Creating a Lightweight User Interface Description Language: An Overview and Analysis of the Personal Universal Controller Project

JEFFREY NICHOLS
IBM Almaden Research Center
and
BRAD A. MYERS
Carnegie Mellon University

Over six years, we iterated on the design of a language for describing the functionality of appliances, such as televisions, telephones, VCRs, and copiers. This language has been used to describe more than thirty diverse appliances, and these descriptions have been used to automatically generate both graphical and speech user interfaces on handheld computers, mobile phones, and desktop computers. In this article, we describe the final design of our language and analyze the key design choices that led to this design. Through this analysis, we hope to provide a useful guide for the designers of future user interface description languages.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Graphical user interfaces (GUIs)*

General Terms: Design, Human Factors

Additional Key Words and Phrases: User interface description language (UIDL), automatic interface generation, model-based user interfaces, model-based authoring, remote control, appliances, handheld computers, personal digital assistants (PDAs), mobile phone, personal universal controller (PUC), consistency, Pebbles

This work was funded in part by grants from the Pittsburgh Digital Greenhouse, Microsoft, General Motors, and by the National Science Foundation under Grants No. IIS-0117658 and No. IIS-0534349. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsoring organizations.
Authors' addresses: J. Nichols, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95110; email: jwnichols@us.ibm.com; B.A. Myers, Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburg, PA 15213; email: bam@cs.cmu.edu.

DOI 10.1145/1614390.1614392 http://doi.acm.org/10.1145/1614390.1614392

ACM Reference Format:
Nichols, J. and Myers, B. A. 2009. Creating a lightweight user interface description language: An overview and analysis of the personal universal controller project. ACM Trans. Comput.-Hum. Interact. 16, 4, Article 17 (November 2009), 37 pages.
DOI = 10.1145/1614390.1614392 http://doi.acm.org/10.1145/1614390.1614392

1. INTRODUCTION

It has long been a goal of researchers to develop a User Interface Description Language (UIDL) that can describe a user interface without resorting to low-level code. A UIDL can reduce the amount of time and effort needed to make user interfaces by providing useful abstractions and supporting automation of the design process. For example, this might allow the same interface description to be rendered on multiple platforms. We have found the UIDL-based approach to be particularly beneficial for automatically generating interfaces that are personalized to individual users. For example, our Personal Universal Controller (PUC) system has the ability to generate new interfaces that are personally consistent with interfaces with which the user has interacted previously [Nichols et al. 2006a]. Personalization has also been investigated in a number of other domains, such as generating interfaces customized to an individual user's particular disabilities [Gajos et al. 2007].

We developed a UIDL for describing appliance user interfaces as part of the PUC project [Nichols et al. 2002], which was a component of the Pebbles research project [Myers et al. 2004]. The goal of the PUC project is to provide users with user interface devices that can remotely control all of the appliances in the users' environments. We imagine that these user interface devices would run on a variety of platforms, including handheld devices with graphical interfaces and hidden PCs with speech recognition software. To remotely control an appliance, the user interface device engages in two-way communication with the appliance, first downloading a description of the appliance's functions written in our UIDL, and then automatically creating a high-quality interface. The device sends control signals to the appliance as the user interacts with the interface, and also receives feedback on the changing state of the appliance. Automatic generation of the appliance user interface allows the PUC to create interfaces that are customized to the platform of the controller device [Nichols et al. 2002], the user's previous experience [Nichols et al. 2006a], and all the appliances that are present in the user's current environment [Nichols et al. 2006b].

The UIDL that we have designed, which we often refer to as our appliance specification language or just specification language, is a very important part of the PUC system. Over the six years we worked with the language, we identified a number of advantages to our design.

—High quality interfaces can be generated from specifications written in our language. These automatically generated interfaces have been shown in user studies to be better than the existing appliance interfaces [Nichols et al. 2007].

—The UIDL is easy to learn and use as shown through a user study and our own experience. Subjects in a study were able to write complete specifications for a VCR after spending just an hour and a half reading a tutorial document about the language. New members of our research group learned the language in about a day and were proficient after about two weeks.

—The UIDL is capable of describing a wide range of complex appliances. Our group has written specifications for 33 appliances, including several with more than 100 functional elements. This includes appliances such as VCRs, all-in-one printers, an elevator, and the navigation system of a GMC Yukon Denali, as well as some desktop applications such as remote control interfaces for Windows Media Player and PowerPoint.

—User interfaces can be generated for multiple devices and modalities. Using the same specification, we have generated speech user interfaces and graphical user interfaces for desktop computers, handheld computers, and smart phones.

