Automatically Generating High-Quality User Interfaces for Appliances

Thesis Proposal

Jeffrey Nichols
jeffreyn@cs.cmu.edu
Human Computer Interaction Institute
Carnegie Mellon University
April 2004

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

Abstract

With the increasing pervasiveness of wireless technologies, users will benefit from interacting with appliances in their environment using their mobile device. One of the challenges to supporting such ubiquitous control is the creation of user interfaces for the remote control functions on the mobile device. I propose to examine an approach I call the *Personal Universal Controller* (PUC) where the mobile device downloads an abstract description of the appliances' functions and then automatically generates high quality interfaces that enable control. The PUC system includes an appliance specification language that contains no specific details about how a user interface for the appliance should be designed, a protocol for communication between controller devices and appliances, custom software and hardware for adapting existing appliances to support the PUC protocol, and interface generators for three platforms: PocketPC, Smartphone, and desktop computers.

The interface generators will support several features that are unique to the PUC system. The interface generators will use a general technique called Smart Templates to render standard conventions with which users are familiar, such as the arrangement of buttons on a telephone dial-pad or the conventional play, stop, and pause icons on a media player. Interfaces generated by the PUC system will be based on previous designs for similar appliances in order to achieve consistency for the user. The PUC system will also be able to generate a single combined user interface for multiple connected appliances, such as a home theater system.

To evaluate the completed PUC system, I will develop specifications for a wide variety of appliances that are representative of the common appliances that users encounter every day, and perform studies that compare user performance on the manufacturers' user interfaces versus automatically generated interfaces for some of those appliances. My thesis is:

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.

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, VCR, alarm clock, and stereo at home, and the 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. As the user I have to spend time learning how to use every device in my environment, even when they are similar to other devices that I already know how to use.

One solution to this problem is to move the user interface from the appliance to some other intermediary “UI device.” 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 will soon have the ability to communicate with other devices in the environment through 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 on an appliance.

This thesis will build a *Personal Universal Controller* (PUC), which enables intermediary UI devices to be constructed. The PUC system supports *two-way communication*, which allows the controller device to download from the appliance an abstract description of that appliance’s functions. The controller then *automatically generates* a user interface that allows the user to send control signals to the appliance and receive feedback on the appliance’s state.

The benefits of having a PUC device are:

Manufacturers are no longer solely responsible for creating a high quality user interface for their complex appliances. The trend has been that as appliances increase in complexity, their interfaces decrease in quality [3]. This happens in part because the manufacturer’s sole concern is the sale of their product, and unfortunately price and functionality, not usability, seem to be the most important factors in making a sale. The PUC moves responsibility for the user interface to specialized interface generator software and hardware whose sole concern is usability.

The interface generation software can run on many platforms. The abstract appliance specification language contains enough general information for interfaces to be generated on a variety of platforms. I propose to build interface generators for Microsoft’s PocketPC and Smartphone, and for desktop computers, and to collaborate with researchers in the Universal Speech Interfaces (USI) group here at Carnegie Mellon on building a speech interface generator.

Special user interface technology can be built into a PUC device that would not be affordable on every appliance. It is not practical to build a large color LCD and a touch-sensitive screen into every appliance, but a PUC device that has such hardware can use it to improve the generated user interfaces for every appliance.

One user interface can be created for multiple connected appliances. Some appliances are commonly used with several other appliances, which may be thought of by the user as one conceptual system. Some examples are a home theater system, or a presentation system in a conference room that consists of the PowerPoint application on a desktop computer and a data projector. If information is available regarding how a system of appliances is connected, then that information can be combined

with the abstract specifications for each appliance to create a single interface for the conceptual system.

Interfaces can be made consistent for the user, both within the controller device and across similar appliances. PUC interfaces will be easier to use because they can leverage the user's existing knowledge about the controller device and appliances that the user has used in the past. A PUC interface generator will build interfaces that are consistent with other applications that run on that controller device, so an interface built on a PocketPC will look like other applications that run on the PocketPC. The interface generator will also make interfaces consistent across similar appliances. This could solve one problem mentioned above by making the interfaces on my controller device for my home and car stereos look similar because both of these appliances have similar functions.

User preferences can be taken into account in the generated interfaces. For example, an elderly person who has failing eyesight could set a preference to increase the font size and the size of interface elements, making the generated interfaces easier to use.

An interface is not necessarily required on the appliance. A PUC controller can provide a user interface to the full functionality of an appliance, rather than a subset as most current "universal controls" do. While removing the interface on the appliance will not always be beneficial, it is necessary for some "invisible" computing appliances envisioned by ubiquitous computing researchers. Already some of today's appliances, such as televisions and VCRs, have very few physical controls and rely on a remote control and video screen to provide most of the user interface.

The idea of a system for controlling appliances is not new. Consumer electronics companies have sold universal remote controls for many years, but the PUC system improves on universal remote control technology in a number of ways. The PUC system supports two-way communication with appliances, allowing controllers to display state information and disable functions that are not currently available. A PUC controller can also control the full functionality of an appliance, unlike universal remote controls that are limited to controlling the most common subset of functions across every appliance. Finally, PUC controllers automatically generate user interfaces and are not limited to producing interfaces for a set of appliances that are pre-programmed into the controller at the factory.

Researchers have also examined how appliances can be controlled. Systems such as the universal interactor [11] and ICrafter [26] investigated infrastructures for distributing appliance user interfaces to controller devices in the environment. Though both of these systems supported simple automatic generation of user interfaces, both preferred to use hand-generated interfaces when available. The PUC system differs from these systems in its focus on automatic user interface generation and the quality of the resulting interfaces. Xweb [23] provided an infrastructure for interactive computing that was analogous to the world-wide web. All user interfaces in Xweb were automatically generated from an abstract description of the interactive service being controlled. The PUC system extends the ideas of Xweb with more detail in the specification language, support for ensuring consistent user interfaces across similar appliances, and the ability to generate a single user interface for controlling multiple appliances.

Automatic user interface generation has also been investigated by researchers in the past. User interfaces generated in this way were typically known as model-based user interfaces [33]. Unlike the PUC system, most model-based UI work relied on an interaction designer to guide the generation process and/or to edit the resulting user interfaces to fix any design problems. I do not believe that end-users of the PUC system will be willing to spend the time and effort to modify their automatically generated interfaces in this way, and thus the PUC system will need to generate high quality user interfaces on the first attempt.

The novel contributions of the PUC system are:

- An abstract appliance modeling language for describing the complete functionality of a wide-range of appliances

- Algorithms for automatically generating high quality interfaces for each appliance from that language.
- 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 interfaces modalities.
- Algorithms for determining the similarity between a new appliance specification and specifications that have been used in the past, and algorithms that use that similarity information to apply design decisions from previously generated interfaces in the generation process for a new interface.
- A *distributed* task-modeling language for describing the sub-tasks specific to an appliance in a multi-appliance system. The language will also contain special linking information that allows inter-appliance tasks to be inferred from each appliance's sub-tasks.
- Algorithms for automatically generating a single user interface for multiple appliances using distributed task information.
- Interface generation software on multiple platforms: Microsoft PocketPC, Microsoft Smartphone, and desktop computers, which use the above algorithms to produce interfaces that are shown by user testing to be better than manufacturers' interfaces for the same appliances.

