for each appliance object as a Boolean equation. This information gives the interface generator some approximate a priori knowledge of when the object will be enabled (see the Appliance Objects section above for more information).

Five kinds of dependencies can be specified. Each of these dependencies specifies a state that is depended upon, and a value or another state variable to compare with.

- **Equals** - True when the specified state has the specified value.
- **GreaterThan** - True when the specified state has a value greater than the specified value.
- **LessThan** - True when the specified state has a value less than the specified value.
- **Defined** - True when the specified state has any value.
- **Undefined** - True when the specified state does not have any value.

These dependencies can be composed into Boolean formulas using AND and OR. NOT may also be used.

Specifying dependencies on list state variables must be done using the special **apply-over** operation. This element applies the dependencies it contains over some items in a list and returns a value depending on the value of its *true-if* property. The dependencies are applied over the set of items based on the *items* property, which may be set to *all* or *selected*. The *true-if* property may be set to *all*, *any* or *none*. If *true-if* is *all*, then all of the dependencies contained in the **apply-over** element must be true for elements if true is returned. If *true-if* is *any*, then all of the dependencies contained in the **apply-over** must be true for one of the

elements. If *true-if* is *none*, then the dependencies must not be satisfied by any element to return true.

Smart Templates

Smart Templates are standardized in advance, so that specification authors can specify high-level conventions that the interface generators will understand. A number of templates have already been defined, as described in the Smart Templates Appendix at the end of this document.

To use a Smart Template in a specification, the author must do two things:

Tag a group or appliance object with the name of the Smart Template.

Ensure that the group or appliance object conforms to the restrictions that have been specified for that Smart Template. The appendix at the end of this document describes the restrictions for each Smart Template.

You may find that the Smart Templates that we have already defined are not expressive enough to use with the appliance that you are specifying. If you encounter this situation, please mark the group or object with the appropriate smart template name but specify contents for the template that match the functionality of the appliance. This will help us improve and extend our Smart Templates in the future.

2. Authoring an Appliance Specification with XML

Extensive documentation for the XML language is available on the web here:

http://www.pebbles.hcii.cmu.edu/puc/specification.html

Documentation on Smart Templates is available here:

http://www.pebbles.hcii.cmu.edu/puc/highlevel-types.html

XML Editors

Using a special XML editor will make authoring specifications a much easier process. We support two editors in this study: XMLSpy Home Edition and Visual Studio .NET 2003. You may use a different editor, but we will not necessarily be able to help you if you have problems.

An important reason to use an XML editor is that you can ensure that your specification has the proper format by validating it against a schema. To use this feature of your XML editor, you will need to download the schema for the PUC specification language, which is available here:

http://www.pebbles.hcii.cmu.edu/puc/puc.xsd

We strongly recommend using the schema with your editor, as it guarantees that your document will be correctly formatted.

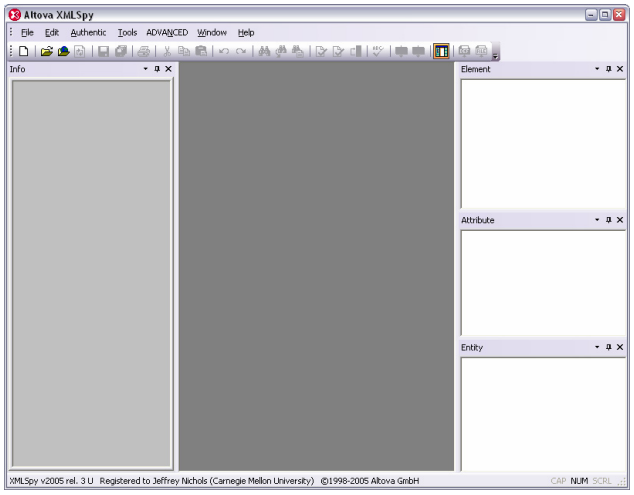
XMLSpy Home Edition

XMLSpy Home Edition can be freely downloaded from the following link:

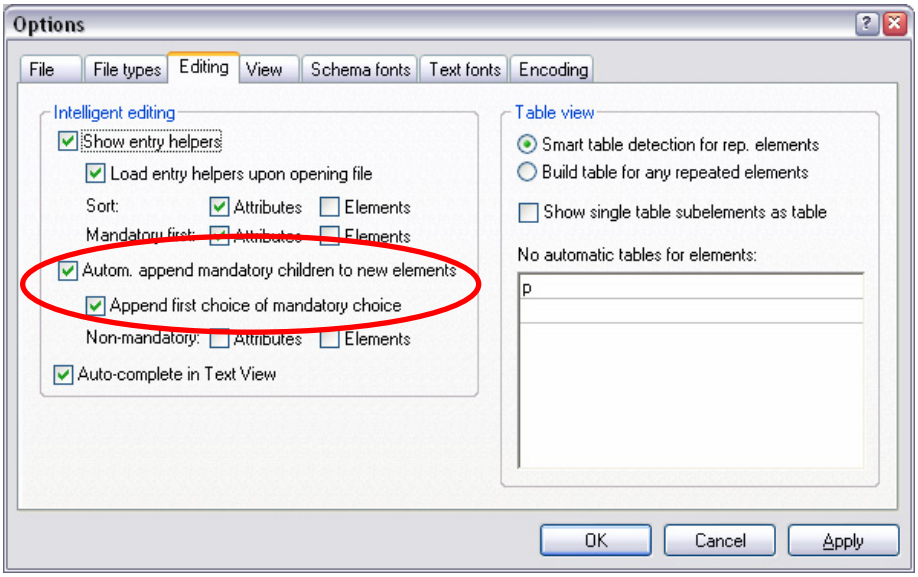
http://origin.altova.com/download_components.html

Near the top of this page you will find links for downloading the installer and requesting a free license.

Once you have the XMLSpy application installed and running, you should screen like this:

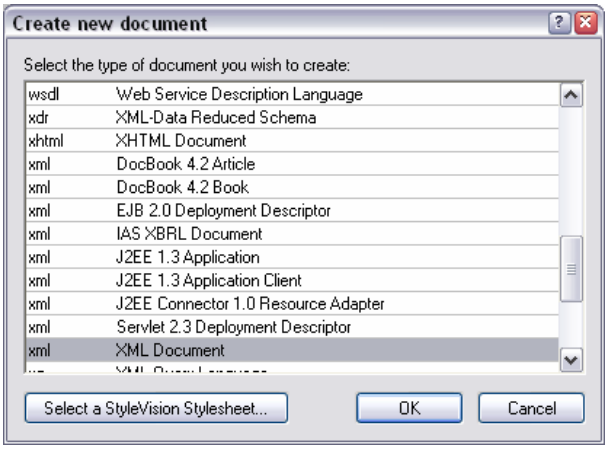


Before you create a new file, open the Tools menu at the top of the page and select the Options… item. A dialog box will open containing a set of seven tabbed panes. Click on the tab for the "Editing" pane. Once you have done this, you should see the following screen (except the red circle):

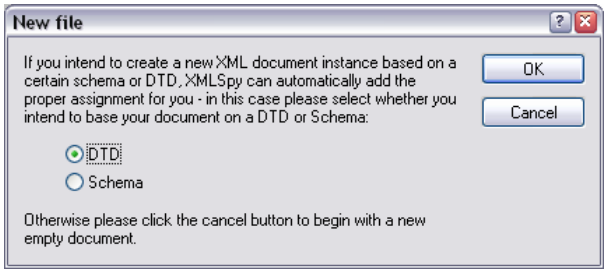


I recommend that you uncheck the "Autom. append mandatory children to new elements" checkbox (circled in red). This feature of XMLSpy seems to cause some curious behavior when used with the PUC language schema. Click OK to exit the dialog box.

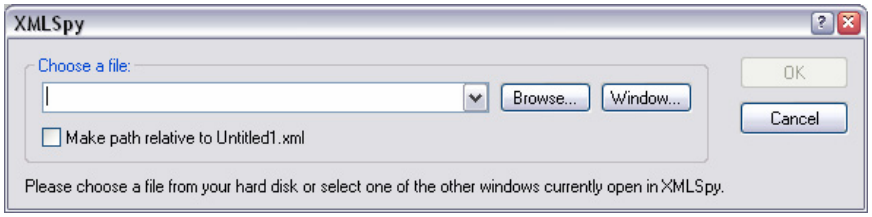
Now you can create a new appliance specification by opening the "File" menu and selecting "New…" The following dialog box will be displayed:



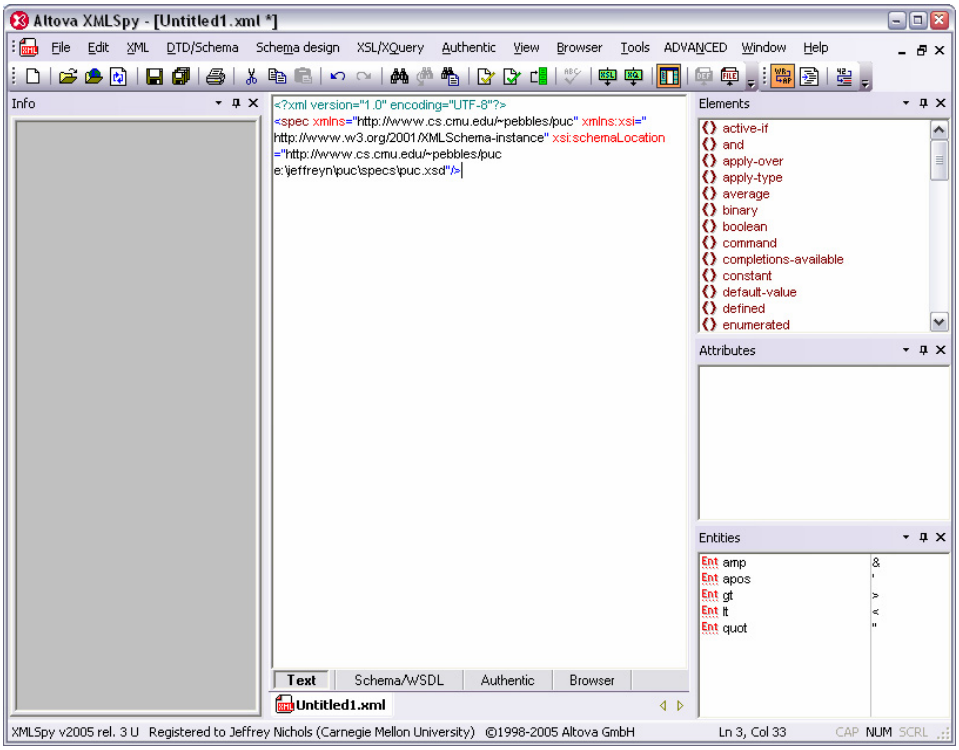
Be sure that "XML Document" is selected and then click "OK." You will then see this dialog box:



Be sure that you have selected "Schema" and then click "OK." If you have not yet downloaded the PUC schema (see the link above), you should do so now and note the location in which you save the file. You will now see the following dialog box:

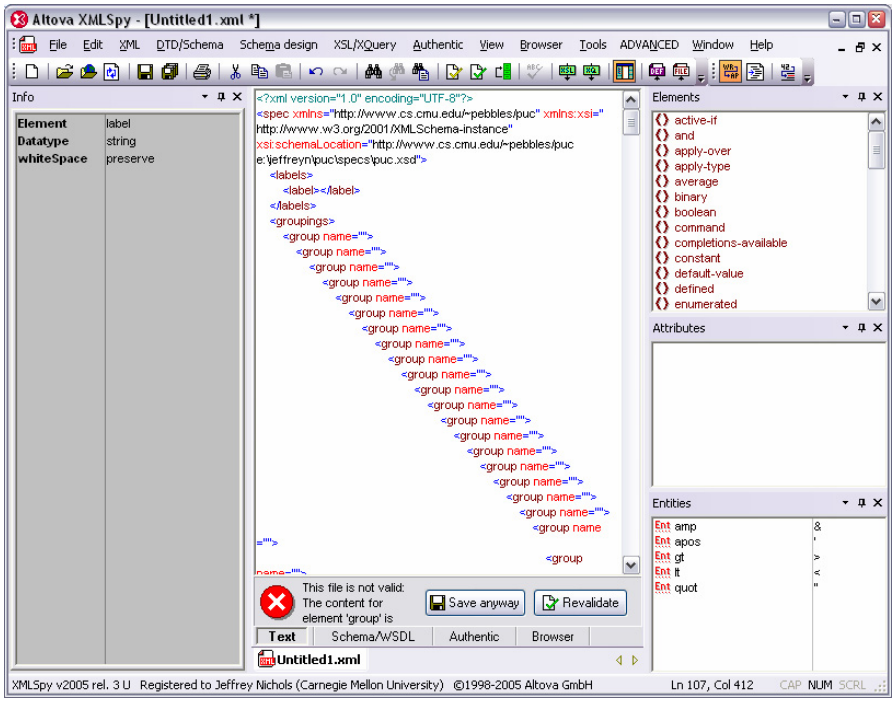


Click the "Browse…" button and find the PUC schema file on your local hard drive. Click "OK" to create the new file. Your application screen should look something like this:



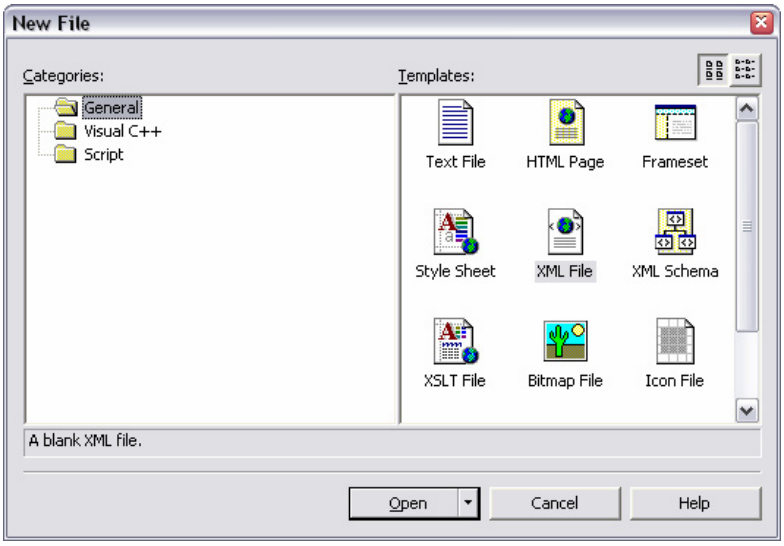
You may now start writing your specification.