—The Smart Templates feature of our UIDL (see Section 6) allows the description of high-level features in terms that are convenient for the appliance, and also provides a structured workaround for situations in which the language may not be sufficiently expressive.

—The UIDL is similar in form to a database schema, allowing schema matching algorithms to be applied automatically to find similarities between specifications for different appliances. This is a key feature of the language that allows our PUC system to be able to automatically generate interfaces that are *consistent* with previous interfaces the user has seen [Nichols et al. 2006a].

—Logical dependencies expressed in our UIDL allow algorithms to reason about appliance behavior. We use this feature to find groups of functions that could never be enabled at the same time, which influences layout during interface generation [Nichols et al. 2002]. We also use an AI planning algorithm to automatically manipulate appliance functions based on high-level requests by users [Nichols et al. 2006b].

These advantages suggest that the PUC UIDL could be an important starting point for the design of future UIDLs that focus on describing appliance interfaces or other user interfaces with similar characteristics. We also believe there are lessons to be learned from the development of our language that will be useful to designers of future UIDLs that target other types of user interfaces. We found our design process to be particularly effective in generating a high-quality UIDL, and we made several key design decisions that allowed us to make the language expressive without making it unnecessarily complicated. Our process and these decisions might be applicable to the design of future UIDLs.

Although this article focuses on the PUC UIDL, it is important to note that the PUC system is a completely implemented end-to-end system. PUC interface generation software can be installed on handheld devices running the Windows Mobile operating system. We have created proxy software and hardware to

allow PUC devices to control real appliances, either by translating proprietary protocols built-in to the appliances or modifying their hardware to enable control. We have also created a communication protocol to allow handheld devices running our software to connect to real appliances, download a specification written in the PUC UIDL, and generate an interface. This interface can then be used to remotely control the appliance. Complete details are available in the first author's doctoral dissertation [Nichols 2006].

This article starts by discussing related work to our UIDL. Then we elaborate on the design principles for our language, followed by a description of a study we conducted to inform our design. Then the language is described in detail, followed by an analysis of the strengths and weaknesses of our design and design process.

2. RELATED WORK

Research in user interface description languages has a long history dating back to the User Interface Management Systems (UIMSs) developed in the mid-80's, such as COUSIN [Hayes 1985]. The original goal of these systems was to automate the design of the user interface so that programmers, who were typically not trained in interface design, could produce applications with high-quality user interfaces. This work led to the creation of systems in the late 80's and early 90's, such as UIDE [Sukaviriya et al. 1993], ITS [Wiecha et al. 1990], Jade [Vander Zanden and Myers 1990], and Humanoid [Szekely et al. 1992], which required designers to specify models of their applications that could then be used to automatically generate a user interface. The generated interfaces could often be modified by a trained interface designer to produce a final user interface. These interfaces were often called *model-based user interfaces* because of the models underlying their creation. Early work in this area has been surveyed in depth elsewhere [Szekely 1996].

These early model-based systems had several drawbacks. Most notably, creating the models that were needed for generating an interface was a very abstract and time-consuming process. The modeling languages had steep learning curves and often the time needed to create the models exceeded the time needed to manually program a user interface by hand. Finally, automatic generation of the user interface was a very difficult task and often resulted in low-quality interfaces [Myers et al. 2000].

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

—*Very large-scale user interfaces* assembled with existing techniques are difficult to implement and later modify, and detailed models of the user interface can help organize and partially automate the implementation process. The models can then be used to help designers revisit the interface and make modifications for future versions.

—A recent need for *device-independent interfaces* has also motivated new research in model-based user interfaces and specifically on fully automated generation. Work in this area has also explored applications of automatic

generation to create interfaces that would not be practical through other approaches.

Many recent systems have attempted to address both of these issues.

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

In the design of the PUC's specification language, we made an explicit decision not to include a task model. We made this decision for two reasons. First, in our analysis of appliances, it seemed that many tasks only required a single step and thus the task model would have added little additional information over our functional model. Second, task models can be difficult to create, often involving hours of observation of people using a similar system. Training is also often needed to translate observations into a correct and robust task model. We anticipated that PUC specifications would be written by engineers at appliance manufacturers and not by trained human interface experts with experience in task modeling, thus a functional model seemed more appropriate.

Mobi-D [Puerta 1997] is model-based user interface development environment capable of producing very large-scale user interfaces. The Mobi-D development process differs from previous systems in that a series of declarative models were created iteratively, starting with models of the users and their tasks and ending with a presentation model that represented the final interface. All of these models were stored together and many relations were defined between the different models to assist the system and designer with interface building and maintenance. Mobi-D's process favored having a designer involved in all aspects of the design process. Work on Mobi-D led to the design of the eXtensible Interface Markup Language (XIML) [Puerta 2002], which is a general purpose language for storing and manipulating interaction data. XIML can store interaction data, many different types of user interface models, and relations between the models and data. It may be possible to express the information in the PUC specification language within an XIML document, but XIML also supports many other types of information that will not be needed by the PUC, such as concrete descriptions of user interfaces.