An important part of this thesis is embodied in the last contribution, which requires the generated interfaces to meet a particular standard. There are two ways in which the PUC system must be validated in order to be judged a success: *breadth* of appliances supported by both the specification language and the interface generators, and the *quality* of the generated interfaces as compared to the manufacturers' interfaces for the same appliances. Of these two, breadth is the most difficult to evaluate because it cannot be proven, only demonstrated. I will show breadth by creating a list of appliances that are interesting for their complexity or for a unique feature, writing specifications for these appliances, and generating interfaces from these specifications on each interface generation platform. A partial list and the methods for determining this list are discussed later in the Validation section.

Of the appliances chosen for the breadth validation, I will pick between three and five appliances to conduct comparison user studies. The choice of these appliances is discussed in more detail later, but will likely be based on the complexity and availability of the actual appliance. The user studies will compare user performance, as measured by metrics such as time, errors and help requested, for the automatically generated interface and the manufacturer's appliance interface using techniques similar to those in my preliminary user studies [19].

My thesis is:

> **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.**

In the next section I survey the related work in this area. The following section describes some preliminary user studies that I conducted as a basis for designing and implementing the PUC system. Section 4 describes the architecture of the PUC system, followed by a section describing the interface generation process in more detail. Sections 6-8 describe some of the novel features that I am proposing to implement in the PUC system, followed by a section describing some related features that I am not planning to implement. This is followed by sections describing how the system will be validated, a schedule for completing the thesis, and concludes with a summary of the proposed contributions.

2. Related Work

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 [10], JINI [32], and UPnP [36]. Another standards group, INCITS/V2 [37], 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 [23], ICrafter [26], and Supple [41]. At the end of this section I also survey past and present work on automatic interface generation and the use of various models to support the generation process.

2.1 Existing 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 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.

Figure 1. A Philips Pronto remote control device.

Figure 2. A Harmony remote control device.

In the past few years, several more complicated universal remote controls have been developed that deserve mention: the Philips Pronto [25] and the Harmony remote [13] from Intrigue technologies. The Philips Pronto (see Figure 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 Harmony remote (see Figure 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 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 remotecentral.com and 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 tediousness and time-consuming nature of programming the remote controls, but only for appliances where someone has uploaded the codes. For other appliances, the programming process is just as time-consuming when using these advanced universal remotes.

2.2 Emerging Commercial Standards

A number of industry groups have been formed to create new standards for remotely controlling devices. Four of the most prominent are the Microsoft-led Universal Plug and Play (UPnP) [36] initiative, Sun Microsystem's JINI system [32], the Home Audio Video Interoperability (HAVi) initiative which is led by "eight of the world's leading manufacturers of audio-visual electronics" [10] and the INCITS/V2 effort [37] 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.

Of the four, 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 also produces a VCR that supports 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 [2]. 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.

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.

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 from a number of vendors and a Pan/Tilt video camera from Axis Communications. UPnP currently has standardized five different service descriptions, and more devices are likely to appear on the market as the number of standardized service specifications grows.

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 [37], which is currently developing standards for 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. I have provided feedback to the V2 group in the past that led to the current design of their specification, and plan to continue collaborating with V2 at some point in the future. A detailed report is available analyzing the similarities and differences between the V2 and PUC systems [18].

2.3 Research Systems for Controlling Appliances

A number of research groups are working on controlling appliances from handheld devices. Hodes, et. al. [11] 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 interface 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.

An IBM project [8] 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.

The Stanford ICrafter [26] 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. One example described by the authors is the use of a digital camera and a printer. In most current architectures, the user would have to take a picture with the camera, save that picture to some temporary location, and then send the picture to the printer. Using ICrafter, the user can simply take the picture and send it directly to the printer. To support this, ICrafter requires every appliance to implement generic programming interfaces, such as the DataProducer or DataConsumer interfaces, that describe to the infrastructure how interconnections can be made. This allows ICrafter to provide a user interface to the appliance connection in addition to the particular appliances. The downside of this approach is that the user must use several different interfaces to interact with their connected appliances (one for each appliance and additional interfaces for every connection). The PUC will go beyond this work by integrating all of these separate user interfaces into a single interface for the entire system.

The Xweb [23] 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 [28]. 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 [23]. 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.

2.4 Model-Based User Interface Research

The idea of automatically generating user interfaces from abstract specifications is not new. It was explored in several systems in the early 1980's and the work was extended by at least a dozen systems since then. The interfaces created by these systems came to be known as *model-based user interfaces* because they were built from detailed models of program functionality, user knowledge, presentation environment, etc. This sub-section discusses many of the model-based interface systems and discusses how the PUC system will build on this work. A more detailed summary of model-based interface work can be found in [33].

The motivation for the earliest model-based systems was to simplify the user interface creation process by integrating it with the implementation of application logic. It was hoped that a User Interface Management System (UIMS) could be created that would manage the details of the user interface just like the Database Management Systems (DBMSs) had abstracted many of the details of dealing with large quantities of data. One of these early systems was Mickey [22], 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.

Jade [38] 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 references, 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.

Systems of the late 80's and early 90's, such as UIDE [31], HUMANOID [34] and ITS [42] 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 [6], and the DON system [15] 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 [30], 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 outside the domain of what I propose to explore for this thesis.

ITS 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" [42] 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.

HUMANOID [34] 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.

The MASTERMIND project [35] 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 [33] 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.

TRIDENT [39], 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. While the PUC system will not expect the end-user to be involved in every phase of the design process, several of TRIDENT's automated techniques will be used or extended. For performing layout, TRIDENT specified 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. The PUC system does not currently generate interfaces using any of TRIDENT's techniques, but I may use them in future versions of the interface generators.

In addition to the ACG, there are a number of languages for specifying task models. The formal specification language LOTOS [14] has been used for modeling tasks, and GOMS [4] can also be used. ConcurTaskTrees [24] 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/what is performing the task, whether it be the user, the system, or an interaction between the two. ConcurTaskTrees or one of these other languages will be the basis of the distributed task modeling language used by the PUC system.

UIML [1] is an XML language that claims to provide a highly-device independent method for user interface design. UIML differs from the PUC in its tight coupling with the interface. UIML specifications can define the types of components to use in an interface and the code to execute when events occur. The PUC specification language leaves these decisions up to each interface generator. Some work is currently underway to make UIML more device independent by including more abstract information such as task models in the interface description [16].

Recently a new general purpose language has been introduced for storing and manipulating interaction data. The eXtensible Interface Markup Language (XIML) [27] is an XML-based language that is 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.

A new system named SUPPLE was recently created for automatically generating layouts for user interfaces [9]. SUPPLE uses a branch-and-bound search to find the optimal choice of controls and their arrangement. Optimality is determined by a cost function that takes into account user navigation and a matching function that depends on the choice of a control for a given abstract interface element. 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 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.

3. Preliminary User Studies

Much of the related work shows that automatically generating interfaces is a hard problem, and no previous system has successfully automatically created user interfaces measured to be of high-quality. The problem can be broken down into two sub-problems: determining what information an abstract appliance specification should include, and building an interface generator that can design usable and aesthetically pleasing interfaces from those abstract specifications. As a beginning to solving these problems, I started by hand-designing remote control interfaces for appliances (rather than begin with designing the appliance specification language). Then user studies were conducted to compare the hand-designed interfaces to the manufacturers' interfaces (full results of this study are described elsewhere [19]). This approach allowed me to concentrate on what functional information about the appliance is necessary to create a usable interface and to show that a PUC controller could be easier to use than interfaces on actual appliances.