If you see a screen that looks more like the image below, then you should repeat the instructions above for turning off the “automatic append mandatory children” feature.



Visual Studio .NET 2003

To create a new XML document in Visual Studio, open the "File" menu and the "New…" sub-menu, and then select "File…" item. The following dialog box will be displayed:



Select “XML File” in the right pane and click “Open.” A new XML file will be created in Visual Studio.

To validate your XML file with the PUC schema, you will need to copy the PUC schema file to a particular location in the Visual Studio directory structure and then add a specific statement to the top of your new file.

The schema file (see the link above) must be placed in:

<Visual Studio Directory>\Common7\Packages\schemas\xml

Where <Visual Studio Directory> is the location of your Visual Studio installation. Typically this is C:\Program Files\Microsoft Visual Studio .NET 2003, but it may vary depending on your installation.

Once you have copied the schema file, you must start your specification with the following line:

```
<spec xmlns=”http://www.cs.cmu.edu/~pebbles/puc” name=”SpecNameHere” version=”PUC/2.3”>
```

The name attribute may be filled in with an appropriate name of your choice. The xmlns attribute is the most important, as it defines to Visual Studio the schema that you want to use for validation. To verify that Visual Studio has appropriately linked your file to the schema, open the “XML” menu and select the “Validate XML Data” item. If you see the following error, then the schema has not been found:

```
Visual Studio could not locate a schema for this document. Validation can only ensure this is a well formed XML document and cannot validate the data against a schema.
```

Double-check that you have copied the schema file to the appropriate directory above. If everything appears to be correct, then try restarting the Visual Studio application. If you still have problems, please contact me.

3. Example Specification Walkthrough

Now we will walk through the design of a specification for a hypothetical to-do list application. Note that while this example uses one process for building a specification, you do not need to mimic this process when designing your own specifications. We do not know what the best process is for writing specs, and part of this study is exploring different methods and their outputs.

The to-do list application has the following functionality:

- List of to-do items
- Detailed information about each item, including a description, category, completion date, and completion flag
- Add and delete functions for to-do list items
- Sorting functionality for to-do list on category, completion date, or completion flag
- A customizable list of categories
- Add and delete functions for category list items
- Reminder frequency setting which may be weekly, daily, or hourly
- Display preference for choosing whether or not to show completed tasks

For a regular appliance, you should be able to discover many more details about each of the various functions than I have presented above. It is important that you explore the features through the user manual and using the appliance so that you can ensure that your specification is as accurate as possible.

Defining a State Variable

We will start by specifying state variables for the reminder frequency and display preference. These are variables because each has a value that can be manipulated by the user. You could choose to have separate commands for these settings, but in general it is better to use a state variable wherever possible.

For the card-based method, we start by defining a state variable with an object card. Figure D.1a shows the object card for the reminder frequency state variable. Because this object card

defines a state variable, a state variable type card must be attached to specify the type of the state variable (shown in Figure D.1b). The reminder frequency can be set to three different values, so I have picked an enumerated type for this state that contains three items. Labels must be specified for each of the values of the enumerated type, which is done by attaching three label cards to the type card (see Figure D.1c).

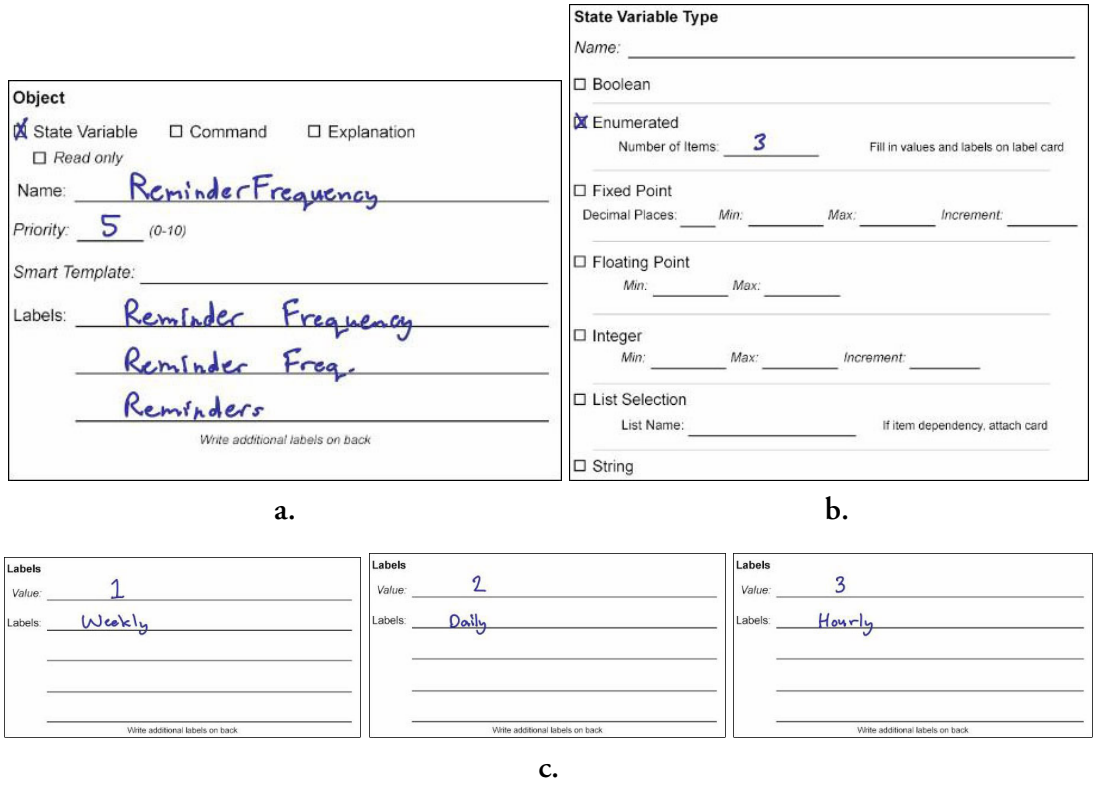


Figure D.1. The cards that specify the reminder frequency setting. All of these cards would be attached together with a paper clip.

You may have noticed that the object card shown for the reminder frequency setting has a priority value of "5" specified for it. You may wonder where this value came from, especially because priority values are chosen relative to the other members of the group that contains this state variable. This value was chosen much later in the specification process, after I knew the other items that would be in the same group as these state variables. In most cases, it will be difficult to assign a priority value when you are initially defining an object because the contents of the group that contains that object will not be known.

The XML for this state variable has all of the same components as the card, formatted according to the schema (see Figure D.2).

```
<state name="ReminderFrequency" priority="5">
    <type>
        <enumerated>
            <item-count>3</item-count>
        </enumerated>
        <value-labels>
            <map index="1">
                <labels>
                    <label>Weekly</label>
                </labels>
            </map>
            <map index="2">
                <labels>
                    <label>Daily</label>
                </labels>
            </map>
            <map index="3">
                <labels>
                    <label>Hourly</label>
                </labels>
            </map>
        </value-labels>
    </type>
    <labels>
        <label>Reminder Frequency</label>
        <label>Reminder Freq.</label>
        <label>Reminders</label>
    </labels>
</state>
```

Figure D.2. The XML code for the reminder frequency state variable.

The display preference setting is also a state variable, and has a definition that is very similar to that shown for the reminder frequency variable. As an exercise, I suggest writing your own description of this variable and then comparing it to my cards or XML shown in Appendix D.2.

Specifying List Data

Lists are specified using the special list group feature, which is specified on the group card (see Figure D.3) or with the <list-group> element in XML (see Figure D.4). I have decided

that the list does not have any bounds and there will be one selection, and these decisions are reflected in the descriptions that you see below. Note that in the card-based system, a list parameter card must be attached to the group card to define the bounds and selections.

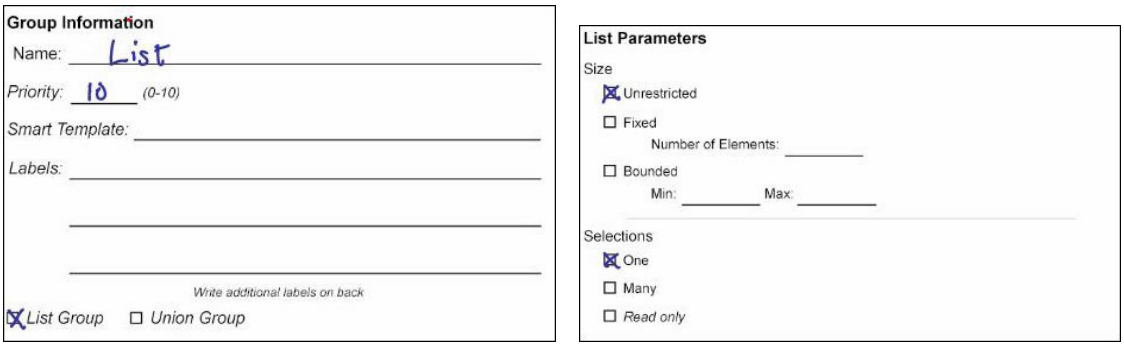


Figure D.3. The group and list parameters card that describe the to-do list. This list has four state variable members (not shown).

```
<list-group name="List" priority="10">
   <selections number="one"/>

   ...
</list-group>
```

Figure D.4. The XML code for describing the to-do list. The four state variable are defined where the "..." appears in the above code.

Each of the variables contained within a list group has multiple values; one value for each item in the list. If you are familiar with programming languages, this is similar to defining an array of records or structs. Each to-do list item has four values (description, completion date, category, and completion flag) so this list group will contain four different state variables. The specification of the description, completion date, and completion flag variables is left as an exercise, but the category variable is discussed in the next sub-section.

There is also another list in the to-do list application, which contains each of the valid categories that a to-do list item may have. As an exercise, I suggest that you attempt to specify this list group, including its state variable(s). You can check your work by referring to the completed specifications in Appendices B & C.

Using the List Selection Type

The categories variable in the to-do list has a value that is one of the items in the user-configurable categories list. This means that we cannot use an enumerated type for this variable, because we do not know in advance what categories the user will choose.

Our language supports this problem with the list-selection state variable type. In this case, we can create a category variable within the to-do list group that has a list-selection type. The type will specify that it selects an item from the category list, which is elsewhere in the specification. When the user changes the category list, the changes will automatically appear when they later choose to modify their to-do list items. The cards and XML for the category state variable are shown in Figure D.5 and Figure D.6 respectively.

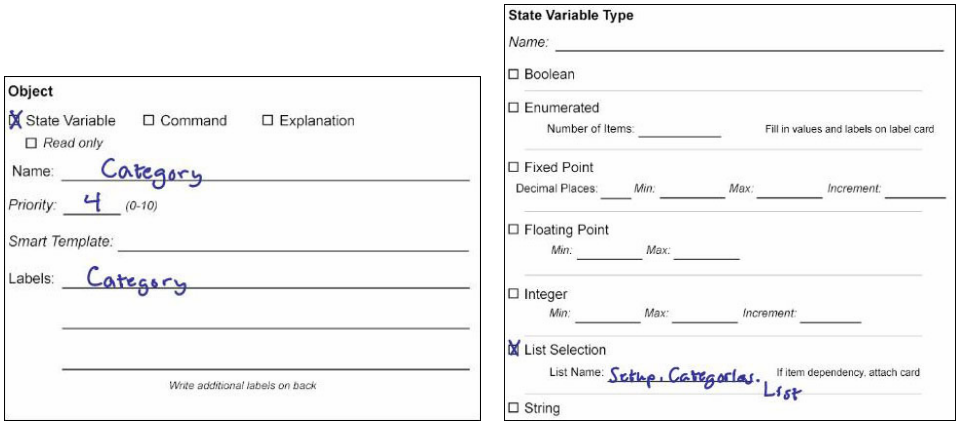


Figure D.5. The object and state variable type cards for the Category state variable that is contained in the to-do list group. Note the use of the list-selection type on the state variable type card.

```
<state name="Category" priority="4">
    <type>
        <list-selection list="Setup.Categories.List"/>
    </type>
    <labels>
        <label>Category</label>
    </labels>
</state>
```

Figure D.6. The XML code for the Category state variable that is contained in the to-do list group. Note the use of the list-selection type in this variable.

Providing a User Interface for Modifying List Data

The description of the to-do list application includes functions for adding and removing items from both the to-do list and the categories list. In our description language, we have chosen to use a Smart Template for defining these special functions and allowing the interface generators to take special action when rendering them.