The User Interface Markup Language (UIML) [Abrams et al. 1999, 2007] claims to provide a highly device-independent method for user interface design, but it differs from the PUC in that its specifications can define the types of components to use in an interface and the code to execute when events occur. Since the original design of UIML, several extensions have been made to improve its device independence. The TIDE interface design program [Ali et al. 2002] requires the designer to specify the interface more generically first, using a task model. Then the task model is mapped, with the designer's assistance, into a generic UIML model which is then further refined into a specific user interface. Other work has explored the creation of a generic UIML mapping that can be adapted to VoiceXML, WML, HTML, and other interface types [Simon et al. 2004]. Some recent work integrates the UIML language with an engineering tool called LiquidUI [Abrams et al. 2007], which seems to be similar to previous systems such as Mobi-D.

A more substantial extension to UIML that deserves additional discussion is the Dialog and Interface Specification Language (DISL) [Schaefer et al. 2006]. The goal of DISL is to support creation of interfaces for multiple devices and modalities. The designers of DISL modified UIML in two ways to support these goals: (1) replacing UIML's concrete widgets with more generic versions similar to the work of Simon et al. [2004], and (2) replacing UIML's behavior specifications with a new model that includes features such as state variables and abstract descriptions about how these variables change. The PUC specification language also includes state variables, but has built-in assumptions about how these variables can change rather than requiring the specification author to describe the possible changes in detail. We believe our design decision simplifies our language and makes our specifications easier to write, even though it gives up some descriptive power. For our domain of interest, we did not find it necessary to support more detailed descriptions of behavior as included in DISL.

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

Microsoft's eXtensible Application Markup Language (XAML) [Microsoft 2006], Mozilla's XML User interface Language (XUL) [Bojanic 2006], and Adobe's MXML language [Coenraets 2004] are three different languages for specifying a user interface. XAML is used in the .NET Framework and the Vista operating system to describe most graphics content that is rendered to the screen. XUL is currently used to define the user interfaces of most, if not all, Mozilla software products. MXML is a declarative language used for laying out interfaces and specifying some interface behavior in Adobe Flex applications.

Documents written in these languages are similar to the presentation models used by many model-based systems, which abstract some platform-specific elements but are typically fixed to one interface modality with restrictions on the supported form factors and input techniques. In this case, the languages are designed for desktop-size graphical interfaces. XUL has been shown to be beneficial for porting applications across various platforms of this type, including Windows, Linux, and Macintosh. The PUC specification language differs from these languages in that it describes appliance functionality without any specific details of the user interface, allowing the specification to apply for interfaces in different modalities and substantially different format factors with different input techniques.

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) initiated the V2 standardization effort [INCITS/V2 2003], which developed standards for a Universal Remote Console (URC) that enables 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. We have provided feedback to the V2 group in the past that influenced the current design of their specification. A detailed report is available analyzing the similarities and differences between the V2 and PUC systems [Nichols and Myers 2004].

3. DESIGN PRINCIPLES

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

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

—There should be *sufficient detail to generate a high-quality interface*. We conducted a user study, discussed in the next section, to determine how much detail would be needed in our specification language. Note that this principle is different than the first. It would have been possible to completely describe the appliance without the readable labels and adequate grouping information that are needed for generating a good user interface. For example, the

Universal Plug and Play (UPnP) standard [UPnP 2005] includes an appliance description language that does not include sufficient detail for generating good interfaces, such as human-readable labels, an organizational structure, or information suggesting how the various functions of an appliance might relate to each other.

—*No specific layout information* should be included in the specification language. We wanted to ensure that our language would be sufficiently general to work for interface generators running on a wide variety of platforms. Another solution for addressing the multiplatform problem is to include multiple concrete interface descriptions in the appliance specification (as in the INCITS/V2 standard [Zimmermann 2002] and ICrafter [Ponnekanti et al. 2001]). We chose not to take this approach because it does not support future platforms that cannot be anticipated at design time. This approach also makes it difficult to support many of the expected benefits of automatically generating interfaces, such as adaptation and personalization.