We chose to focus on two common appliances for our hand-designed interfaces: the Aiwa CX-NMT70 shelf stereo with its remote control, and the AT&T 1825 telephone/digital answering machine. We chose these two appliances because both are common, readily available, and combine several functions into a single unit. I own the Aiwa shelf stereo that we used, and the AT&T telephone is the standard unit installed in many offices at Carnegie Mellon. Aiwa-brand stereos seem to be particularly common (at least among our subject population) because ten of our twenty-five subjects owned Aiwa systems.

The hand-designed interfaces were created in two phases, initially as paper prototypes for a PalmOS device and later as Visual Basic implementations on a Microsoft PocketPC (see Figure 3). Each interface supported the complete set of appliance functions. At each phase, the interfaces were iteratively improved with heuristic analyses, followed by a user study. The user study was dual-purpose: to compare the hand-designed interfaces with the interfaces on the actual appliances and to see what problems users had with the hand-designed interfaces.

Unfortunately, it was not possible to use the PocketPC to actually control either of the appliances, but I still wanted the users of the hand-designed interfaces to receive feedback from their actions in a manner that was consistent with the appliances. Control was simulated for the users using a wireless network from our PDA to a laptop. The laptop was connected to external speakers, and it generated audio

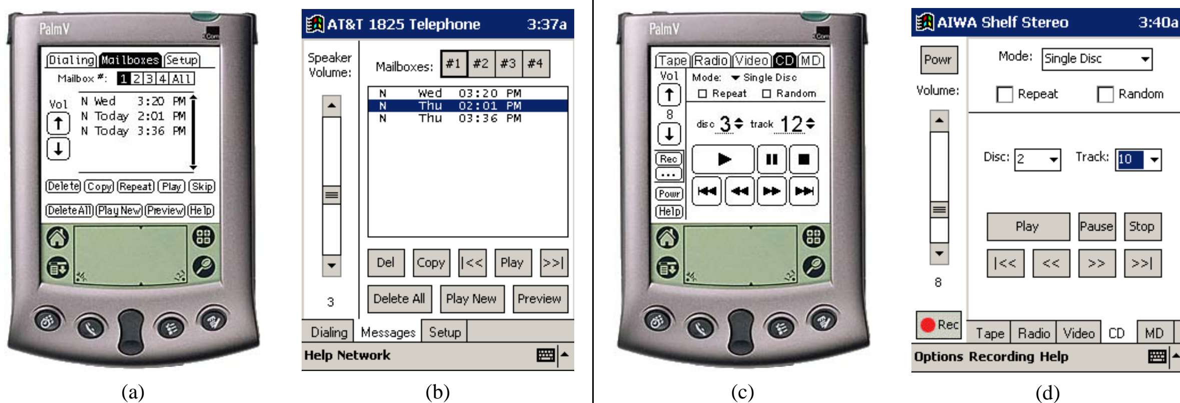

Figure 3. Hand-designed interfaces for the phone (a-b) and stereo (c-d). Functional interfaces for the PocketPC are shown in b and d, and paper prototype interfaces for the Palm are shown in a and c.

feedback that was consistent with what would be expected if the PocketPC were actually controlling the appliances.

Two between-subjects comparison studies were conducted of the hand-designed PDA interfaces that we created and the interfaces on the actual appliances. 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.

In order to compare user interface performance 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.

The results of both studies indicate that subjects made fewer missteps ($p < 0.05$) and asked for help less ($p < 0.001$) using the hand-designed interfaces than using the actual appliances. The first study did not measure the time to perform the tasks (because of the paper shuffling for the prototype interfaces), but the second study found that users of the hand-designed interfaces were 1.5 to 3 times faster ($p < 0.001$). The results of the second study are shown in Figure 4. In summary, 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 handheld interfaces.

Users had great difficulty using the actual appliances, but were able to understand and operate the hand-designed interfaces with reasonable ease. One exception to this was found in the paper 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.

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 some of this large difference on the actual appliances.

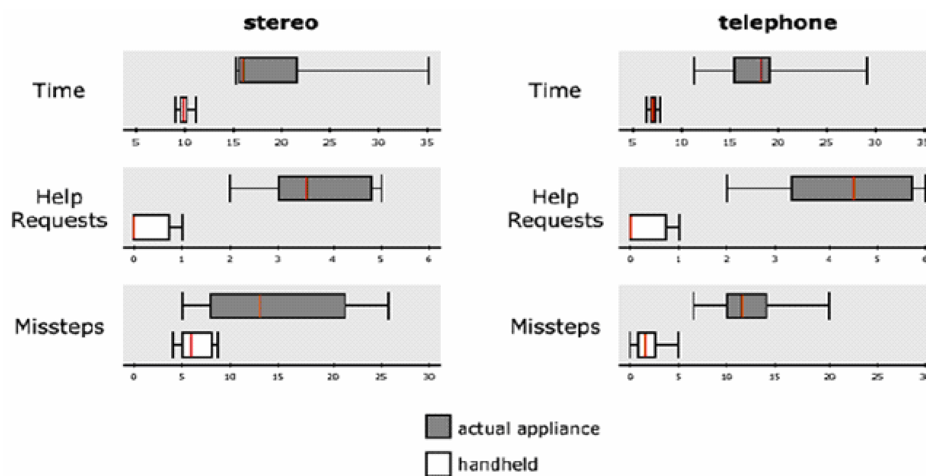

Figure 4. Box-plots of results from the second user study.

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 and vice versa. Even the numeric access codes for the answering machine must be set using arrow buttons rather than the phone number-pad. All but one subject tried to use the number-pad 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.

4. System Architecture

The PUC architecture is designed to allow users to control appliances in their environment through an intermediary user interface. The intermediary interface might be a graphical user interface on a handheld computer or a mobile phone, or it could be a speech interface that uses microphones in the room. When a

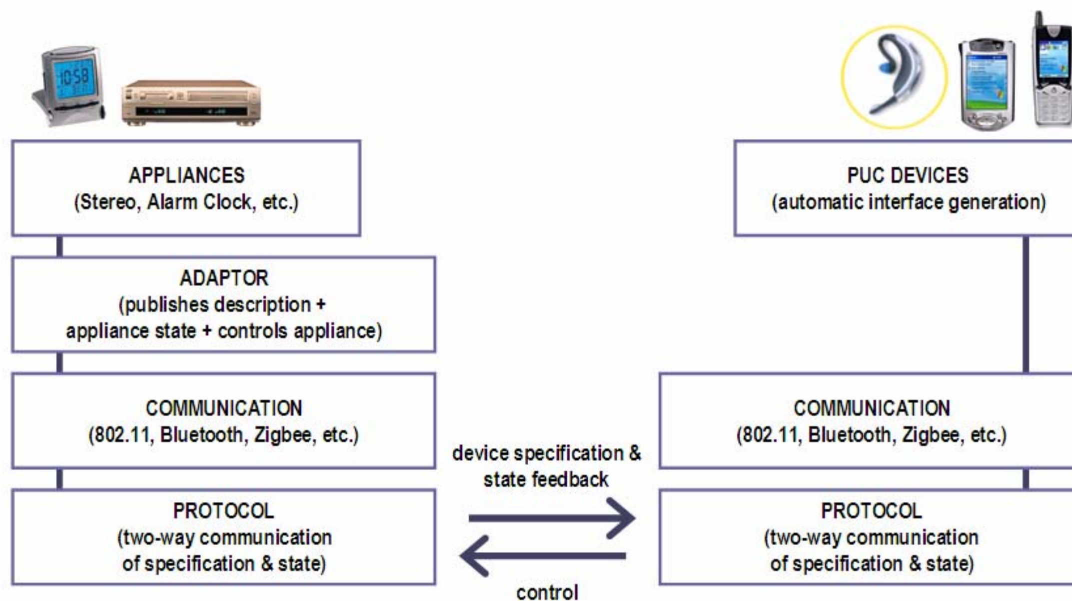

Figure 5. An architectural diagram of the PUC system showing one connection (multiple connections are allowed at both ends).

user decides to control an appliance, their controller device would download from that appliance an abstract functional description and use that description to automatically generate an interface for controlling that appliance. A two-way communication channel between the controller and the appliance allows the user's commands to be sent to the appliance and feedback to be provided to the user.