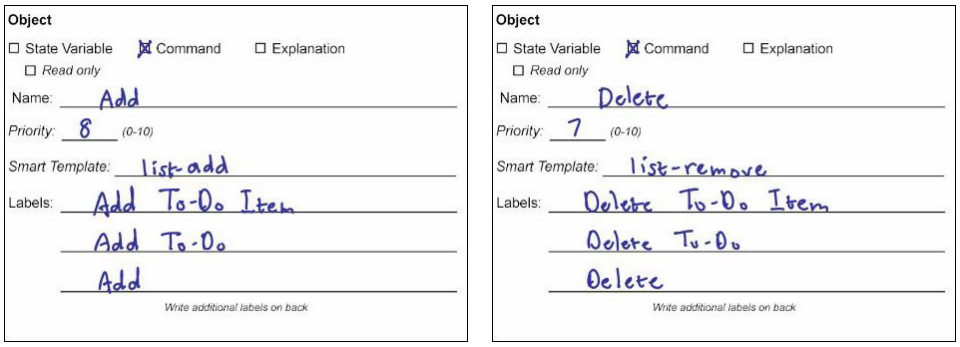


Figure D.7. The object cards that describe the add and remove commands for the to-do list. These objects are contained within a group that is tagged with the "list-commands" Smart Template.

```
<group name="Commands" is-a="list-commands" priority="10">
   <command name="Add" is-a="list-add" priority="8">
      <labels>
         <label>Add To-Do Item</label>
         <label>Add To-Do</label>
         <label>Add</label>
      </labels>
   </command>

   <command name="Delete" is-a="list-remove" priority="7">
      <labels>
         <label>Delete To-Do Item</label>
         <label>Delete To-Do</label>
         <label>Delete</label>
      </labels>
      <active-if>
         <greaterthan state="ToDo.List.Length">
            <constant value="0"/>
         </greaterthan>
         <defined state="ToDo.List.Selection"/>
      </active-if>
   </command>
```

Figure D.8. The XML code for describing the add and remove commands for the to-do list. Note the group that contains these items. Also note that dependencies have been provided for the Delete command. This will be discussed in the "Dependency Information" section below.

The Smart Template used in this case is called "list-commands". You may wish to read the description of this template in Appendix D.1 in order to understand its usage. A group that is tagged with this template may contain any number of commands that are tagged with more specialized templates, such as "list-add," "list-move-after" or "list-clear." In this case, we only wish to add commands for adding and removing items from the list, so we will add two commands that implement the "list-add" and "list-remove" templates. The cards for these commands are shown in Figure D.7 and the XML is shown in Figure D.8.

You should note that the list-XX smart templates do not have restrictions on the labels or dependencies of these commands, so this information can vary depending on what is being described. If there were two ways to add data to this list for example, then there could be two commands implemented the list-add Smart Template, each with different labels. Even if there is only one command of each type, it is important to fill out all of the information for each object and group because an interface generator might not know about the Smart Template that you have used. If this is the case and you have included all of the information, the generator can still create an interface. If information is missing, then this would not be possible.

Another variable that uses a Smart Template in this specification is the variable for completion date in the to-do list. As an exercise, I suggest that you find the appropriate Smart Template and apply it when specifying this state variable.

Dependency Information

If you looked at XML code for the Delete command above (in Figure D.8), you may have noticed that this command includes dependency information. Specifically, the Delete command is only available if there are more than zero items in the to-do list and an item in the to-do list is selected. The card for this dependency formula (see Figure D.9) should be attached to the object card for the Delete command.

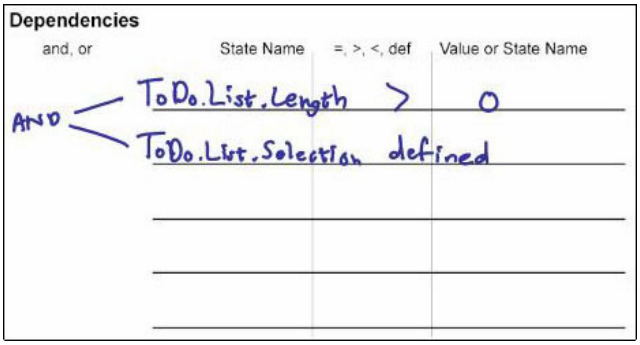


Figure D.9. The dependency card for the Delete command. Note that extended names are used here for the state variables. You may find it necessary to go back and make names more explicit on dependency cards after you have organized the variables in your specification.

Organizing the Specification

Once all of the state variables, commands, and lists have been defined, you should consider further organizing your description to make it more useful for an interface generator. In general, you should try to make the hierarchy as deep as possible so that an interface generator for a small screen device will be able to make intelligent decisions about how to separate the pieces of the user interface.

In this specification, I chose to have two main groups. There is a group that contains the to-do list, which consists of the list containing the to-do list data and the group of list commands for modifying that data. The group of list commands also contains the sorting function. The second main group is called "Setup," and contains the display preference setting, the reminder frequency setting, and the list of categories.

You will find in your specifications that you have many more functions than were included in this to-do list application, which will required much more organization. As we mentioned above, organization data is very important to the interface generator and may affect the quality of an interface generated from your specification. You should be sure to put some effort into designing an interface hierarchy that is accurate and intuitive for your appliance.

Appendix D.1. Smart Templates

The following Smart Templates have been defined:

- address
- date
- date-time
- four-way-dpad
- four-way-dpad-with-enter
- list-commands
- media-controls
- time-absolute
- time-duration
- zoom-controls

The format for each of these templates is described below.

address

Overview

Represents an address, like might be entered into a navigation system.

Contents

This template only supports the Multi-State form. In the future, it might make sense to support a single state string form that contains a parsable address, but this is not required now.

There are no required states in this template. This allows the template to be used in a number of situations where a partial piece of an address is needed, such as zip code by itself. The different states in this template are:

`StreetName`

> This state represents the name of the street. It may have a string or enumerated type. It must support the use of completions and server-side error correction, if either of those features are available.

`ApartmentNumber`

> This state represents the number of an apartment or suite at this address. This state may have a string or integer type. When the integer is used, it may have type restric-

tions such as bounds or an increment. The labels of this state must be taken into account in order to know whether the number is for a suite, apartment, etc.

`StreetNumber`

This state represents the street number of the address. It must have an integer type, but restrictions such as bounding are optional. It must support the use of completions and server-side error correction, if either of those features are available.

`City`

This state represents the city of the address. It must have a string or enumerated state. It must support the use of completions and server-side error correction, if either of those features are available.

`State`

This state represents the state or province in which the address is located. It may have a string or enumerated type.

`ZipCode`

This state represents a zip code in the address. It may have a string, integer, or enumerated type. It must support the use of completions and server-side error correction, if either of those features are available.

`Country`

This state represents the country of the address. It may have a string or enumerated type. It must support the use of completions or server-side error correction, if either of those features are available.

date

Overview

This template describes data that stores a date. This might be used by an appliance to store and display the current date, or to record a date in the future when some pre-defined action will be taken. This template currently supports both single state and multiple state instantiations.

Contents

The contents of this Smart Template may be represented in several different ways. One state with the string type may be used, or multiple states may be used. For the single variable case there are the following type restrictions:

String

The state variable must have a string type. The value of the state variable must be a parseable date string in one of the ISO 8601 international standard formats.

If multiple states are used to represent the `date`, they must have the following form:

`Month`

This state may have an integer type with bounds from 1-12, or an enumerated type with 12 items.

`Day`

This state must have an integer type ranging between 1 and 31. The constant maximum may also point to a state variable that gives the proper maximum for the given month. If maximum is set to a constant value of 31, then the user interface will enforce the proper maximum for the given month and year.

`Year`

This state must have an integer type. Bounds on this variable are optional. Negative values in this state variable will be interpreted as years BC, and positive years as AD. Increments may also be specified for this state.

date-time

Overview

This template combines a date template with a time-absolute template to form a complete representation for dates and times together.

Contents

The contents of this Smart Template may be represented in several different ways. One state with the string type may be used, or the template may contain two groups that separately instantiate the date and time-absolute smart templates. If one state variable is used, it has the following restrictions:

String

The state variable must have a string type. The value of the state variable must be a parseable date-time string in one of the ISO 8601 international standard formats.

If two groups are used, then one of those groups must be tagged with `date` and the other must be tagged with `time-absolute`. These groups must conform to the restrictions for these other Smart Templates.

four-way-dpad

Overview

This template represents a four-way directional control.

Contents

This template may only be applied to a group that includes four commands. The commands must be:

`Left`
: The command for the left button in the directional control.

`Right`
: The command for the right button in the directional control.

`Top`
: The command for the top button in the directional control.

`Bottom`
: The command for the bottom button in the directional control.

four-way-dpad-with-enter

Overview

This template represents a four-way directional pad with an enter function.

Contents

This template must include the `four-way-dpad` smart template and one extra command:

`Enter`
: The enter command

list-commands

Overview

This template is for commands that manipulate list data, such as adding, deleting, or moving list items. This is the first example of a *nested* template, which means that the template must contains other templates. In this case, there is a `list-commands` template which may only contain `list-add`, `list-remove`, `list-clear`, `list-move-after`, or `list-move-before` templates.

One tricky aspect of the `list-commands` template is linking each template instance to the list that it manipulates. This is done by requiring that the template be at the same level and immediately precede the list group that it manipulates. We suggest that each pair of commands and list group be placed in their own group, but this is not required.

Contents

The `list-commands` template only supports the multi-state form. The constituent templates `list-add`, `list-remove`, `list-clear`, `list-move-after`, and `list-move-`

`before` must be single commands. Constituent templates may be used outside of the `list-commands` template.

As mentioned above, any instance of the `list-commands` template must be placed at the same level of the specification tree immediately preceding the list-group that the command modify. This requirement is also true of any of the consituent templates when used outside of the `list-commands` template.

The following are descriptions of the constituent templates:

`list-add`
: Commands with this template should add a new item to the list. The server upon receiving the command should insert an item into the list at the appropriate location in the list. The template implementation can automatically display a dialog box in this case, or it may rely on the required-if infrastructure to determine whether an editing dialog box should be displayed.

`list-remove`
: Commands with this template should delete an item from the list, preferably the currently selected item(s).

`list-clear`
: Commands with this template should remove all items from the list.

`list-move-after`
: Commands with this template should move the currently selected item(s) to the next higher index.

`list-move-before`
: Commands with this template should move the currently selected item(s) to the next lower index.

media-controls

Overview

This template represents the interactions that control the playback of any audio/visual media, such as a CD, MP3, or VHS tape. This template supports either state- or command-based representations of the controls, and also handles related functions such as *Next Track* and *Previous Track*.

Contents

This Smart Template may represent the controls as a single state variable and/or several commands. The state-based representation must have an enumerated type with standardized labels mapped to each value in the type in the `valueLabels` section. The state must have the name `Mode`. The labels are described below. Other labels may be included in addition to the standardized ones for the benefit of interface generators that don't recognize this Smart Template.

`Stop`
: This label identifies the state when the media is stopped.

`Play`
: This label identifies the state when the media is playing.

`Pause`
: This label identifies the state when the media is paused.

`Rewind`
: This label identifies the state when the media is rewinding.

`Fast-Forward`
: This label identifies the state when the media is fast-forwarding.

`Record`
: This label identifies the state when media is being recorded.

If no state information is available from the appliance, a command-based description may be used instead. Such a representation must include one or more of the following commands. These commands should **not** be included if the `Mode` state is included in the description. Allowable commands are:

`Play`
: When this command is activated, the appliance should begin playing.

`Stop`
: When this command is activated, the appliance should stop playing.

`Pause`
: When this command is activated, the appliance should pause playback.

`Rewind`
: When this command is activated, the appliance should begin rewinding.

`FastFwd`
: When this command is activated, the appliance should begin fast-forwarding.

`Record`
: When this command is activated, the appliance should begin recording.

In addition to having either the state or a set of commands, additional states and commands may be specified that will be integrated with the rest of the playback controls.

Commands

`NextTrack`

When this command is activated, the next track should be selected.

`PrevTrack`

When this command is activated, the previous track should be selected.

NOTE: Other playback modes may be possible and should probably be considered. These include:

- Reverse Play
- Fast-Forward while Playing vs. while Stopped
- Rewind (same thing)
- Different speeds of fast-forward and rewind
- Play New (for answering machines)

time-absolute

Overview

This template specifies absolute time, i.e. time-of-day. This includes the time value along with parameters of the time such as the time zone, 12/24 hour mode, etc.

Contents

This template supports both the single and multi-state specification methods. The single state method may only represent an absolute time value in 24-hour units. This value may be rendered as a 12-hour time value with AM/PM depending on the configuration of the user's device. The multi-state method may also include parameters for the time value, as described below.

The single state form may have one of four primitive types:

Integer

The value of the state variable should contain the number of seconds since midnight. No increment is allowed.

String

The value of the state variable must be of the form: `hh:mm:ss` or `hh:mm`, where h = hours, m = minutes, and s = seconds. Note that unlike the `time-duration` template, digits may be omitted if they are not significant. In other words, `1:45:56` has the same meaning as `01:45:56`. An arbitrary number of digits may be used for representing the fractions of a second.

If multiple states are used to represent the `time-absolute` template, then they must have the following form:

`Time`

This state specifies the time. It may have any of the types listed above for the single-state format. Either this state, or some combination of the hour/minute/etc. states below may be included, but not both.

`Hours`

This state must have an integer type. The minimum and maximum must be 0-23. This state is required if the `Time` state is not specified, and may not be included if the `Time` state is defined.

`Minutes`

This state must have an integer type ranging from 0 to 59. This state must be specified if the `Time` state is specified, and may not be included if the `Time` state is defined.

`Seconds`

This state must have an integer type ranging from 0 to 59. This state must not be included if the `Time` state, but is otherwise optional.

`TimeZone`

This state optionally specified the time zone. It must have an enumerated type where each value has at least one label of the form *GMT-X* or *GMT+X*, where X specifies the number of hours from Greenwich Mean Time. The labels should also include common names for each time zone (such as "Eastern" for GMT-5) to ensure that generators not equipped with this Smart Template can still render an understandable interface.

`DaylightSavings`

This state optionally specifies whether it is currently daylight savings time. This state must have a boolean type.

`HourMode`

This state optionally specifies whether the time is specified in 24 or 12 hour mode. The state must have a boolean type with labels for each value. One value must be la-

beled as "12Hr" and the other as "24Hr". If this state exists, it should be linked to the rendering for the time value (if included).