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

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

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

4. PRELIMINARY USER STUDIES

These principles guide the design of our language, but do not suggest what information should be included or what level of detail is needed to automatically generate high-quality interfaces. In order to determine what content should be included in a specification, we hand-designed several remote control interfaces for existing appliances. Then user studies were conducted to compare the hand-designed interfaces to the manufacturers' interfaces (described in more detail in Nichols [2003]). This approach allowed us to concentrate on the functional

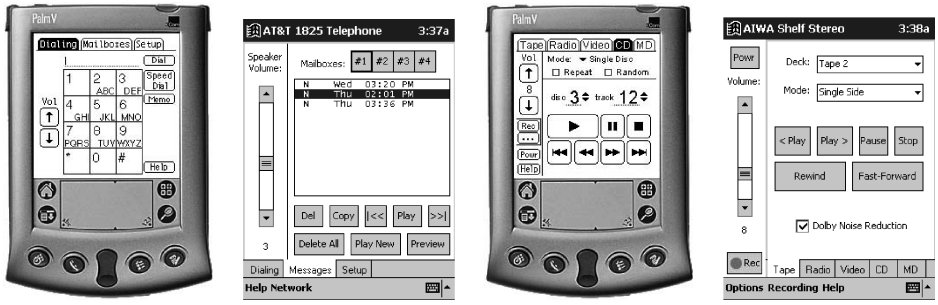

Fig. 1. Hand-designed interfaces for the phone (a) (b) and stereo (c) (d) on the Palm and PocketPC. The Palm interfaces are paper prototypes, whereas the PocketPC interfaces were actually implemented in Microsoft's embedded Visual Basic.

information that should be included as content in the specification language. It also showed that a PUC device 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. The first author owns the Aiwa shelf stereo that we used, and the AT&T telephone was the standard unit installed in many offices at Carnegie Mellon University at the time. Aiwa-brand stereos seem to be particularly common (at least among our subject population) because ten of our twenty-five subjects coincidentally owned Aiwa systems.

We created our hand-designed interfaces in two phases, initially on paper for the Palm platform and later as Visual Basic implementations on a Microsoft PocketPC (see Figure 1). Each interface supported the complete set of appliance functions. At each phase, we iteratively improved the interfaces with heuristic analyses and performed a user study. The user study in each phase was dual-purpose: to compare our hand-designed interfaces with the interfaces on the actual appliances and to find problems in the hand-designed interfaces.

The comparison study in both phases showed that our hand-designed interfaces were much better than the manufacturer's interfaces on the actual appliances [Nichols 2003]. In both studies, users were asked to perform a variety of simple and complex tasks. Some simple tasks were dialing the phone or changing the volume on the stereo, whereas some complex tasks were programming a list of tracks into the stereo's CD player or copying a message between two of the four mailboxes on the telephone's built-in answering machine. We found that for both hand-designed interfaces, Palm paper prototypes and PocketPC implementations, users completed tasks in one-half the time and with one-half the errors as compared to the actual appliances [Nichols 2003].

The large differences in this study can be attributed to problems with the appliance interfaces. Most of the problems users had with the built-in appliance interfaces could be traced to poor button labels and inadequate interface

feedback. Both appliances had buttons with two functions, one when the button was pressed and released and one when the button was pressed and held. Our subjects rarely discovered the press-and-hold function. The stereo also had buttons that changed function with the appliance's mode.

4.1 Interface Analysis

Once we were confident that our interfaces were usable, we analyzed them to understand what functional information about the appliance was needed for designing the interfaces. This included questions such as "why are these elements grouped together?" or "why are these widgets never shown at the same time?" These are questions that might suggest what information should be contained in the specification language.

As we intuitively expected, grouping information was very important for our hand-designed interfaces. We noted that grouping information could generally be specified as a tree, and that the same tree could be used for interfaces of many different physical sizes. User interfaces designed for small screens would need every branch in the tree, whereas large-screen interfaces might display some deeper branches together on the same screen.

We also found that grouping is influenced by modes. For example, the Aiwa shelf stereo has a mode that determines which of its components is playing audio. Only one component can play at a time. In the stereo interfaces shown in Figure 1(c) and (d) you will note that a tabbed interface is used to overlap the controls for the CD player, tape player, etc. Other controls that are independent of mode, such as volume, are available in the sidebar. Unlike regular grouping information, information about modes gives explicit ideas about how the user interface should be structured. If two sets of controls cannot be available at the same time because of a mode, they should probably be placed on overlapping panels. We designed dependency equations to describe appliance mode information in our language.

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

Each of our interfaces used different labels for some identical functions. For example, the Palm stereo interface (see Figure 1(c)) used the label "Vol" to refer to volume, whereas the PocketPC stereo interface (see Figure 1(d)) used "Volume." We expected that this problem would be even worse for much smaller devices, such as mobile phones or wristwatches. Thus we felt it would be important for our specification language to include multiple labels among which an interface generator could choose when designing its layouts.