The PUC system has four parts: a specification language, a communication protocol, appliance adaptors, and interface generators (see Figure 5). All of these pieces are described in more detail elsewhere [20]. Automatic generation of user interfaces is enabled by the specification language, which allows each appliance to describe its functions in an abstract way. Our goal in designing this language was to include enough information to generate a good user interface, but not include any specific information about look or feel. Decisions about look and feel are left up to each interface generator. Included in our language are state variables and commands to represent the functions of the appliance, a hierarchical "group tree" to specify organization, dependency information that defines when states and commands are available to the user based on the values of other states, and multiple human-readable strings for each label in a specification. The appropriate string is chosen based upon the available space in the automatic layout.

The communication protocol allows controller devices to download specifications, send control messages, and receive feedback messages that report the state of the appliance. The two-way nature of this protocol allows the PUC to provide a better user interface than an ordinary one-way remote control, because users receive feedback on the success of their actions.

One goal of our system is to control real appliances. Since there are no appliances available that natively implement our protocol, we must build *appliance adaptors*, i.e. a translation layer between our protocol and an appliance's proprietary protocol. We have built a number of appliance adaptors already, including a software adaptor for the AV/C protocol that can control most camcorders that support IEEE 1394 and another adaptor that controls Lutron lighting systems. We have also built hardware adaptors for appliances that do not natively support any communication protocol. We are also thinking about building general purpose adaptors to a emerging industry standards such as UPnP [36] or INCITS/V2's AIAP [12]. This architecture allows a PUC to control virtually any appliance provided the appliance has a built-in control protocol or someone has the hardware expertise to add one.

The last, but most important, piece of the PUC architecture is the interface generator. We have built interface generators on several different platforms, including graphical interface generators on PocketPC, Microsoft's Smartphone, and desktop computers, and a speech interface generator that uses the Universal Speech Interfaces framework [28]. Each interface generator uses a rule-based approach to create interfaces, extending the work of many previous systems [33]. The interface generators and the appliance specification are the focus of my thesis, and will be described in more detail in the next section.

5. Generating Interfaces from Appliance Descriptions

The PUC system automatically generates interfaces by using a set of rules to transform an appliance specification into a concrete interface. I expect that users will want to use the generated interfaces immediately, which puts several constraints on our design. First, the generated interfaces must be usable because users will not be willing to take the time to modify a user interface in order to complete their task. The interface generation process must also be accomplished quickly, because it is not reasonable to expect the user to wait several minutes to use their copy machine or turn on their television. The generated interfaces must also be able to control the complete functionality of the appliance, because users will get frustrated if they cannot access the feature that they want to use. This section describes the design choices made and the features included to ensure that the PUC system is able to meet these requirements.

When designing the appliance specification language, a number of design choices were made to ensure that appliance specifications were easy to author and to address the requirements mentioned above. A few of the most important decisions made were:

No specific layout information: Including specific layout information in the specification has several disadvantages: appliance specifications get longer because they may contain multiple complete concrete designs and they will not necessarily be forward compatible with new handheld devices that have different characteristics than current devices. Omitting layout information also means that the PUC system can support features such as ensuring consistency across user interface for similar appliances, generating both graphical and speech interfaces from the same specification, and creating a single user interface to multiple connected appliances. These features are possible because each interface generator makes its own decisions about interaction style and the positioning of UI controls.

Objective vs. subjective information: The information contained in each appliance description could be categorized as either objective or subjective. Objective information can be extracted from the behavior of the actual appliance circuits and code, or by manipulating the appliance and observing its output. Subjective information may be biased by the opinions and taste of the appliance specification author. It is important to find a good balance between the amount of objective and subjective information that is included in a specification language.

Only one way to specify anything: I have tried to ensure that the specification language does not allow more than one way to specify the same information. Not only does this make the rules for interface generators easier to implement, but it should also make specifications easier to author.

The specification language has these features:

State Variables, Commands, and Explanations: The most primitive descriptive elements in the specification language are state variables, commands, and explanations. I have found, as have others, that most of the manipulable elements of an appliance can be represented as state variables. Each variable has a primitive type that tells the interface generator how it can be manipulated. For example, a radio station state variable would have a numeric type, and the interface generator could infer the tuning function because it knows how to manipulate a numeric type. Other state variables could include the current track of the CD player and the status of the tape player (stop, play, fast-forward, etc.). However, state variables are not sufficient for describing all of the functions of an appliance, such as the seek button on a radio. Pressing the seek button causes the radio station state variable to change to some value that is not known in advance. To represent the seek function "commands" are needed, parameter-less functions whose results cannot be described easily in a specification. Unlike state variables and commands, explanations are not manipulable by the user. 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 is used to explain this. Explanations are used very rarely in specifications that we have written.

Group Tree and Complex Types: Proper organization is important for a well-designed user interface. The PUC specification language uses an *n*-ary tree to represent this organization, just as in many previous systems, with a state variable, command, or explanation at every leaf node (leaf nodes may be present at any level of the tree). Each branching node is a "group," and each group may contain any number of state variables, commands, and other groups. There are two novel aspects to the PUC group tree:

1) The group tree is subjective information and thus considered to be unreliable. Where possible, the interface generators use other information from the specification (such as dependency information, discussed below) along with the group tree to reach the final concrete organization. For this reason, 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 components across two screens. Interface generators for larger screens can ignore the deeper branches in the group tree and put all of the components onto a single panel.

2) The group tree is used to specify complex data types, such as lists and unions. The state variable types might have been extended to include these complex structures instead, but I chose this approach to limit the number of ways in which types can be specified. In particular, if each state variable was allowed to have a complex type, then specification authors could have defined several closely-related elements as a state variable with a record type containing several fields or as a group containing multiple state variables. As discussed above, limiting the number of ways to specify a piece of information has advantages for both interface generators and specification authors.

Labels: The interface generator must be able to appropriately label the interface components that represent state variables and commands. Providing this information is difficult because different form factors and interface modalities require different kinds of label information. An interface for a mobile web-enabled phone will probably require smaller labels 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. The PUC specification language provides this information with a generic structure called a *label dictionary*. The assumption underlying the label dictionary is that every label contained within, whether it is phonetic information or plain text, will have approximately the same meaning. Thus the interface generator can use any label within a label dictionary interchangeably. For example, this allows a graphical interface generator to use a longer, more precise label if there is lots of available screen space, but still have a reasonable label to use if space is tight.