Increments may be specified **only** for the least significant unit. For example, if a particular template defines an hours state and a minutes state then an increment may be specified only for the minutes state. Another template that defines all four of the possible states may only have an increment specified for the fraction state.

time-duration

Overview

This template describes data that stores a duration of time. This could be used by media player devices to describe the length of a song or the current playback point in a song, or by microwaves to display the amount of cooking time that remains. This template supports resolutions in the fractions of a second, seconds, minutes and hours. The fractions of a second resolution is purposely left ambiguous and may be defined by the specification designer by defining a range for the state. Milliseconds might be the fraction used for a timer application, whereas frame number might be used by a VCR or other video application.

Contents

The contents of this Smart Template may be represented in several different ways. One state with one of four primitive types may be used, or multiple states with an integer type may be used. If one state is used, the Smart Template must be applied directly to that state (see Example #2 above). The allowable types are:

Integer

The value of the state variable should contain the number of seconds in the time duration.

Fixed Point

The pointpos may be set as necessary. The value of the state variable should contain the number of seconds in the time duration. The decimal component represents the fractions of a second that have elapsed.

Floating Point

The value of the state variable should contain the number of seconds in the time duration, with the decimal component representing the fractions of seconds in the duration.

String

The value of the state variable must be of the form: `hh:mm:ss:fff`, `mm:ss:fff`, `hh:mm:ss`, or `mm:ss`, where h = hours, m = minutes, s = seconds, and f = fractions of a second. Note that it is **not** optional to omit digits if they are not significant. In other words, `1:45:56` is not valid, but both `01:45:56` and `01:45:560` are. Also note that three digits are required for fractions of second component, if it is used.

If multiple states are used to represent the `time-duration`, they must have the following form:

`Hours`

This state must have an integer type. A minimum and maximum may be optionally specified.

`Minutes`

This state must have an integer type ranging between 0 and 60 if the `Hours` state is specified. Minimum and maximum are optional if `Hours` is not specified. This state must be included if both the `Hours` and `Seconds` states are specified.

`Seconds`

This state must have an integer type ranging between 0 and 60 if the `Minutes` state is specified. Minimum and maximum are optional if `Minutes` is not specified. This state must be included if both the `Minutes` and `Fraction` states are specified.

`Fraction`

This state must have an integer type. The ranges may be specified to fit the type of fraction being used.

Increments may be specified **only** for the least significant unit. For example, if a particular template defines an hours state and a minutes state then an increment may be specified only for the minutes state. Another template that defines all four of the possible states may only have an increment specified for the fraction state.

zoom-controls

Overview

This template represents controls for zooming something on the appliance.

Contents

This template may include two commands, only one of which must be included:

`In`

The zoom in command.

`Out`

The zoom out command.

Appendix D.2. Complete XML Specification for To Do List Application

```
<?xml version="1.0" encoding="utf-8" ?>
<spec     xmlns="http://www.cs.cmu.edu/~pebbles/puc"     name="ToDoApp"     ver-
sion="PUC/2.2">

    <!--
        Labels for specification
    -->
    <labels>
        <label>PUC To-Do List Application</label>
        <label>To-Do List App</label>
        <label>To-Do List</label>
    </labels>


    <!--
        Groups
    -->
    <groupings>
        <group name="ToDo" priority="10">
            <labels>
                <label>To-Do List</label>
                <label>List</label>
            </labels>

            <list-group name="List" priority="10">
                <selections number="one"/>

                <state name="Completed" priority="5">
                    <type>
                        <boolean/>
                        <value-labels>
                            <map index="true">
                                <labels>
                                    <label>Done</label>
                                </labels>
                            </map>
                            <map index="false">
                                <labels>
                                    <label>Incomplete</label>
                                </labels>
                            </map>
                        </value-labels>
                    </type>
                    <labels>
                        <label>Completed</label>
                    </labels>
                </state>
```

```
    <state name="Category" priority="4">
        <type>
            <list-selection list="Setup.Categories.List"/>
        </type>
        <labels>
            <label>Category</label>
        </labels>
    </state>

    <state name="Description" priority="8">
        <type>
            <string/>
        </type>
        <labels>
            <label>Description</label>
            <label>Desc.</label>
        </labels>
    </state>

    <state name="CompletionDate" is-a="date" priority="3">
        <type>
            <string/>
        </type>
        <labels>
            <label>Finish By</label>
            <label>Due Date</label>
            <label>Due</label>
        </labels>
    </state>
</list-group>

<group name="Commands" is-a="list-commands" priority="10">
    <command name="Add" is-a="list-add" priority="8">
        <labels>
            <label>Add To-Do Item</label>
            <label>Add To-Do</label>
            <label>Add</label>
        </labels>
    </command>

    <command name="Delete" is-a="list-remove" priority="7">
        <labels>
            <label>Delete To-Do Item</label>
            <label>Delete To-Do</label>
            <label>Delete</label>
        </labels>
        <active-if>
            <greaterthan state="ToDo.List.Length">
                <constant value="0"/>
            </greaterthan>
```

```
                <defined state="ToDo.List.Selection"/>
            </active-if>
        </command>

        <state name="SortBy" priority="5">
            <type>
                <enumerated>
                    <item-count>3</item-count>
                </enumerated>
                <value-labels>
                    <map index="1">
                        <labels>
                            <label>Category</label>
                        </labels>
                    </map>
                    <map index="2">
                        <labels>
                            <label>Completion Date</label>
                            <label>Date</label>
                        </labels>
                    </map>
                    <map index="3">
                        <labels>
                            <label>Completed</label>
                        </labels>
                    </map>
                </value-labels>
            </type>
            <labels>
                <label>Sort By</label>
                <label>Sort</label>
            </labels>
            <active-if>
                <greaterthan state="ToDo.List.Length">
                    <constant value="0"/>
                </greaterthan>
            </active-if>
        </state>
    </group>
</group>

<group name="Setup" priority="1">
    <labels>
        <label>Setup</label>
    </labels>

    <state name="DisplayPreference" priority="8">
        <type>
            <enumerated>
                <item-count>4</item-count>
            </enumerated>
```

```
            <value-labels>
                <map index="1">
                    <labels>
                        <label>All Items</label>
                        <label>All</label>
                    </labels>
                </map>
                <map index="2">
                    <labels>
                        <label>Incomplete Items</label>
                        <label>Incomplete</label>
                    </labels>
                </map>
                <map index="3">
                    <labels>
                        <label>Past Due Items</label>
                        <label>Past Due</label>
                    </labels>
                </map>
                <map index="4">
                    <labels>
                        <label>Completed Items</label>
                        <label>Completed</label>
                    </labels>
                </map>
            </value-labels>
        </type>
        <labels>
            <label>Display Preference</label>
            <label>Display Pref</label>
            <label>Display</label>
        </labels>
    </state>

    <state name="ReminderFrequency" priority="5">
        <type>
            <enumerated>
                <item-count>3</item-count>
            </enumerated>
            <value-labels>
                <map index="1">
                    <labels>
                        <label>Weekly</label>
                    </labels>
                </map>
                <map index="2">
                    <labels>
                        <label>Daily</label>
                    </labels>
                </map>
                <map index="3">
```

```
                    <labels>
                        <label>Hourly</label>
                    </labels>
                </map>
            </value-labels>
        </type>
        <labels>
            <label>Reminder Frequency</label>
            <label>Reminder Freq.</label>
            <label>Reminders</label>
        </labels>
    </state>

    <group name="Categories" priority="5">
        <labels>
            <label>Category Setup</label>
        </labels>

        <list-group name="List">
            <state name="Category">
                <type>
                    <string>
                        <min><constant value="1"/></min>
                        <average><constant value="10"/></average>
                        <max><constant value="25"/></max>
                    </string>
                </type>
                <labels>
                    <label>Category</label>
                </labels>
            </state>
        </list-group>

        <group name="Commands" is-a="list-commands">
            <command name="Add" is-a="list-add">
                <labels>
                    <label>Add Category</label>
                    <label>Add</label>
                </labels>
            </command>

            <command name="Delete" is-a="list-remove">
                <labels>
                    <label>Remove Category</label>
                    <label>Remove</label>
                </labels>
            </command>
        </group>
    </group>
</group>
</groupings>
```

```
</spec>
```

APPENDIX E

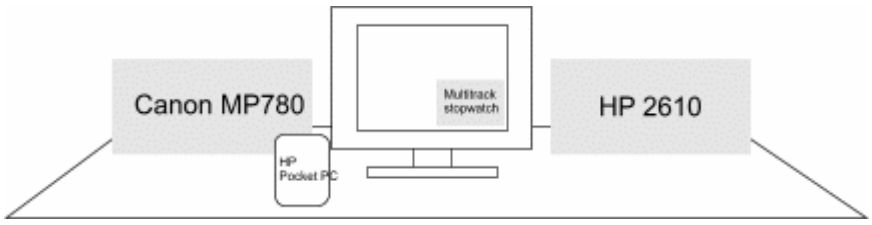
Usability Study Instructions

This appendix contains the informal document describing the process employed in the usablity of the all-in-one printers. The document starts on the next page to preserve formatting.

Conditions

User Group	Sequence in using the interfaces	Number of users
1	physical hp → physical canon	8
2	physical canon → physical hp	8
3	puc hp → puc canon	8
4	puc canon → puc hp	8
5	puc hp → uniform canon	8
6	puc canon → uniform hp	8

Setup of Equipment



Questionnaires

- Background questionnaire: p1-3
- Post questionnaire: p4

Actions during Study

before study	hardware	Move pda/printer in position
		Connect usb cable of pda to desktop
		Point camera at pda/printer
		Warm up printers
		Add more paper

	software	Start **camtasia**
		Start **multitrack stopwatch**
		Start **activsync remote display**
before 1st device 1st task		Start recording in **camtasia**
after 1st device		Save task times in Multitrack stopwatch
		Reset Multitrack stopwatch
After study		Stop recording in camtasia and save video file
		Save task times in Multitrack stopwatch

Script

Our research team is investigating how to create better interfaces for everyday appliances.

Physical: Today, you'll be using two multi-function printers to complete some tasks. You'll be doing a set of 8 tasks using one of the printers, and then you'll do the same set of tasks using the other printer.

Puc->puc & puc->Uniform: Today, you'll be controlling two multi-function printers to complete some tasks. You'll be doing a set of 8 tasks by controlling one of the printers, and then you'll do the same set of tasks by controlling the other printer. You won't be directly interacting with the printers but you'll be controlling them using a handheld computer.

Q: Used a handheld computer before? Multi-function printer? Copier? Fax machine?

The multi-function printers that you're going to use can print, copy, fax and scan, in black and white as well as in color. This study will take about 90 minutes. I'll record audio and video, and the time that you take to complete the tasks. However, all this information will remain anonymous. Before we start, I would like you to sign a consent form, and complete a questionnaire.

Puc->puc & puc->Uniform: Now, we're going to do two exercises. They are just to familiarize you with the interface of the handheld. So you can take your time.

The first exercise is a Pocket PC tutorial. (this handheld computer is called Pocket PC.)

For the second one, I would like you to imagine the interface shown here is the configuration dialog box of a word processor.

Now, we do the actual tasks. You will work on them one by one, and please complete them on your own as I cannot offer help. You have 5 min to complete each task. If you complete a task within the 5 min, you can proceed to the next one. But if at the end of the 5 min, you haven't completed the task, I'll prompt you and ask you to move on. You are required to keep trying within the 5 minutes, until you finish the task or 5 mins has passed. I'll explicitly tell you when you're done with a task -- I'll say "You're done". If I do not say anything, that means you're not done and you should keep trying.

Here's the procedure for each task...

Before each task, I'll ask you to turn around so that you do not look at the screen of the handheld computer. Then, I'll give you a card with the task instruction on it. You can take your time to read the instruction because the time you spend on reading won't count towards the 5 minutes. You can keep the card while you do the task; you can put it on the table. And you can look at it if necessary. And, while you're reading the instruction, I'll reset the interface, so that it will always look the same when you start a new task.

Tasks

Imagine a document has already been properly placed on the device. Make **two black and white** copies of it. Do NOT make one copy at a time.

Imagine you find the copies that the device produces too dark. Improve this by changing one setting of the device. You do not need to produce any actual copy.

The device remembers the current date and time. However, you suspect they are incorrect. Find out at where you can change them. Actual change is not necessary.

Configure the device so that it will print out an error report only when it has problem **receiving a fax** -- it should NOT print out any error report when it has problem sending a fax.

Configure the device so that when faxing it will ALWAYS automatically redial a number that is busy.

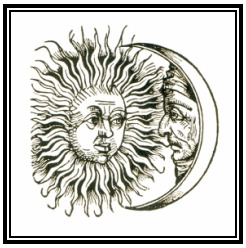
Configure the device so that any received fax larger than the default paper size will fit on one page.

On the device, you can assign speed dials to fax numbers that you use often. There are two types of speed dials. One is called *One-touch Speed Dial*, the other is called *Coded Speed Dial*. You will only use the *Coded Speed Dials* for this task.

Imagine a document has already been properly placed on the device. Fax it using the **third Coded Speed Dial**, in **black and white**.

On the device, you can assign speed dials to fax numbers that you use often. Some speed dials have already been set up for you.

Imagine a document has already been properly placed on the device. Fax it using the **third speed dial**, in **black and white**.



APPENDIX F

Gallery of PUC Interfaces

This appendix contains a number of screenshots of interface generated by the PUC and any interfaces used by the appliance simulators that have been created.