Finally, we found that all of our interfaces used some "conventional" designs that would be difficult to specify in any language. At least one example of a

conventional design can be found in each of the panes in Figure 1: (a) shows a telephone keypad layout, (b) uses standard icons for previous track and next track, (c) shows the standard layouts and icons for play buttons on a CD player, and (d) uses the standard red circle icon for record. We have developed a solution for addressing this problem called Smart Templates [Nichols et al. 2004], which will be discussed later in Section 6.

5. SPECIFICATION LANGUAGE

The design of the specification language has been iterated upon for more than six years. Although new features, such as complex data structure support and content flow information, have been added since the initial version, the basic elements of the language have remained the same.

This design of the language is described here through an example specification for a basic VCR appliance specification. This VCR has five functions that users can manipulate: power, the common media controls including record and eject, channel, TV/VCR, and a list of timed recordings that will take place in the future. There are also two status indicators for determining whether a tape is in the VCR and whether this tape is recordable. The VCR has only one physical input, a standard television antenna, and two physical outputs: an antenna passthrough and the standard three wire yellow/red/white plugs for composite video and stereo audio. All of these features can be described in the specification language, as will be shown shortly. The full specification for the simple VCR can be found in the online appendix and in Appendix A of the first authors' dissertation [Nichols 2006]; snippets of the simple VCR's specification will be shown as each of the language's features is described.

This section focuses on the conceptual aspects of the language with limited discussion of syntax. Readers interested in authoring specifications should see the complete language reference.[1]

5.1 Functional Language Elements

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

—The functions of an appliance are represented by either state variables or stateless commands. Commands and states are collectively called *appliance objects*.

—Each state variable has type information which describes the values that a state variable may have and how they can be manipulated by the interface. The type information helps the interface generator decide how a variable should be represented in the final user interface.

[1]This is also included in the online appendix accessible through the ACM Digital Library or can be downloaded from the PUC Web site at:
`http://www.pebbles.hcii.cmu.edu/puc/specification.html`.

—Label information is also needed in the specification so that users can understand the functions of the appliance. The specification language allows multiple values to be specified for each label, so that, for example, strings of multiple lengths can be provided for use in interfaces for screens of different sizes and pronunciation information can be provided for a speech interface.

—The structures of the hand-designed interfaces were often based upon dependency information. For example, suppose that an interface was being created for a shelf stereo system with a tape and CD player. When the power is off, a screen with only a power button widget would be shown, because none of the other objects would be enabled. When the power is on, a screen is shown with many widgets, because most of the objects are active when the power is on. We might also expect this interface to have a panel whose widgets change based upon whether the tape or CD player is active.

—The final representation of any interface can be described using a tree format. It is not reasonable to include the tree representation of a particular interface in the specification of an appliance, however, because the tree may differ for different form factors. For example, the tree will be very deeply branched on a small screen WAP cellular phone interface, whereas the tree will be broader for a desktop PC interface. The specification language defines a group tree that is deeply branched. It is expected that this information could be used for small screen and large screen interfaces alike, because presumably some of the branches could be collapsed in a large interface.

—Complex data types can also be specified, such as lists and unions. The specification of complex data reuses the tree structure from portions of the appliance specification to improve ease of authoring.

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

Each of these items is described in detail next.

5.1.1 *Appliance Objects.* Three types of appliance objects are supported in the specification language.

—*States*. States are variables that represent data stored within the appliance. Examples might be the radio station on a stereo, the number of rings until an answering machine picks up, the time that an alarm is set for, or the channel on a VCR (see Figure 2(a)). Each variable has a type, and the UI generator assumes that the value of a state may be changed to any value within that type, at any time that the state is enabled. It is possible for the value of state variables to be undefined.

—*Commands*. Commands represent any function of an appliance that cannot be described by variables. They may be used in situations where invoking the command causes an unknown change to a known state variable (such as the

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

(a)

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

(b)

Fig. 2. Examples of (a) a state variable representing the current channel tuned by the VCR and (b) a command for ejecting the tape currently in the VCR.

"seek" function on a radio), or in situations where the state variable is not known (due to manufacturer choice or other reason, for instance, the dialing buttons on a standard phone would all be commands). In the VCR specification, the Eject function is represented by a command (see Figure 2(b)). Commands in the PUC specification language cannot have explicit parameters as they may in other languages such as UPnP. Where parameters are needed, the author can use state variables and specify dependencies that require the user to specify these variables before the command can be invoked. We could have allowed explicit parameters, but this feature would have overlapped with state variables, increased the complexity of the language, and would break our "only one way to specify" principle.