Dependency Information: The two-way communication feature of the PUC allows it to know when a particular state variable or command is unavailable. This can make interfaces easier to use, because the components representing those elements can be disabled. The specification contains formulas that specify when a state or command will be disabled depending on the values of other state variables, currently specified with several types of dependencies, including: equal-to, greater-than, less-than, and is-defined. Each state or command may have multiple dependencies associated with it, combined with the logical operations AND and OR. These formulas can be processed by interface generators to determine whether a component should be enabled when the appliance state changes. Dependency information can also be useful for making decisions about the structure of a user interface, as described below.

I have developed a number of interface generator rules that help transform appliance specifications into concrete user interfaces. In my implementation, rules are grouped into rule phases that accomplish particular parts of the interface generation process. There are currently five phases in the PocketPC interface generator: extracting and organizing information from the specification, choosing UI controls, building an abstract user interface, building a concrete user interface, and fixing layout problems in the concrete interface. Interface generators for other platforms have a similar set of phases and may share some rules within each phase, but each also has its own unique rules, especially as the process gets closer to building the concrete user interface. Where possible, I have tried to leverage previous work in the interface generator. For example, a decision tree is used to choose controls for interface elements, which was first used in the UIDE system [6] and later elaborated in TRIDENT [40].

One of the more interesting rules uses dependency information to determine the panel structure of a user interface. The use of different panels can be beneficial to a graphical interface. Commonly-used or global controls can be made available in a sidebar where they are easily accessible. Controls that are only available in the current appliance mode might be placed on a panel in the middle of the screen that

changes with the mode, hiding the functions of other modes that are not available. Dependency information can help identify situations like these because it gives objective information about when appliance functions will be available. For example, if an interface generator finds that two sets of controls are never available at the same time, then the generator can place them on overlapping panels. Controls that have no dependencies on other state variables might be placed in a sidebar. Specific details of how this algorithm is implemented are available elsewhere [20].

6. Handling Domain-Specific and Conventional Knowledge

A common problem for automatic interface generators has been that their interface designs do not conform to domain-specific design conventions to which users are accustomed. 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 [42] have dealt with this problem by defining specific rules for each application that apply the appropriate design conventions. Other systems [15] 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 time or desire to modify each interface after it is generated, especially if the interface was generated in order to perform a specific task.

I have developed one solution to this problem called *Smart Templates* [21], which augments the PUC specification language's primitive type information with high-level semantic information. For example, the `media-controls` template defines that a state variable with particular attributes controls the playback of some media. PUC interface generators can use the information added by a Smart Template to apply design conventions and make interfaces more usable. If an interface generator does not recognize a template however, a user interface can still be created because Smart Templates are constructed from the primitive elements of our specification language. Figure 6 shows the same instance of a Smart Template rendered on different platforms.

An important innovation is that Smart Templates are parameterized, which allows them to cover both the common and unique functions of an appliance. For example, the media playback template supports play and stop, but also optional related functions such as next track for CDs, fast-forward and reverse-play for

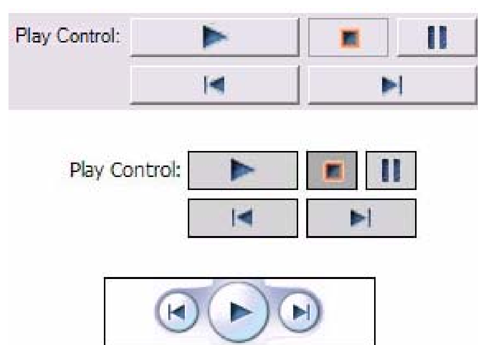

Figure 6. Media controls rendered for a Windows Media Player interface on each of our three platforms. At the top is the desktop, the middle is PocketPC, and the bottom shows Smartphone. The Smartphone control maintains consistency for the user by copying the layout for the Smartphone version of Windows Media Player, the only media player application we could find on that platform. This interface overloads pause and stop onto the play button.

Figure 7. Different arrangements of media playback controls automatically generated for several different appliances from a single Smart Template (`media-controls`).

tape players, and "play new" for phone answering machines (see Figure 7). Smart Templates also give appliances the flexibility to choose a functional representation that matches their internal implementation. For example, our `time-duration` Smart Template allows single state variables with integer or string types, or multiple state variables (e.g. a state for hours and another for minutes). Allowing multiple representations for Smart Templates is the only case where the principle of "one way to specify anything" is explicitly violated. In this case it seems better to support the different mechanisms that appliances might use internally to represent high-level data, rather than restrict appliances to just one representation.

I have built a preliminary implementation of Smart Templates into the existing PUC system. So far the PUC supports a few of the many Smart Templates that the PUC system will need: `media-controls`, `time-duration`, `image`, and `image-list`. Many more will need to be implemented to show that the technique is effective. I have developed a list of about ten Smart Templates that should be implemented, including `date`, `mute`, `power`, and `volume`, and I expect the list to at least double in size as I look at new and different appliances. I also expect that some Smart Templates will naturally combine with others to create new templates. For example, date and time are often used together, as are volume and mute. I hope to implement Smart Templates in such a way that templates can be flexibly combined with less work than creating a new template from scratch.

My final implementation of Smart Templates will also provide several other advantages for interface generators. For example, certain Smart Templates might provide a connection to data already stored on the PUC device. `Address` or `phone-number` Smart Templates could connect to the contact database on a PocketPC, and `date` or `time-absolute` Smart Templates could connect to the calendar database. This would allow users to easily enter addresses into a navigation system in a car, or to set their coffee pot to start making coffee based on their first appointment of the day. Smart Templates may also be useful for assigning particular appliance functions to physical interaction objects on a handheld device, such as a physical button or jog dial. For example, a physical jog dial might be used if an appliance has a `volume` Smart Template.

7. Interface Consistency

Many books on user interface design contain one or more guidelines related to keeping user interface consistent [7, 29]. The benefit of consistency is that users can leverage knowledge from their previous experiences with one part of an interface to help them use another part of that interface (*internal consistency*) or another user interface (*external consistency*). Consistency is widely used in interface design today from desktop applications to the physical panels of an appliance. An example of internal consistency can be found on nearly all Nokia mobile phones, which allow users to press number keys to jump to a particular item in any menu. Nearly all Windows applications have external consistency in the location of the menu bar and the particular menus that are available, such as *File* and *Edit*.

The PUC system has a unique opportunity to ensure external consistency between all of the interfaces that a user generates, because PUC users have their own personal devices that they use. This allows the PUC controller device to remember previously generated interfaces and record usage statistics for those interfaces, which can then be used to ensure that newly generated interfaces are consistent with older ones. For example, the interface that I generated for my new car stereo will probably be more usable if its layout and organization is similar to my home stereo interface that I use frequently.