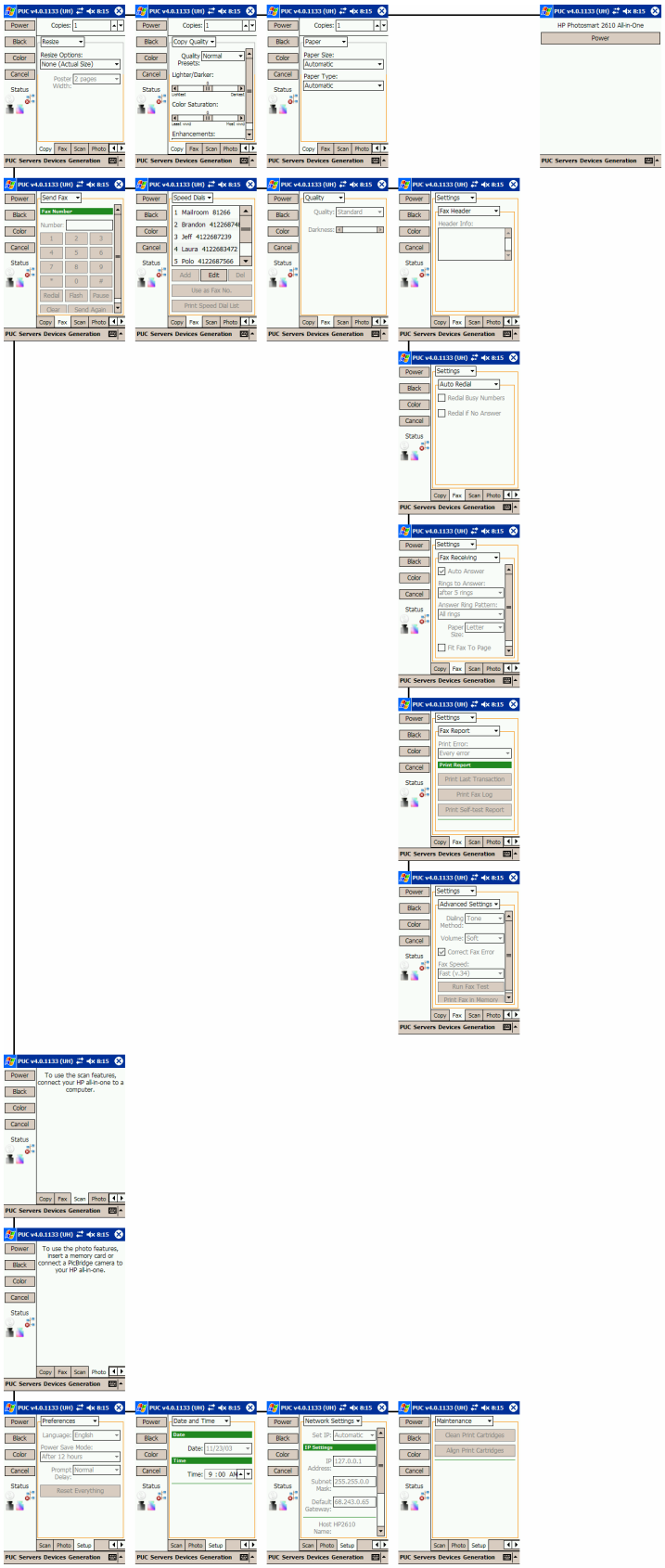


Figure F.1. The full interface for the HP All-In-One printer generated without consistency, as seen by users in the usability study.

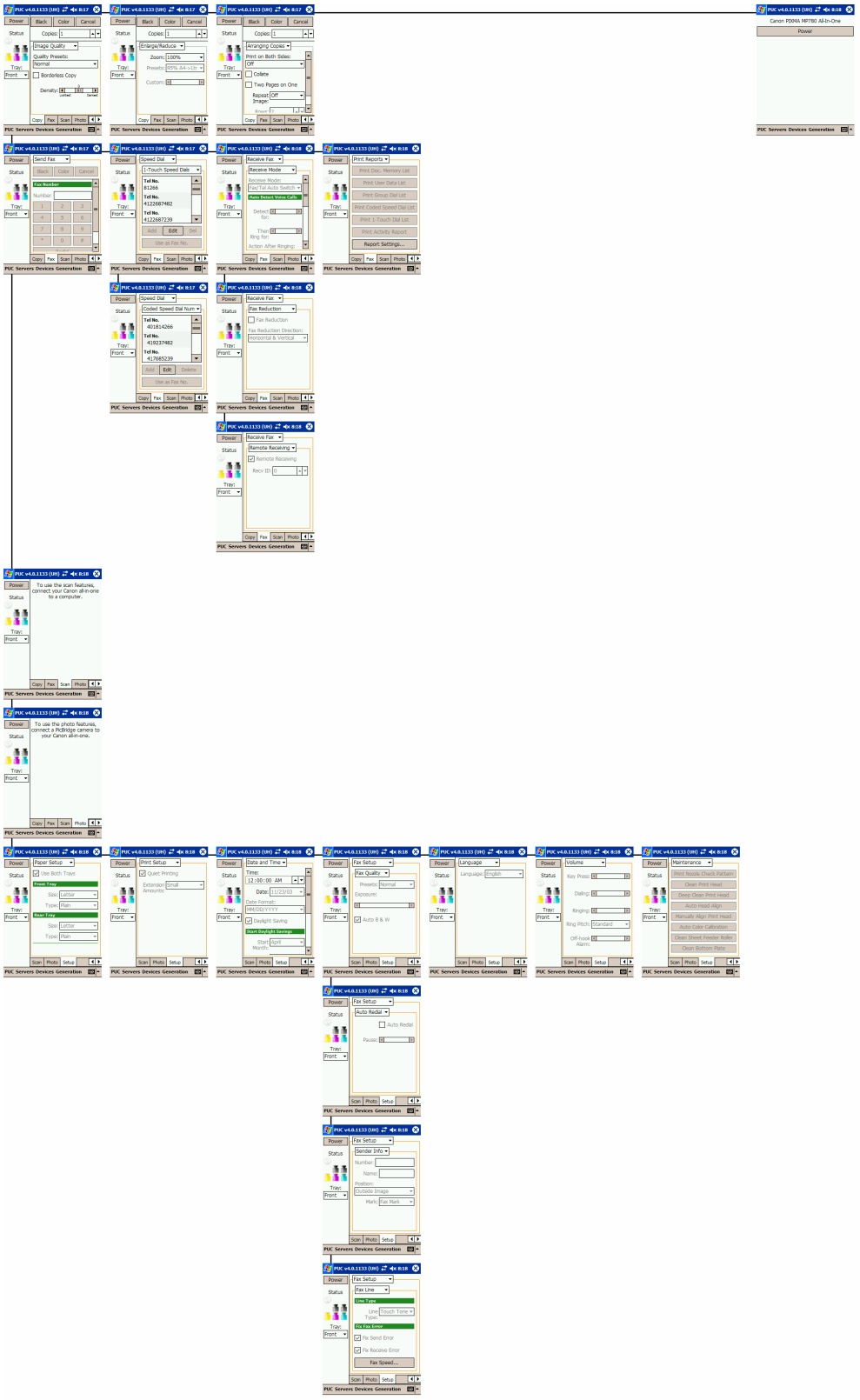


Figure F.2. The full interface for the Canon All-In-One printer generated without consistency, as seen by users in the usability study.

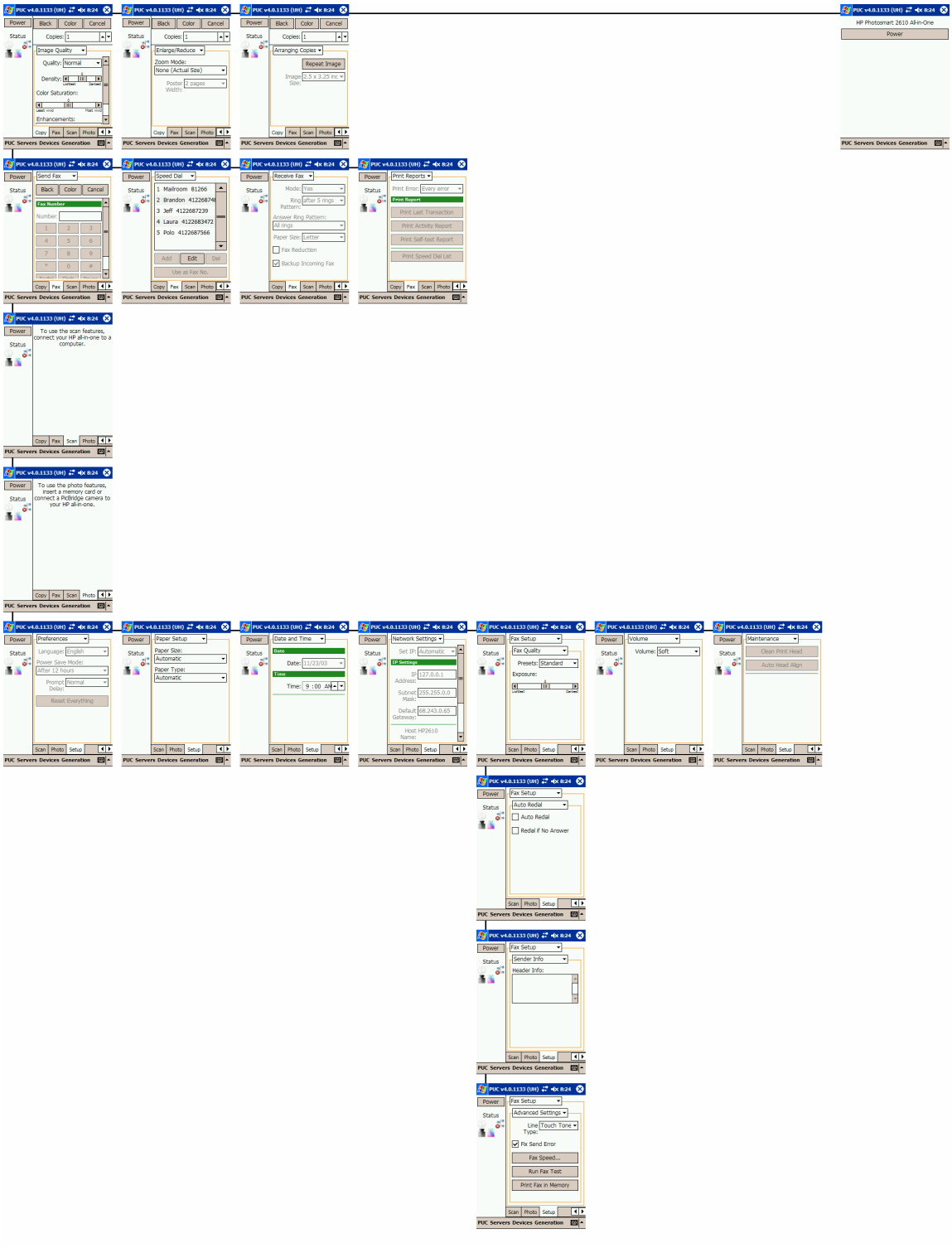


Figure F.3. The full interface for the HP All-In-One printer generated to be consistent with the Canon printer, as seen by users in the usability study.

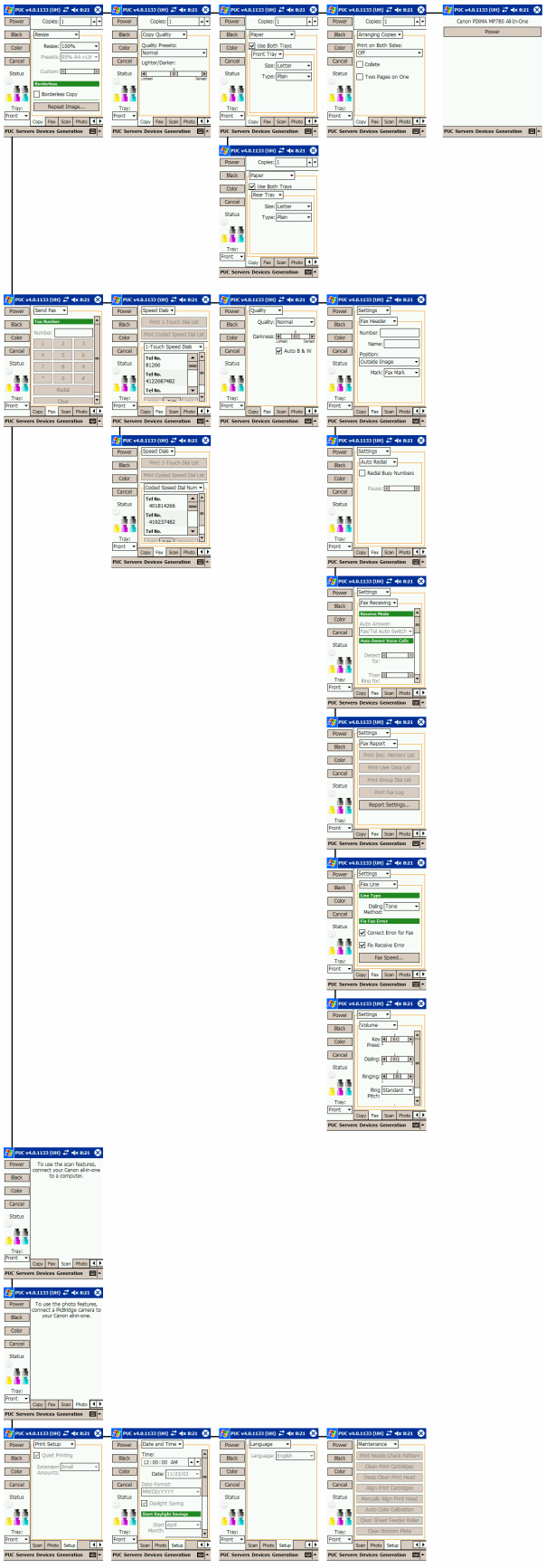


Figure F.4. The full interfaces for the Canon All-In-One printer generated to be consistent with the HP printer, as seen by users in the usability study.