—*Explanations*. Explanations are static labels that are important enough to explicitly appear in the user interface, but are not the label of an 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.

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

5.1.2 *Type Information.* Each state variable must be specified with a type so that the interface generator can understand how it may be manipulated. For example, the Channel state in Figure 2(a) has an integer type. We define seven primitive types that may be associated with a state variable.

—binary
—boolean
—enumerated
—fixed point
—floating point
—integer
—string

Many of these types have parameters that can be used to restrict the values of the state variable further. For example, the integer type can be specified with minimum, maximum, and increment parameters (see Figure 2(a)).

It is important to note that complex types often seen in programming languages, such as records, lists, and unions, are not allowed to be specified as the type of a state variable. Complex type structures are created using the group tree, as discussed later in Section 5.1.6.

5.1.3 *Label Information.* The interface generator must also have information about how to label appliance objects. Providing this information is difficult because different form factors and interface modalities require different kinds of label information. An interface for a mobile Web-enabled phone will probably require smaller labels than an interface for a PocketPC with a larger screen. A speech interface may also need phonetic mappings and audio recordings of each label for text-to-speech output. We have chosen to provide this information with a generic structure called a *label dictionary*.

Each dictionary contains a set of labels, most of which are plain text. The dictionary may also contain phonetic representations using the ARPAbet (the phoneme set used by CMUDICT [CMU 1998]) and text-to-speech labels that may contain text using SABLE mark-up tags [Sproat 1998] and a URL to an audio recording of the text. The assumption underlying the label dictionary is that every label contained within, whether it is phonetic information or plain text, will have approximately the same meaning to the user. Thus the interface generator can use any label within a label dictionary interchangeably. For example, this allows a graphical interface generator to use a longer, more precise label if there is sufficient screen space, but still have a reasonable label to use if space is tight. Figure 3 shows the label dictionary for the Play Controls group of the VCR, which has two textual labels and a text-to-speech label.

5.1.4 *Dependency Information.* The PUC has a two-way communication feature that allows it to know when a particular state variable or command is active. This can make interfaces easier to use because the controls representing elements that are inactive can be disabled. The specification contains formulas that specify when a state or command will be disabled depending on the values

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

Fig. 3. The label dictionary for the playback controls group of the VCR. This dictionary contains two textual labels and some text-to-speech information.

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

Fig. 4. An example of a common type of dependency equation specifying that a variable or command is not available if the appliance's power is turned off.

of other state variables. These formulas can be processed by the PUC to determine whether a control should be enabled when the appliance state changes. Five kinds of dependencies can be specified: Equals, GreaterThan, LessThan, Defined, and Undefined. Each of these specifies a state that is depended upon and a value or another state variable to compare with. These dependencies can be composed into Boolean formulas using AND, OR, and NOT. Figure 4 shows an example dependency formula.

We have discovered that dependency information can also be useful for structuring graphical interfaces and for interpreting ambiguous or abbreviated phrases uttered to a speech recognizer. For example, dependency information can help the speech interfaces interpret phrases by eliminating all possibilities that are not currently available. The use of these formulas for interface generation is discussed elsewhere [Nichols et al. 2002].

5.1.5 *Group Tree.* Interfaces are always more intuitive when similar elements are grouped close together and different elements are kept far apart. Without grouping information, the start time for a timed recording might be placed next to a real-time control for the current channel, creating an unusable interface. This is avoided by explicitly including grouping information in the specification using a hierarchical group tree.

The group tree is an n-ary tree that has a state variable or command at every leaf node (see Figure 5). State variables and commands may be present at any level in the tree. Each branching node is a "group," and each group may contain any number of state variables, commands, and other groups. Designers are encouraged to make the group tree as deep as possible, in order to help space-constrained interface generators. These generators can use the extra detail in the group tree to decide how to split a small number of controls across two screens. Interface generators for larger screens can ignore the depth in the group tree and put all of the leaf controls on one panel.