The user interfaces generated by a PUC can be made consistent:

1. with other applications on the same controller device

2. with interfaces generated in the past for appliances with similar functions

The first can be achieved by using the standard toolkit available on the controller device, and using generation rules that match the device's UI guidelines. Each of my interface generator implementations produces interfaces that are consistent with other applications on its platform. The second is more

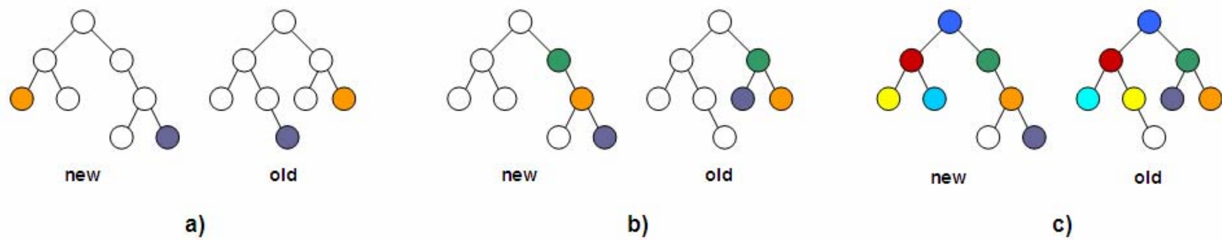

Figure 8. Examples of the three different levels of similarity. The trees represent the structure of the user interface as given by the appliance specification, with one tree representing the new specification and the other a previously generated specification. Nodes with the same shading indicate functions that were found to be similar across both appliances. a) sparse similarity: the appliances have a small number of similar functions spread throughout the tree. b) branch similarity: the appliances have a number of similar functions in one branch of the structure. c) significant similarity: the appliances share mostly the same functions, though they might be organized differently.

challenging and can be broken down into two sub-problems: finding previously generated interfaces that are relevant, and determining how to integrate decisions from the past interfaces into the new interface.

A relevant previously generated interface must include some of the same functionality as a new interface that is being generated. Unfortunately, it is difficult to conclusively know whether two functions on different appliances are the same. The only way to be certain that two functions have the same meaning is if they are part of the same Smart Template. Functions may also be similar if their corresponding state variables or commands have the same name or share some of the same labels. Type information from state variables and dependency information may also be used to gauge similarity, but neither of these by themselves can tell us anything conclusive. For example, a volume function might be represented by an integer ranging from 0-50 on one appliance and as an enumeration with 10 possible values on another appliance. I am exploring algorithms for solving this problem that compare all of the parameters of a function at once, such as nearest neighbor algorithms [5].

Once previous interfaces with similar functions have been found, the interface generator can examine those interfaces and decide how to make the new interface consistent. Choosing the right technique to make the interfaces consistent will depend in part on an estimation of how familiar the user is with the previous appliance interfaces. The interface generator will want to focus on the interface that the user has the most experience with. Furthermore, if the user is not at all familiar with the relevant pieces of the old interface, then it may not be worth the extra generation expense to make the new interface consistent. Usage statistics will be stored for every user interface to assist in this part of the algorithm. Finding the right technique to ensure consistency will also depend on how similar the previous appliances are to the new appliance. So far I have identified three levels of similarity between the old and new appliances (see Figure 8), each of which will use a different technique to achieve consistency.

Appliances with *sparse similarity*, those which only share a few unrelated functions, will try to represent each function with the same interface controls that the user saw in the older interface. This might mean that a volume control would be represented as a scroll bar instead of a combo box, or vice versa. The decision to use a different control than normal will be based on how similar the two functions are and how much control could be lost. For example, it would not make sense to substitute a combo box for a scroll bar if the underlying state is an integer ranging from 0-100, because combo boxes are unwieldy for interacting with many items.

Appliances with *branch similarity*, those which share a group of related functions, will try to replicate in the new interface the layout and organization of the related functions in the previous interface. This might mean that the panel for controlling the CD player in a sophisticated alarm clock would look the same as the CD player interface on a car stereo. If the previous CD player interface had been on a tabbed panel, then the new appliance interface might be organized using tabs to match the previous organizational

layout. Such a feature would help users find similar functions the same way in many, if not most, of their interfaces.

Appliances with *significant similarity*, those which share most of the same functions, will try to use the same layout and organization in the new interface that the user has seen in previous interfaces. This would mean that the user interfaces for different navigation systems in two different cars might have nearly identical user interfaces on a PUC device. My approach for this technique will be to take the group tree for the previous interface and replace the functions of the older interface with those from the new appliance. Functions that were not present in the older interface will be integrated as best as possible.

All of these techniques must be applied carefully, especially taking into account the degree of similarity between functions. I plan to conduct a study to ensure that my techniques create consistent interfaces and that they improve usability.

8. Generating Interfaces for the "Experience"

A novel feature of the PUC system will be its ability to generate a single user interface for multiple appliances that have been connected together. One example of a use for this feature is a typical home theater, which includes separate VCR, DVD player, television, and stereo appliances, but might be more easily thought of as a single integrated system. A PUC interface for a home theater would ideally have features like a "Play DVD" button that would turn on the appropriate appliances, set the TV and stereo to the appropriate inputs, and then tell the DVD player to "Play."

A key question is how to model the connections between appliances and the interactions that users have that span appliances. Ideally a wiring diagram showing how each appliance physically connects to the others will be the only piece of system-specific modeling that is required. Tasks that users want to perform might be assembled from the wiring diagram and sub-tasks that are stored as part of each appliance's specification. I plan to develop a new *distributed* task modeling language, based on previous languages such as ConcurTaskTrees [24], to facilitate this process.

9. Outside the Scope

Work on the PUC system could be taken in many directions. This section describes issues related to the PUC system that I will *not* be looking at for my thesis:

> **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 [30]. The PUC system would certainly benefit from a help system, but the previous work in this area is extensive and it is unlikely that I would create anything new of research value.
>
> **Automated trouble-shooting for complex systems:** The functionality of a complex system, like the home theater system described in the previous section, can often depend 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 may 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 help users find solutions for their particular systems. This is something that I will not be looking at as part of my thesis work however.
>
> **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 [36] and JINI [32]. The PUC system will rely on existing techniques and not attempt to further this research.

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

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

Inter-operability with INCITS/V2: The INCITS/V2 standard shares many features with the PUC standard. I hope to collaborate with the creators of this standard in the future, but I do not intend to make the PUC interoperate with the standard. Too much work would be required to ensure interoperability, and this work could be better focused on other tasks of research value.

10. Validation

I have two goals for the PUC system:

- **Breadth:** The appliance specification language is capable of describing a wide range of appliances.
- **Quality:** Interfaces generated for specifications across that range beat the usability of the manufacturers' interfaces for the same appliances.

I tried to choose my goals to be achievable but interesting. The goals are interesting because, as far as I know, no formal user-centric evaluation of a system for automatically generating interfaces has ever been conducted. The goals are achievable because though automatically generating high-quality interfaces from an abstract specification is a difficult task, the user interfaces created by most appliance manufacturers are not usually very easy-to-use. I also chose my goals to mirror the user studies that I conducted at the beginning of the PUC project (described in section 3), where I compared actual appliance interfaces with interfaces that I created by hand. As described above, the hand-designed interfaces showed a factor of two improvement in quality over the manufacturers' interfaces [19].

Table 1. Appliances that the PUC system will be validated against.

Already Specified & Generated	Propose to Specify
Axis Pan & Tilt Camera	GMC Denali Navigation System
Audiophase Stereo	Phone/Answering Machine
GMC Denali Driver Information Center	Windows Media Player with playlists
GMC Denali Climate Control System	Automated Teller Machine (ATM)
Sony Camcorder	Microsoft Windows XP Media Center PC
Elevator Simulation	Photocopier
Lutron Home Lighting System	TV Tuner
Windows Media Player without playlists	Personal Video Recorder (e.g. TiVO)
X10 Lighting	Powerpoint
	Alarm Clock
	Projector

One issue with my quality goal is defining "usability." In previous user studies, I have analyzed the usability of interfaces by giving users a set of tasks to perform and measuring certain aspects of their performance while completing the tasks. In particular, I looked at the time to complete the tasks, the number of times a user manual was needed, and the number of missteps taken. This approach seems well suited to comparing user interfaces with identical functions, such as two user interfaces for the same appliance, and is the approach I intend to use for evaluating the PUC system.