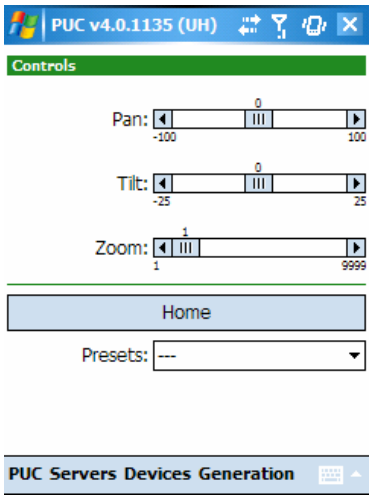


Figure F.5. The full interface generated for the Axis Pan-Tilt-Zoom UPnP surveillance camera. Presets specified on the device can be accessed through the combo box at the bottom of the screen.

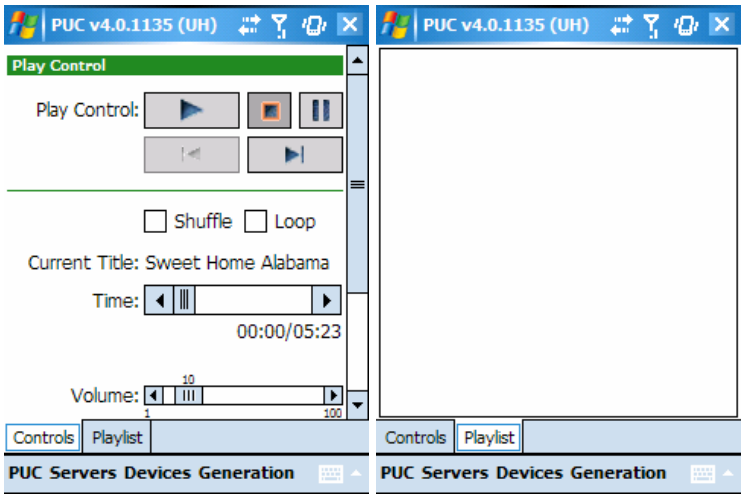


Figure F.6. Both screens of the interface generated for controlling Windows Media Player. The left screen contains controls for playback and the right screen shows the current playlist (empty in this example).

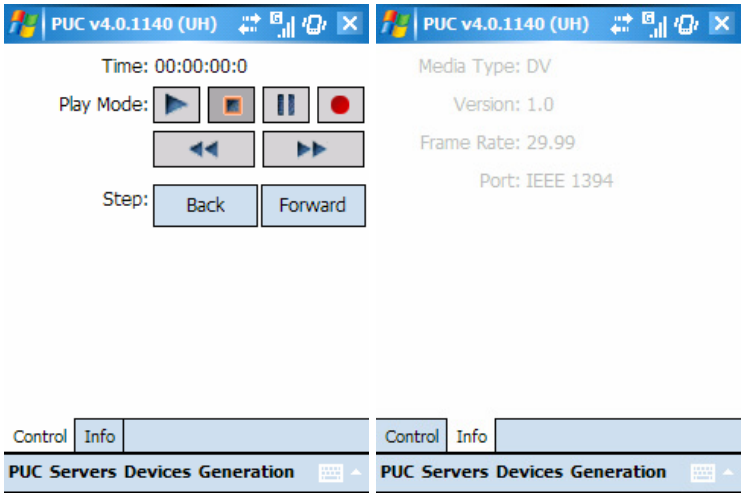


Figure F.7. The full interface for the Sony Camcorder appliance.

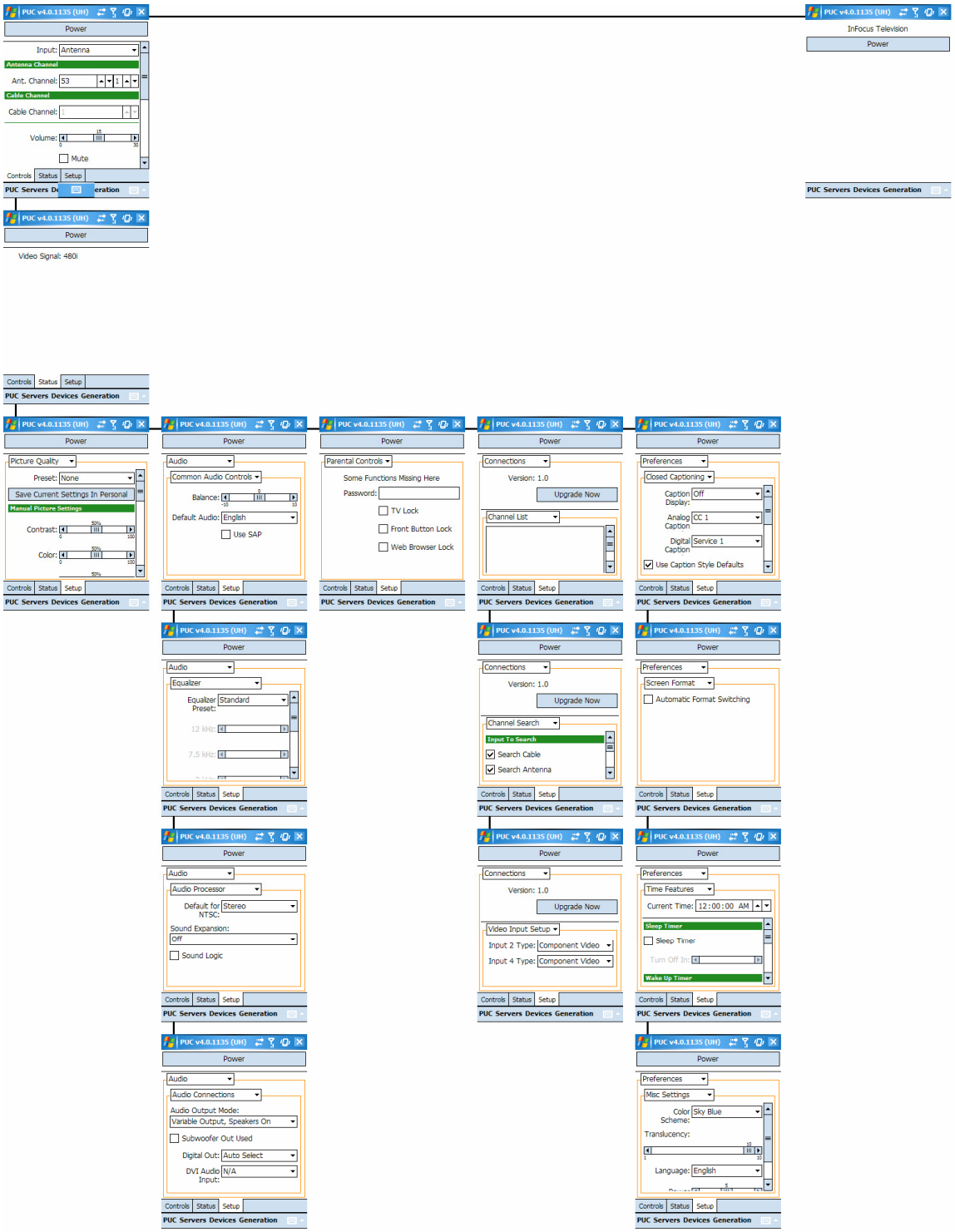


Figure F.8. The full interfaces for the InFocus television, which was a component of the home theater system used for interface aggregation.

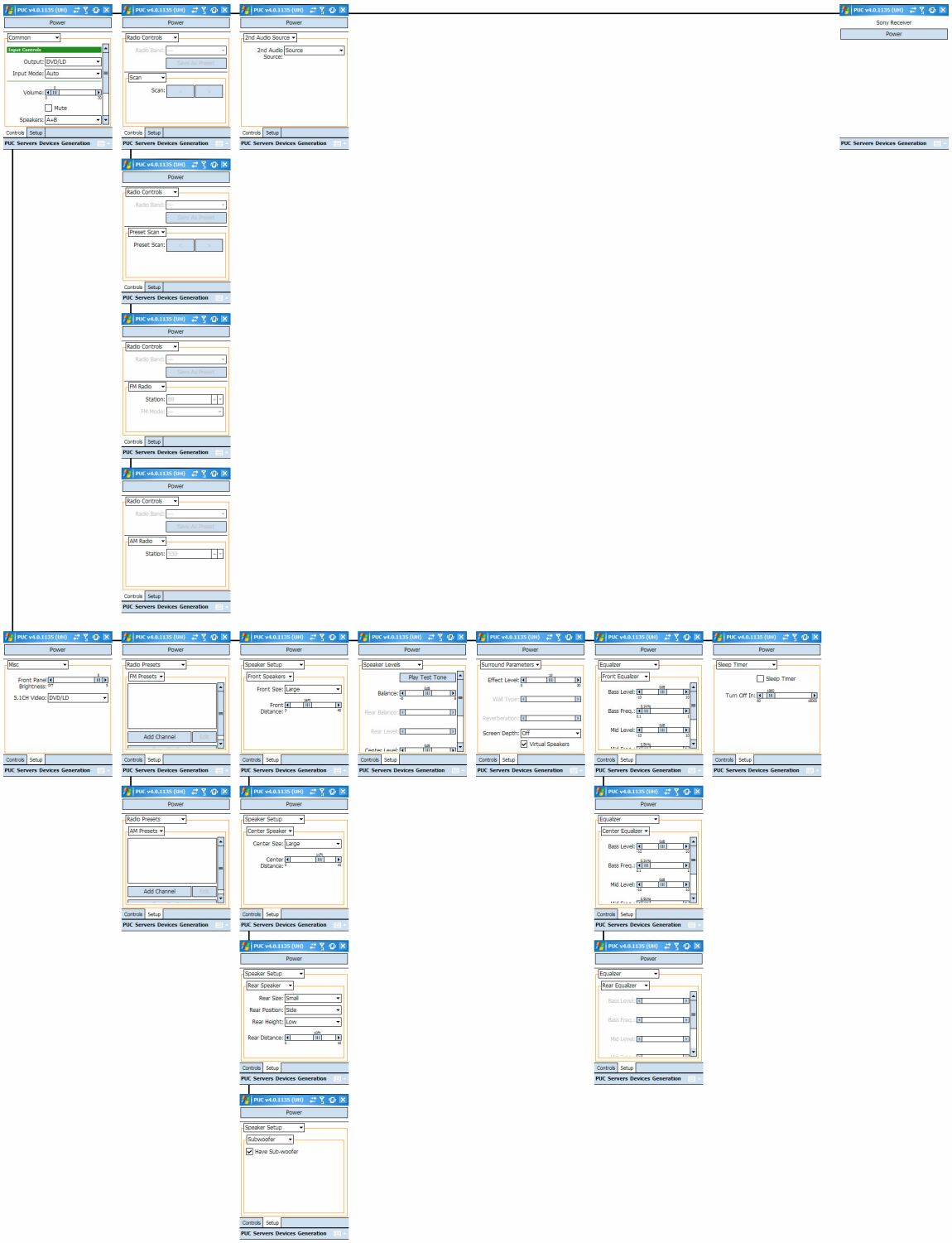


Figure F.9. The full interfaces for the Sony A/V Receiver, which was a component of the home theater system used for interface aggregation.

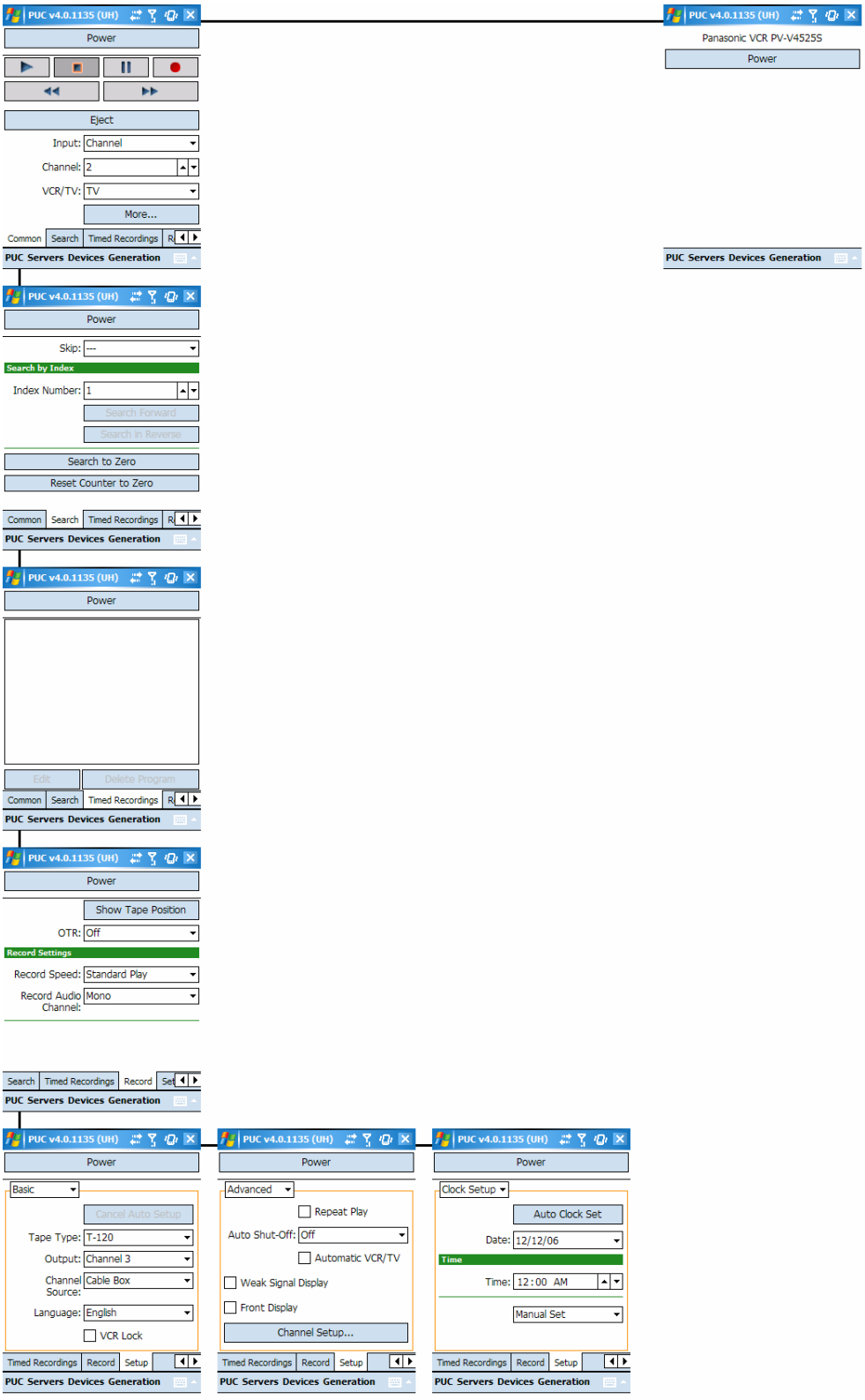


Figure F.10. The full interface for the Panasonic VCR on the PocketPC. This appliance was used both in the home theater scenario for interface aggregation and as one of the test appliances for the consistency system.