5.1.6 *Complex Data Structures.* The PUC specification language uses the group tree to specify complex type structures often seen in programming

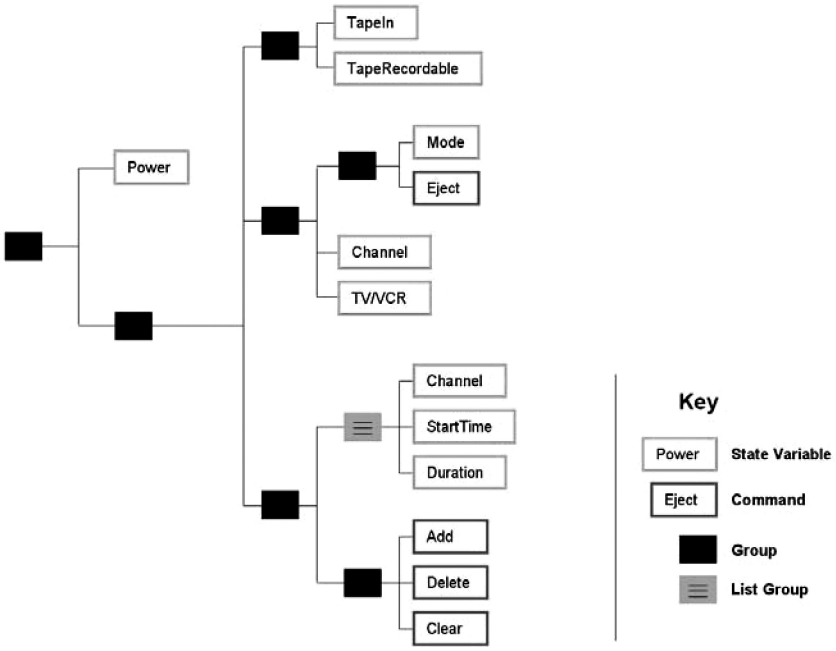

Fig. 5. The group tree for the sample VCR specification.

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

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

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

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

Fig. 6. An example of a list group used in the VCR specification to describe the list of 0 to 8 timed recordings that may be specified by the user.

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

Figure 6 shows an example of the list group element added for specifying lists. Specifying a list group is similar to specifying an array of records in a

programming language, and multiple list groups can be nested to create multi-dimensional lists. Each list group has an implicit length state variable (named "Length") that always contains the current length of the list. The specification may define bounds on the length of the list in order to help the interface generator create a better rendering. An exact fixed size or a minimum and/or maximum size may be specified. List groups also maintain an implicit structure to keep track of one or more selected elements of the list; more details about this can be found in the online appendix.

We also developed a special dependency operator for lists, called apply-over. This element applies the dependencies it contains over items in a list and, depending on the value the true-if property, returns true if the dependencies were satisfied for either all of the items or at least one of the items. The set of items that the dependencies are applied over is based on the items property, which may be set to *all* or *selected*.

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

5.2 Content Flow Language Elements

Information about content flow can be useful to describe the relationships between appliances that have been connected together in multi-appliance systems. The specification language allows authors to describe the input and output ports that an appliance possesses and the internal content flows that use those ports. For example, the content flow elements allow us to specify that our VCR may pass content from its antenna input directly to its antenna output, and if a tape is playing then the content from this tape will appear on a particular channel of the antenna output. Wiring information for a system of appliances can be combined with the port and content flow information from each appliance to build a model of content flow through the entire system. This content flow model is very useful for generating interfaces that aggregate functions from multiple appliances, such as in our Huddle system [Nichols et al. 2006b].

Specifying content flow information is optional, and the PUC is able to generate user interfaces even when this information is not specified.

5.2.1 *Ports.* The input and output ports of an appliance define that appliance's relationship with the outside world. In order to match the user's intuitive understanding of ports, specification authors are encouraged to create a port for each of the physical plugs that exist on the outside of an appliance. Future tools could then use this information to help users correctly wire their systems. The specification language also supports port groups, which allow the author to give a single name to collection of related ports, typically ports that carry a piece of a larger content stream. For example, in Figure 7 the "Output" port group is a combination of the physical "Video" port with the physical "Left" and "Right" audio ports. The port group convenience makes it easy for specification

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

Fig. 7. The ports section of the example VCR specification.

authors to define that a video stream uses the "Output" port without needing to specify each of its constituent ports.

5.2.2 *Internal Flows: Sources, Sinks, and Passthroughs.* Our language allows three different types of internal content flows to be specified.

—*Sources* represent content that originates within the appliance, such as from a DVD player playing a DVD or a VCR playing a videotape. Display devices that have internal tuners, such as televisions receiving broadcast signals through antennas, are not defined as sources, however, because the content does not originate inside of the tuning device.

—*Sinks* represent locations where content may either be displayed to the user or stored for later retrieval. For example, the television screen, receiver speakers, and VCR tape (for recording) may all be sinks for content.

—*Passthroughs* represent an appliance's ability to take in some content as an input and redirect it through one or more of its outputs. For example, our VCR has the capability of taking a television broadcast signal as an input and making it available as an output for other appliances.