Another problem is that it is impossible to prove that either goal has been achieved with absolute certainty. Doing so would require writing a specification for every appliance in existence, generating an interface from that specification, and then performing a comparative user study to show matching usability. I plan to deal with this issue by developing a list of appliances that are interesting, either because of their complexity or some unique feature, and testing them against my breadth goal. I will then choose a subset of those appliances, probably two or three, to test against my quality goal.

Table 1 shows the list of appliances that I have assembled so far, separated by those I have actually written specifications for. Note that a "GMC Denali" is a large sport utility vehicle manufactured by General Motors. I plan to develop this list further as my thesis work progresses.

11. Schedule

Figure 9 shows a graphical depiction of my proposed schedule. My goal is to finish in December 2005, in about one year and nine months. The schedule is divided into three rough phases. In the first phase, from now until October, I will design and implement algorithms for generating consistent interfaces and continue to build on my existing Smart Template work. During this period I will also do a preliminary user interface comparison study as part of an on-going collaboration with General Motors. In the second phase, from October until June 2005, I will design and build a sub-system for generating a single user interface for multiple appliances. This will likely involve designing a new distributed task modeling language. During the first two phases, I will continue to improve the robustness and quality of the generated interfaces on each of the platforms that I support. In the third and final phase, from June until the end of 2005, I will analyze my system by conducting an extensive set of user studies on several complex appliances and write my dissertation.

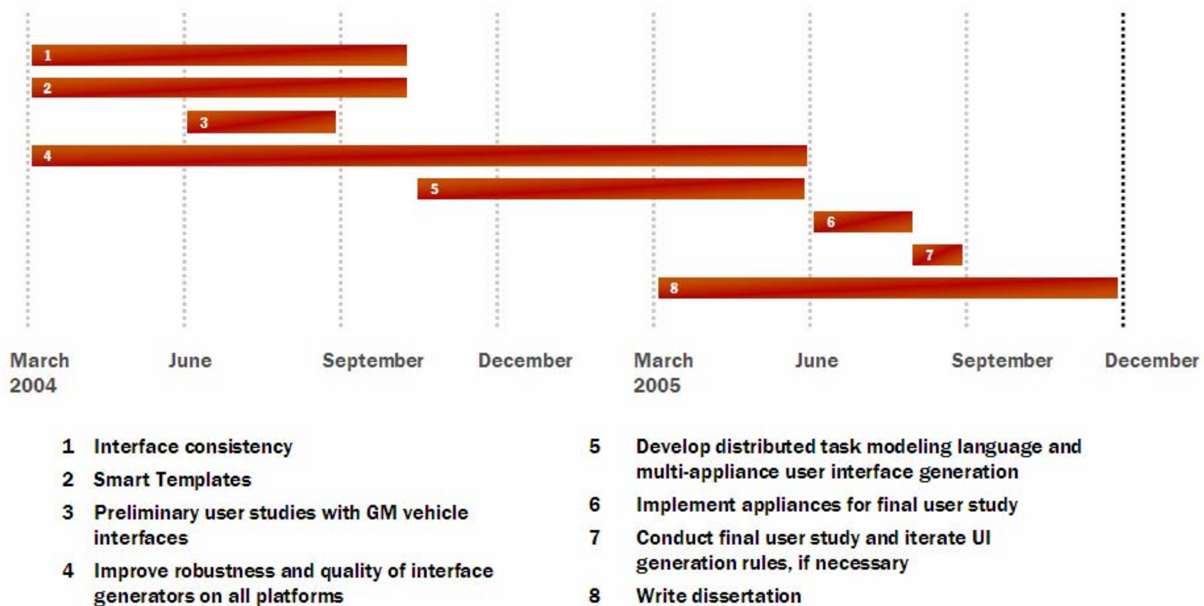

Figure 9. Proposed schedule for completion of my thesis

12. Conclusion

Most appliances on the market today are computerized, and within the next ten years most appliances will also feature networking technologies. Unfortunately, the user interfaces for most of these appliances will be complicated and difficult to use. I am proposing to develop a system called the *personal universal controller* that will address this problem by moving the user interface from the appliance to an intermediary "user interface device." The contributions of my thesis work will be:

- An abstract appliance modeling language for describing the complete functionality of a wide-range of appliances
- Algorithms for automatically generating high quality interfaces for each appliance from that language.
- 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 interfaces modalities.
- Algorithms for determining the similarity between a new appliance specification and specifications that have been used in the past, and algorithms that use that similarity information to apply design decisions from previously generated interfaces in the generation process for a new interface.
- A *distributed* task-modeling language for describing the sub-tasks specific to an appliance in a multi-appliance system. The language will also contain special linking information that allows inter-appliance tasks to be inferred from each appliance's sub-tasks.
- Algorithms for automatically generating a single user interface for multiple appliances using distributed task information.
- Interface generation software on multiple platforms: Microsoft PocketPC, Microsoft Smartphone, and desktop computers, which use the above algorithms to produce interfaces that are shown by user testing to be better than manufacturers' interfaces for the same appliances.

Over approximately the next twenty-one months, I will prove my thesis that:

> **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.**

13. Acknowledgements

I would like to thank Brad Myers for advising me on this work. This work is conducted as a part of the Pebbles [17] project, with the help of many other people including Michael Higgins and Joseph Hughes of MAYA Design, Kevin Litwack, Thomas K. Harris, Roni Rosenfeld, Mathilde Pignol, Rajesh Seenichamy, Pegeen Shen, Stefanie Shriver, and Jeff Stylos of Carnegie Mellon University. I would also like to thank Naomi Ramos, Daniel Avrahami, Gaetano Borriello, Andrew Faulring, James Fogarty, Krzysztof Gajos, Darren Gergle, Johnny Lee, Trevor Pering, Desney Tan, and Roy Want for their many comments and discussions on this work. This work was funded in part by grants from NSF, Microsoft, General Motors, Intel, DARPA, and the Pittsburgh Digital Greenhouse, and equipment grants from Mitsubishi Electric Research Laboratories, VividLogic, Lutron, Lantronix, IBM Canada, Symbol Technologies, Hewlett-Packard, and Lucent. Intel Corporation funded this work as part of a four-month internship at the Intel Research Seattle lablet. The National Science Foundation funded this work through a Graduate Research Fellowship for the first author and under Grant No. IIS-0117658. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