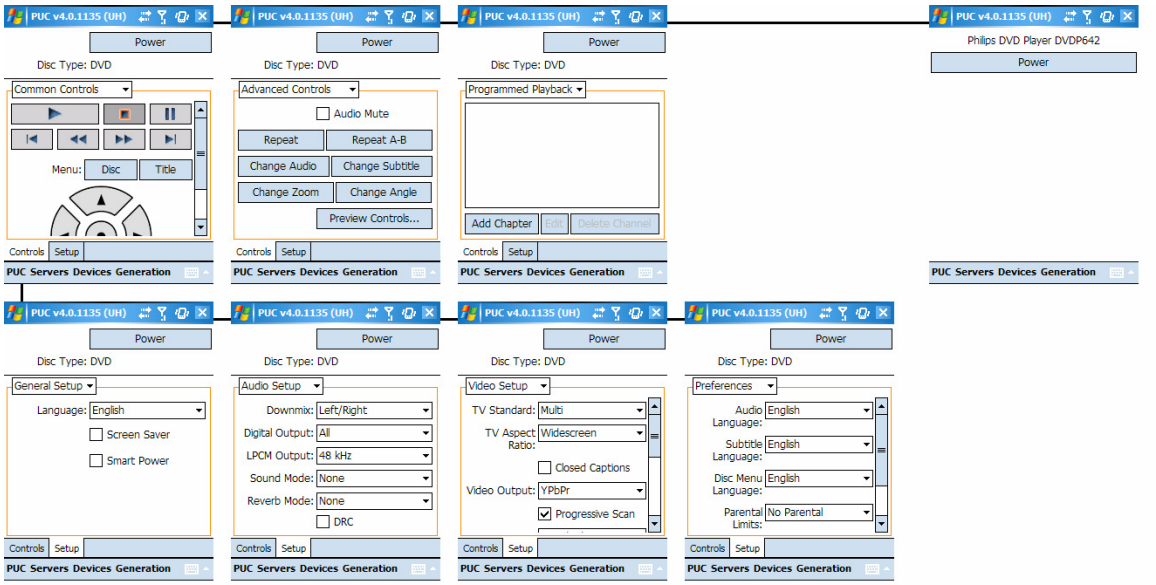


Figure F.11. The full interface for the Philips DVD player, which was a component of the home theater system used for interface aggregation.

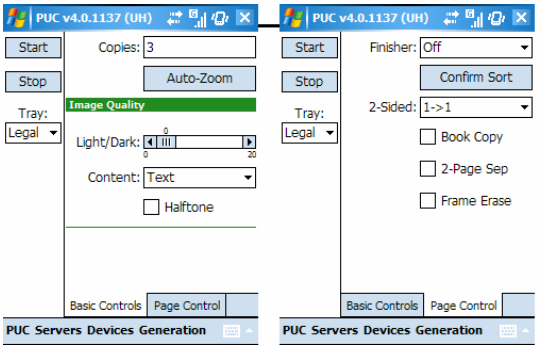


Figure F.12. The full interface* for Copier A generated without consistency.

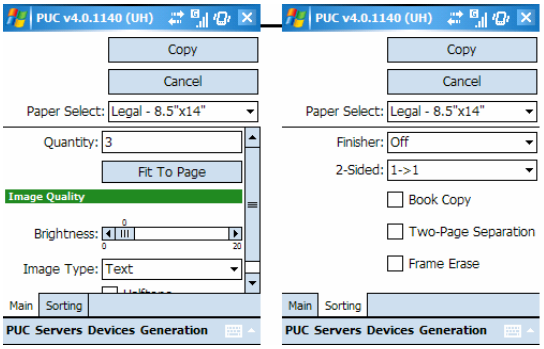


Figure F.13. The full interface* for Copier B generated without consistency.

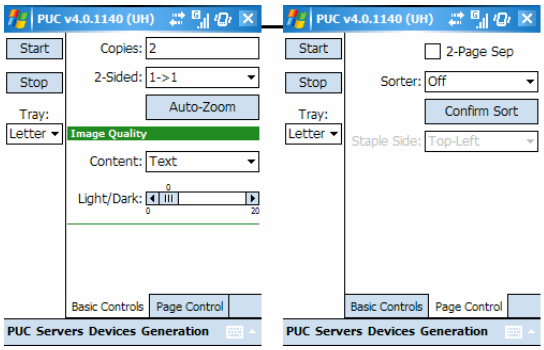


Figure F.14. The full interface* for Copier A generated to be consistent with Copier B.

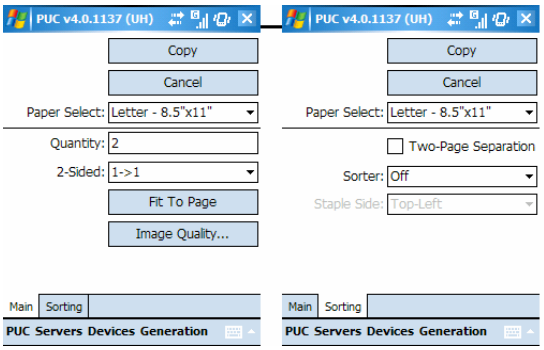


Figure F.15. The full interface* for Copier B generated to be consistent with Copier A.

* These interfaces were generated with a new heading separator feature that did not exist during the original work on the consistency. The new feature caused the generation process to produce slightly different interfaces than those shown in Figure 1.5.

Bibliography

[Ali 2002] Mir Farooq Ali, Manuel A. Perez-Quinones, Marc Abrams and Eric Shell. "Building Multi-Platform User Interfaces with UIML," *Computer-Aided Design of User Interfaces,* Valenciennes, France, May, 2002. pp. 255-266.

[Association 1996] 1394 Trade Association. *AV/C Digital Interface Command Set*. 1996. http://www.1394ta.org/.

[Banavar 2004a] Guruduth Banavar*, et. al.* "An Authoring Technology for Multidevice Web Applications," *IEEE Pervasive Computing*. July-Sept, 2004a. **3**(3). pp. 83-93.

[Banavar 2004b] Guruduth Banavar, Lawrence D. Bergman, Yves Gaeremynck, Danny Soroker and Jeremy Sussman. "Tooling and system support for authoring multi-device applications," *The Journal of Systems and Software*. 2004b. **69**(3). pp. 227-242.

[Barakonyi 2004] Istvan Barakonyi, Thomas Psik and Dieter Schmalstieg. "Agents That Talk And Hit Back: Animated Agents in Augmented Reality," *Proceedings of the Third IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR),* Washington D.C., Nov 2-5, 2004. pp. 141-150.

[Barnard 1981] P.J. Barnard, N.V. Hammond, J. Morton and J.B. Long. "Consistency and compatibility in human-computer dialogue," *Interactional Journal of Man-Machine Studies*. 1981. **15** pp. 87-134.

[Baudisch 2004] Patrick Baudisch, John Pruitt and Steve Ball. "Flat Volume Control: Improving Usability by Hiding the Volume Control Hierarchy in the User Interface," *Proceedings of CHI,* Vienna, Austria, April 24-29, 2004. pp. 255-262.

[Bojanic 2006] Peter Bojanic. *The Joy of XUL*. 2006. http://developer.mozilla.org/en/docs/The_Joy_of_XUL.

[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.

[Card 1983] Stuart K. Card, Thomas P. Moran and Allen Newell. *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, Lawrence Erlbaum Associates. 1983.

[CMU 1998] CMU. *Carnegie Mellon Pronuncing Dictionary*. 1998. http://www.speech.cs.cmu.edu/cgi-bin/cmudict.

[CMU 2006] CMU. *Speech at CMU*. Pittsburgh, PA, **2006.** 2006. http://www.speech.cs.cmu.edu/.

[de Baar 1992] D.J.M.J. de Baar, Foley, J.D., Mullet, K.E. "Coupling Application Design and User Interface Design," *Conference on Human Factors and Computing Systems,* Monterey, California, ACM Press. 1992. pp. 259-266.

[Denis 2003] Charles Denis and Laurent Karsenty. "Inter-usability of multi-device systems: A conceptual framework." *Multiple User Interfaces.* A. Seffah and H. Javahery, Eds. 2003: John Wiley & Sons. pp. 373-385.

[DLNA 2006] DLNA. *Digital Living Network Alliance Home Page*. 2006. http://www.dlna.org/.

[Doan 2001] AnHai Doan, Pedro Domingos and Alon Halevy. "Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach," *SIGMOD,* 2001. pp. 509-520.

[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.

[Eustice 1999] K.F. Eustice, T. J. Lehman, A. Morales, M. C. Munson, S. Edlund and M. Guillen. "A Universal Information Appliance," *IBM Systems Journal.* October, 1999. **38**(4). pp. 575-601. http://www.research.ibm.com/journal/sj/384/eustice.html.

[Florins 2004] Murielle Florins, Daniella G. Trevisan and Jean Vanderdonckt. "The Continuity Property in Mixed Reality and Multiplatform Systems: A Comparative Study," *CADUI'04,* Funchal, Portugal, January 13-16, 2004. pp. 323-334.

[Foltz 2001] Mark A. Foltz. "Ligature: Gesture-Based Configuration of the E21 Intelligent Environment," *Proceedings of the MIT Student Oxygen Workshop,* 2001.

[Gajos 2004] K. Gajos, Weld, D. "SUPPLE: Automatically Generating User Interfaces," *Intelligent User Interfaces,* Funchal, Portugal, 2004. pp. 93-100.

[Gajos 2005a] Krzysztof Gajos, David Christianson, Raphael Hoffmann, Tal Shaked, Kiera Henning, Jing Jing Long and Daniel S. Weld. "Fast And Robust Interface Generation for Ubiquitous Applications," *Seventh International Conference on Ubiquitous Computing (UBICOMP),* Tokyo, Japan, 2005a. pp. 37-55.

[Gajos 2005b] Krzysztof Gajos, Anthony Wu and Daniel S. Weld. "Cross-Device Consistency in Automatically Generated User Interfaces," *Proceedings of the 2nd Workshop on Multi-User and Ubiquitous User Interfaces,* San Diego, January 9, 2005b. pp. 7-8.

[Gajos 2006] Krzysztof Z. Gajos, Jing Jing Long and Daniel S. Weld. "Automatically Generating Custom User Interfaces for Users With Physical Disabilities," *ASSETS,* Portland, OR, 2006. p. To appear.

[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.

[Grudin 1989] Jonathan Grudin. "The Case Against User Interface Consistency," *CACM. CACM.* Oct, 1989, 1989. **32**(10). pp. 1164-1173.

[Hall 2003] Richard S. Hall and Humberto Cervantes. "Gravity: Supporting Dynamically Available Services in Client-Side Applications," *Proceedings of ESEC/FSE,* Helsinki, Finland, September 1-5, 2003. pp. 379-382.

[Harris 2004] Thomas K. Harris and Roni Rosenfeld. "A Universal Speech Interface for Appliances," *International Conference on Speech and Language Processing (ICSLP),* Jeju, Korea, 2004.

[HAVi 2003] HAVi. *Home Audio/Video Interoperability*. **2003.** 2003. http://www.havi.org.

[Hayes 1985] Philip J. Hayes, Pedro A. Szekely and Richard A. Lerner. "Design Alternatives for User Interface Management Systems Based on Experience with COUSIN," *Human Factors in Computing Systems,* San Francisco, CA, Apr, 1985, 1985. pp. 169-175.

[Herman 2006] Ivan Herman and Jim Hendler. *Web Ontology Language OWL / W3C Semantic Web Activity*. 2006. http://www.w3.org/2004/OWL/.

[Hodes 1997] Todd D. Hodes, Randy H. Katz, Edouard Servan-Schreiber and Lawrence Rowe. "Composable ad-hoc mobile services for universal interaction," *Proceedings of the Third annual ACM/IEEE international Conference on Mobile computing and networking (ACM Mobicom'97),* Budapest Hungary, September 26 - 30, 1997. pp. 1 - 12.

[IMTC 2006] Georgia Tech IMTC. *Projects: V2 Universal Remote Console Standard.* Atlanta, GA, **2006.** 2006. http://www.imtc.gatech.edu/projects/technology/images/v2_winamp.html.

[INCITS/V2 2003] INCITS/V2. *Universal Remote Console Specification.* Alternate Interface Access Protocol. Washington D.C., December 31, 2003.

[ISO 1988] ISO. *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on Temporal Ordering of Observational Behavior.* 1988.

[Ivory 2001] Melody Y. Ivory and Marti A. Hearst. "The State of the Art in Automating Usability Evaluation of User Interfaces," *ACM Computing Surveys.* December, 2001. **33**(4). pp. 470-516.