14. References

1. Abrams, M., Phanouriou, C., Batongbacal, A.L., Williams, S.M., and Shuster, J.E. "UIML: An Appliance-Independent XML User Interface Language," in *The Eighth International World Wide Web Conference.* 1999. Toronto, Canada: http://www8.org/ and http://www.uiml.org/.
2. Association, T., "AV/C Digital Interface Command Set," 1996. http://www.1394ta.org/.
3. Brouwer-Janse, M.D., Bennett, R.W., Endo, T., van Nes, F.L., Strubbe, H.J., and Gentner, D.R. "Interfaces for consumer products: "how to camouflage the computer?"" in *CHI'1992: Human factors in computing systems.* 1992. Monterey, CA: pp. 287-290.
4. Card, S.K., Moran, T.P., and Newell, A., *The Psychology of Human-Computer Interaction.* 1983, Hillsdale, NJ: Lawrence Erlbaum Associates.
5. Dasarathy, B.V., *Nearest neighbor (NN) norms : nn pattern classification techniques.* 1991, Los Alamitos, CA: IEEE Computer Society Press.
6. de Baar, D.J.M.J., Foley, J.D., Mullet, K.E. "Coupling Application Design and User Interface Design," in *Conference on Human Factors and Computing Systems.* 1992. Monterey, California: ACM Press. pp. 259-266.
7. Dix, A., Finlay, J., Abowd, G., and Beale, R., *Human-Computer Interaction.* 1993, New York: Prentice-Hall. 570.
8. Eustice, K.F., Lehman, T.J., Morales, A., Munson, M.C., Edlund, S., and Guillen, M., "A Universal Information Appliance." *IBM Systems Journal*, 1999. **38**(4): pp. 575-601. http://www.research.ibm.com/journal/sj/384/eustice.html.
9. Gajos, K., Weld, D. "SUPPLE: Automatically Generating User Interfaces," in *Intelligent User Interfaces.* 2004. Funchal, Portugal: pp. 93-100.
10. HAVi, "Home Audio/Video Interoperability," 2003. http://www.havi.org.
11. Hodes, T.D., Katz, R.H., Servan-Schreiber, E., and Rowe, L. "Composable ad-hoc mobile services for universal interaction," in *Proceedings of the Third annual ACM/IEEE international Conference on Mobile computing and networking (ACM Mobicom'97).* 1997. Budapest Hungary: pp. 1 - 12.
12. INCITS/V2, "Universal Remote Console Specification," in *Alternate Interface Access Protocol*2003. Washington D.C. pp.
13. Intrigue, "Harmony Remote Control Home Page," 2003. http://www.harmonyremote.com/.
14. ISO, *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on Temporal Ordering of Observational Behavior.* 1988.
15. Kim, W.C. and Foley, J.D. "Providing High-level Control and Expert Assistance in the User Interface Presentation Design," in *Proceedings INTERCHI'93: Human Factors in Computing Systems.* 1993. Amsterdam, The Netherlands: pp. 430-437.
16. Mir Farooq, A., Abrams, M. "Simplifying Construction of Multi-Platform User Interfaces using UIML," in *European Conference UIML 2001.* 2001. Paris: Harmonia & Aristotle.
17. Myers, B.A., "Using Hand-Held Devices and PCs Together." *Communications of the ACM*, 2001. **44**(11): pp. 34-41. http://www.cs.cmu.edu/~pebbles/papers/pebblescacm.pdf.
18. Nichols, J., and Myers, B.A., *Report on the INCITS/V2 AIAP-URC Standard.* February 9, 2004.
19. Nichols, J., Myers, B.A. "Studying The Use Of Handhelds to Control Smart Appliances," in *23rd International Conference on Distributed Computing Systems Workshops (ICDCS '03).* 2003. Providence, RI: pp. 274-279.
20. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Pignol, M. "Generating Remote Control Interfaces for Complex Appliances," in *UIST 2002.* 2002. Paris, France: pp. 161-170. http://www.cs.cmu.edu/~pebbles/papers/PebblesPUCuist.pdf.
21. Nichols, J., Myers, B.A., Litwack, K. "Improving Automatic Interface Generation with Smart Templates," in *Intelligent User Interfaces.* 2004. Funchal, Portugal: pp. 286-288.
22. Olsen Jr., D.R. "A Programming Language Basis for User Interface Management," in *Proceedings SIGCHI'89: Human Factors in Computing Systems.* 1989. Austin, TX: pp. 171-176.

23. Olsen Jr., D.R., Jefferies, S., Nielsen, T., Moyes, W., and Fredrickson, P. "Cross-modal Interaction using Xweb," in *Proceedings UIST'00: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 2000. San Diego, CA: pp. 191-200.
24. Paterno, F., Mancini, C., Meniconi, S. "ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models," in *INTERACT.* 1997. Sydney, Australia: pp. 362-269.
25. Philips, "Pronto Intelligent Remote Control," 2003. Philips Consumer Electronics: http://www.pronto.philips.com/.
26. Ponnekanti, S.R., Lee, B., Fox, A., Hanrahan, P., and T.Winograd. "ICrafter: A service framework for ubiquitous computing environments," in *UBICOMP 2001.* 2001. Atlanta, Georgia: pp. 56-75.
27. Puerta, A., Eisenstein, J. "XIML: A Common Representation for Interaction Data," in *7th International Conference on Intelligent User Interfaces.* 2002. San Francisco: pp. 214-215.
28. Rosenfeld, R., Olsen, D., Rudnicky, A., "Universal Speech Interfaces." *interactions: New Visions of Human-Computer Interaction*, 2001. **VIII**(6): pp. 34-44.
29. Shneiderman, B., *Designing the User Interface: Strategies for Effective Human-Computer Interaction, Second Edition.* 1992, Reading, MA: Addison-Wesley Publishing Company. 573.
30. Sukaviriya, P. and Foley, J.D. "Coupling A UI Framework with Automatic Generation of Context-Sensitive Animated Help," in *Proceedings UIST'90: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1990. Snowbird, Utah: pp. 152-166.
31. Sukaviriya, P., Foley, J.D., and Griffith, T. "A Second Generation User Interface Design Environment: The Model and The Runtime Architecture," in *Proceedings INTERCHI'93: Human Factors in Computing Systems.* 1993. Amsterdam, The Netherlands: pp. 375-382.
32. Sun, "Jini Connection Technology," 2003.
33. Szekely, P. "Retrospective and Challenges for Model-Based Interface Development," in *2nd International Workshop on Computer-Aided Design of User Interfaces.* 1996. Namur: Namur University Press. pp. 1-27.
34. Szekely, P., Luo, P., and Neches, R. "Beyond Interface Builders: Model-Based Interface Tools," in *Proceedings INTERCHI'93: Human Factors in Computing Systems.* 1993. Amsterdam, The Netherlands: pp. 383-390.
35. Szekely, P., Sukaviriya, P., Castells, P., Muthukumarasamy, J., Salcher, E. "Declarative Interface Models for User Interface Construction Tools: the Mastermind Approach," in *6th IFIP Working Conference on Engineering for Human Computer Interaction.* 1995. Grand Targhee Resort: pp. 120-150.
36. UPnP, "Universal Plug and Play Forum," 2003. http://www.upnp.org.
37. V2, I., "Information Technology Access Interfaces," 2002. http://www.ncits.org/tc_home/v2.htm.
38. Vander Zanden, B. and Myers, B.A. "Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces," in *Proceedings SIGCHI'90: Human Factors in Computing Systems.* 1990. Seattle, WA: pp. 27-34.
39. Vanderdonckt, J. "Knowledge-Based Systems for Automated User Interface Generation: the TRIDENT Experience," in *Technical Report RP-95-010.* 1995. Namur: Facultes Universitaires Notre-Dame de la Paix, Institut d' Informatique:
40. Vanderdonckt, J., "Advice-Giving Systems for Selecting Interaction Objects." *User Interfaces to Data Intensive Systems*, 1999. pp. 152-157.
41. Weld, D., Anderson, C., Domingos, P., Etzioni, O., Gajos, K., Lau, T., Wolfman, S. "Automatically Personalizing User Interfaces," in *Eighteenth International Joint Conference On Artificial Intelligence.* 2003. Acapulco, Mexico:
42. Wiecha, C., Bennett, W., Boies, S., Gould, J., and Greene, S., "ITS: A Tool for Rapidly Developing Interactive Applications." *ACM Transactions on Information Systems*, 1990. **8**(3): pp. 204-236.