[John 1996] Bonnie E. John and David E. Kieras. "The GOMS Family of User Interface Analysis Techniques: Comparison and Contrast," *ACM Transactions on Computer-Human Interaction.* 1996. **3**(4). pp. 320-351.

[John 2004] Bonnie E. John, Konstantine Prevas, Dario D. Salvucci and Ken Koedinger. "Predictive Human Performance Modeling Made Easy," *CHI,* Vienna, Austria, April 24-29, 2004. pp. 455-462.

[Kellogg 1987] Wendy A. Kellogg. "Conceptual consistency in the user interface: Effects on user performance," *Proceedings of INTERACT'87, Conference on Human-Computer Interaction,* Stuttgart, September 1-4, 1987.

[Kieras 1995] David E. Kieras, Scott D. Wood, Kasen Abotel and Anthony Hornof. "GLEAN: A Computer-Based Tool for Rapid GOMS Model Usability Evaluation of User Interface Designs," *Eighth Annual Symposium on User Interface Software and Technology,* Pittsburgh, PA, Nov, 1995. pp. 91-100.

[Kim 2004] Jihie Kim, Marc Spraragen and Yolanda Gil. "An Intelligent Assistant for Interactive Workflow Composition," *Intelligent User Interfaces (IUI),* Madeira, Portugal, 2004. pp. 125-131.

[Kim 1993] Won Chul Kim and James D. Foley. "Providing High-level Control and Expert Assistance in the User Interface Presentation Design," *Human Factors in Computing Systems,* Amsterdam, The Netherlands, Apr, 1993. pp. 430-437.

[Lieberman 2006] Henry Lieberman and Jose Espinosa. "A Goal-Oriented Interface to Consumer Electronics using Planning and Commonsense Reasoning," *Intelligent User Interfaces,* Sydney, Australia, 2006. pp. 226-233.

[Limbourg 2004] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon and Victor Lopez-Jaquero. "UsiXML: a Language Supporting Multi-Path Development of User Interfaces," *9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with 11th International Workshop on Design, Specification, and Verification of Interactive Systems (EHCI-DSVIS'2004),* Hamburg, Germany, 2004. pp. 200-220.

[Madhaven 2001] Jayant Madhaven, Philips A. Bernstein and Erhard Rahm. "Generic Schema Matching with Cupid," *27th VLDB Conference,* 2001.

[Mahajan 1997] R. Mahajan and B. Shneiderman. "Visual and Textual Consistency Checking Tools for Graphical User Interfaces," *IEEE Transactions on Software Engineering.* 1997. **23**(11). pp. 722-735.

[Melnik 2002] Sergey Melnik, Hector Garcia-Molina and Erhard Rahm. "Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching," *18th ICDE,* San Jose, CA, 2002. pp. 117-128.

[Merrill 2006] Duane Merrill. *Mashups: The new breed of Web app.* 2006. http://www-128.ibm.com/developerworks/library/x-mashups.html.

[Microsoft 2006] Microsoft. *XAML.* 2006. http://windowssdk.msdn.microsoft.com/en-us/library/ms747122.aspx.

[Miori 2006] Vittorio Miori, Luca Tarrini, Maurizio Manca and Gabriele Tolomei. "DomoNot: a framework and a prototype for interoperability of domotic middlewares based on XML and Web Services," *International Conference on Consumer Electronics (ICCE'06),* January 7-11, 2006. pp. 117-118.

[Mori 2004] Giulio Mori, Fabio Paterno and Carmen Santoro. "Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions," *IEEE Transactions on Software Engineering.* 2004. **30**(8). pp. 1-14.

[Mori 2002] Guillo Mori, Fabio Paterno and Carmen Santoro. "CTTE: Support for Developing and Analyzing Task Models for Interactive System Design," *IEEE Transactions on Software Engineering.* September 2002, 2002. **28**(9). pp. 797-813.

[Myers 2000] Brad A. Myers, Scott E. Hudson and Randy Pausch. "Past, Present and Future of User Interface Software Tools," *ACM Transactions on Computer Human Interaction.* 2000. **7**(1). pp. 3-28.

[Newman 2002] Mark W. Newman, Shahram Izadi, W. Keith Edwards, Jana Z. Sedivy and Trevor F. Smith. "User Interfaces When and Where They are Needed: An Infrastructure for Recombinant Computing," *UIST'02,* Paris, France, October 27-30, 2002. pp. 171-180.

[Nichols 2002a] J. Nichols, Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Shriver, S. "Requirements for Automatically Generating Multi-Modal Interfaces for Complex Appliances," *ICMI,* Pittsburgh, PA, 2002a.

[Nichols 2002b] Jeffrey Nichols, Brad A. Myers, Michael Higgins, Joseph Hughes, Thomas K. Harris, Roni Rosenfeld and Mathilde Pignol. "Generating Remote Control Interfaces for Complex Appliances," *UIST 2002,* Paris, France, 2002b. pp. 161-170.

[Nichols 2003] J. Nichols, Myers, B.A. "Studying The Use Of Handhelds to Control Smart Appliances," *23rd International Conference on Distributed Computing Systems Workshops (ICDCS '03),* Providence, RI, May 19-22, 2003. pp. 274-279.

[Nichols 2004a] Jeffrey Nichols and Brad Myers. *Report on the INCITS/V2 AIAP-URC Standard*. 2004a. http://www.cs.cmu.edu/~jeffreyn/papers/cmu-puc-v2-report.pdf.

[Nichols 2004b] Jeffrey Nichols, Brad A. Myers and Kevin Litwack. "Improving Automatic Interface Generation with Smart Templates," *Intelligent User Interfaces,* Funchal, Portugal, 2004b. pp. 286-288.

[Nichols 2005] Jeffrey Nichols and Brad A. Myers. "Generating Consistent User Interfaces for Appliances," *Workshop on Multi-User and Ubiquitous User Interfaces (MU3I),* San Diego, CA, 2005. pp. 9-10. http://www.jeffreynichols.com/papers/nichols-mu3i.pdf.

[Nielsen 1993] Jakob Nielsen. *Usability Engineering*. Boston, Academic Press. 1993.

[Nylander 2004] Stina Nylander, Markus Bylund and Annika Waern. "The Ubiquitous Interactor - Device Independent Access to Mobile Services," *Computer-Aided Design of User Interfaces (CADUI),* Madeira, Portgual, 2004. pp. 271-282.

[Olsen Jr. 1989] Dan R. Olsen Jr. "A Programming Language Basis for User Interface Management," *Human Factors in Computing Systems,* Austin, TX, Apr, 1989, 1989. pp. 171-176.

[Olsen Jr. 2000] Dan R. Olsen Jr., Sean Jefferies, Travis Nielsen, William Moyes and Paul Fredrickson. "Cross-modal Interaction using Xweb," *Proceedings UIST'00: ACM SIGGRAPH Symposium on User Interface Software and Technology,* San Diego, CA, 2000. pp. 191-200.

[Omojokun 2006] Olufisayo Omojokun, Jeffrey S. Pierce, Charles L. Isbell Jr. and Prasun Dewan. "Comparing End-User and Intelligent Remote Control Interface Generation," *Personal and Ubiquitous Computing*. April, 2006. **10**(2-3). pp. 136-143.

[OSGi 2006] OSGi. *The OSGi Alliance Home Page*. 2006. http://www.osgi.org/.

[Paterno 1997] Fabio Paterno, C. Mancini and S. Meniconi. "ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models," *INTERACT,* Sydney, Australia, 1997. pp. 362-269.

[Polson 1986] Peter G. Polson, E. Muncher and G. Engelbeck. "A test of a common elements theory of transfer," *SIGCHI conference on Human factors in computing systems,* New York, NY, ACM Press. 1986. pp. 78-83.

[Polson 1988] Peter G. Polson. "The consequences of consistent and inconsistent user interfaces." *Cognitive science and its applications for human-computer interaction.* 1988: Hillsdale, NJ, Lawrence-Erlbaum.

[Ponnekanti 2001] S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan and T.Winograd. "ICrafter: A service framework for ubiquitous computing environments," *UBICOMP 2001,* Atlanta, Georgia, 2001. pp. 56-75.

[Puerta 2002] A. Puerta, Eisenstein, J. "XIML: A Common Representation for Interaction Data," *7th International Conference on Intelligent User Interfaces,* San Francisco, 2002. pp. 214-215.

[Puerta 1997] Angel R. Puerta. "A Model-Based Interface Development Environment," *IEEE Software.* July/August, 1997. **14**(4). pp. 41-47.

[Reisner 1981] Phyllis Reisner. "Formal Grammar and Human Factors Design an Interactive Graphics System," *IEEE Transactions on Software Engineering.* March 1981, 1981. **SE-7**(2). pp. 229-240.

[Reisner 1990] Phyllis Reisner. "What is inconsistency?" *INTERACT,* 1990. pp. 175-181.

[Rheinfrank 1996] J. Rheinfrank and S. Evenson. "Design Languages." *Bringing Design to Software.* T. Winograd, Ed. 1996: New York, Addison-Wesley (ACM Press). pp. 63-80.

[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," *IEEE International Conference on Distributed Computing Systems Workshops,* June, 2005. pp. 514-519.

[Rosenfeld 2001] Roni Rosenfeld, Jr. Olsen, Dan and Alex Rudnicky. "Universal Speech Interfaces," *interactions: New Visions of Human-Computer Interaction.* 2001. **VIII**(6). pp. 34-44.

[Sandridge 2005] Nick Sandridge, Brian LaShomb, Katrina Roan and Terri Chapman. *Code Generation for J2ME and C#-based Mobile Devices using Domain Specific XML-Based Source Languages.* 2005. http://unix.eng.ua.edu/~blashomb/reu/491paper.doc.

[Satzinger 1998] John W. Satzinger and Lorne Olfman. "User Interface Consistency Across End-User Applications: The Effects on Mental Models," *Journal of Information Management Systems.* 1998. **14**(4). pp. 167-193.

[Shriver 2001] S. Shriver, Toth, A., Zhu, X., Rudnikcy, A., Rosenfeld, R. "A Unified Design for Human-Machine Voice Interaction," *Extended Abstracts of CHI 2001,* Seattle, WA, March31-April 5, 2001. pp. 247-248.

[Shvaiko 2005] Pavel Shvaiko and Jerome Euzenat. "A Survey of Schema-based Matching Approaches," *Journal on Data Semantics.* 2005.

[Sproat 1998] R. Sproat, Hunt, A., Ostendorf, P., Taylor, P., Black, A., Lenzo, K., Edgington, M. "SABLE: A Standard for TTS Markup," *International Conference on Spoken Language Processing,* Sydney, Australia, 1998.

[Srivastava 2003] Biplav Srivastava and Java Koehler. "Web Service Composition: Current Solutions and Open Problems," *ICAPS Workshop on Planning and Scheduling for Web and Grid Services,* 2003. pp. 28-35.

[Staab 2003] Steffen Staab*, et. al.* "Web Services: Been There, Done That?" *IEEE Intelligent Systems.* 2003. **18**(1). pp. 72-85.

[Sukaviriya 1990] Piyawadee Sukaviriya and James D. Foley. "Coupling A UI Framework with Automatic Generation of Context-Sensitive Animated Help," *ACM SIGGRAPH Symposium on User Interface Software and Technology,* Snowbird, Utah, Oct, 1990, 1990. pp. 152-166.

[Sukaviriya 1993] Piyawadee Sukaviriya, James D. Foley and Todd Griffith. "A Second Generation User Interface Design Environment: The Model and The Runtime Architecture," *Human Factors in Computing Systems,* Amsterdam, The Netherlands, Apr, 1993. pp. 375-382.

[Sun 2003] Sun. *Jini Connection Technology.* **2003.**

[Sycara 2003] Katia Sycara, Massimo Paolucci, Anupriya Ankolekar and Naveen Srinivasan. "Automated Discovery, Interaction, and Composition of Semantic Web Services," *Journal of Web Semantics.* 2003. **1**(1).

[Szekely 1995] P. Szekely, Sukaviriya, P., Castells, P., Muthukumarasamy, J., Salcher, E. "Declarative Interface Models for User Interface Construction Tools: the Mastermind Approach," *6th IFIP Working Conference on Engineering for Human Computer Interaction,* Grand Targhee Resort, 1995. pp. 120-150.

[Szekely 1992] Pedro Szekely, Ping Luo and Robert Neches. "Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design," *Human Factors in Computing Systems,* Monterey, CA, May, 1992, 1992. pp. 507-515.

[Szekely 1996] Pedro 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.

[Tomko 2004] Stefanie Tomko and Roni Rosenfeld. "Speech Graffiti vs. Natural Language: Assessing the User Experience," *HLT/NAACL,* Boston, MA, 2004.

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

[Vander Zanden 1990] Brad Vander Zanden and Brad A. Myers. "Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces," *Human Factors in Computing Systems,* Seattle, WA, Apr, 1990. pp. 27-34.

[Vanderdonckt 1995] J. Vanderdonckt. "Knowledge-Based Systems for Automated User Interface Generation: the TRIDENT Experience," *Technical Report RP-95-010,* Namur: Facultes Universitaires Notre-Dame de la Paix, Institut d' Informatique, 1995.

[Vanderdonckt 1999] J. Vanderdonckt. "Advice-Giving Systems for Selecting Interaction Objects," *User Interfaces to Data Intensive Systems.* 1999. pp. 152-157.

[W3C 2006] W3C. *Resource Description Framework (RDF).* 2006. http://www.w3.org/RDF/.

[Ward 1990] W. Ward. "The CMU Air Travel Information Service: Understanding Spontaneous Speech," *DARPA Speech and Natural Language Workshop,* 1990.

[Wiecha 1990] Charles Wiecha, William Bennett, Stephen Boies, John Gould and Sharon Greene. "ITS: A Tool for Rapidly Developing Interactive Applications," *ACM Transactions on Information Systems*. Jul, 1990, 1990. **8**(3). pp. 204-236.