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

Each of the internal flow types is specified with three basic pieces of information: a dependency formula defining when the flow is active, a description of

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

Fig. 8. The description of the video tape source content flow from the example VCR specification. Note that dependencies from both of the content groups that contain the source flow are ANDed with the source's own dependencies.

the ports associated with the flow, and a list of state variables, commands, and groups that can be used to modify the behavior of the flow. In the specification language, sinks are divided into two subtypes, recorder and renderer, that describe what the appliance does with the content it receives. Figure 8 shows an example of a source content flow from the example VCR specification.

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

Channels are an important concept in content flow specifications. When a passthrough or sink receives a multichannel input, a channel variable may be specified from the appliance that specifies the particular channel being tuned.

The language can also specify that one channel of a multichannel stream is being replaced by the appliance, which is used by the example VCR specification to describe that the output of the tape source can appear on channel 3 (see Figure 8).

6. SMART TEMPLATES

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

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

A new Smart Template is defined by giving the template a name and defining a set of specification restrictions for the template. A specification author instantiates a template by adding an is-a attribute to a group, variable, or command with the name of the template and then conforming to the template's restrictions within that section of the specification (see Figure 9). When the interface generator encounters a section of a specification referencing a template that it knows about, it can invoke special code to appropriately render the template. If the generator encounters a template that it does not know about, it will use its normal generation rules to render the template's contents. This is possible because every Smart Template is defined using the normal primitive elements of the specification language. For example, Figure 10(a) shows an instance of

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

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

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

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

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

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

(a)

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

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

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

(b)

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

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

(c)

Fig. 9. Three specification snippets showing instantiations different Smart Templates. (a) An instantiation of the media-controls template for the play controls on Windows Media Player. Renderings of this template are shown in Figure 10 (a) and (b). (b) The instantiation of the time-duration template for the counter function on the Sony DV Camcorder. (c) The instantiation of the time-duration template for the song length function on Windows Media Player.

the media-controls Smart Template in Figure 9(a) rendered by a generator with no knowledge of that template and Figure 10(b) shows the same instance rendered by generators on several different platforms that did know about the template.

The restrictions on the specification allow Smart Templates to be parameterized, which allows them to cover both the common and unique functions of an appliance. Parameters are specified in terms of the primitive elements of the specification language and consist of a list of the state variables and commands that the template may contain along with definitions of the names, types, values, and other properties that these elements must have. Some of the elements may be optional to support functions that would not be used in all instantiations of a template. For example, two representations of play controls are allowed by the media-controls template: a single state with an enumerated type or a set of commands. If a single state is used, then each item of the enumeration must be labeled. Some labels must be used, such as Play and Stop, and others are optional, such as Record. If multiple commands are used, then each command must represent a function such as Play and Stop. Some functions must be represented by a command and others are optional. This template also allows

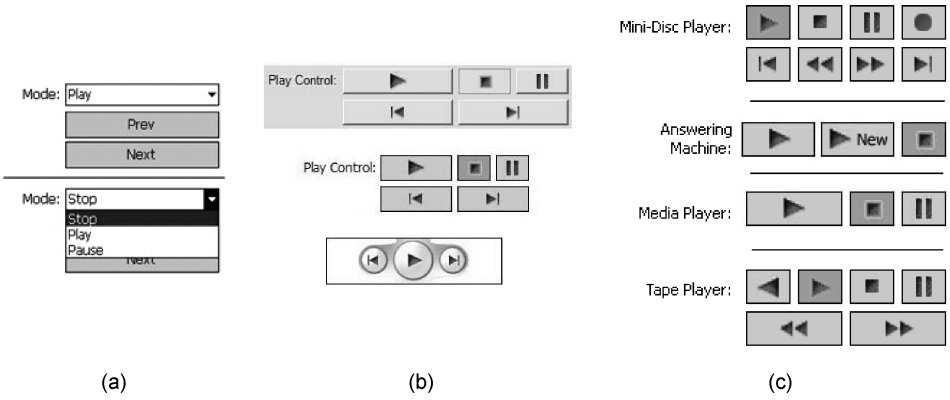

Fig. 10. Renderings of the media-controls Smart Template on different platforms and for different appliances. (a) Media controls rendered with Smart Templates disabled for the Windows Media Player specification shown in Figure 9(a) on the PocketPC platform. (b) Media controls rendered for the same interface with Smart Templates enabled using generators for (top to bottom) the desktop, the PocketPC, and the Microsoft 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 at the time. This interface overloads pause and stop onto the play button. (c) Different configurations of media playback controls automatically generated on the PocketPC for several different appliances.

three commands for functions that are commonly included in the same group as the play controls, including the previous and next track functions for CD and MP3 players, and the "play new" function for answering machines. Allowing many combinations of states and commands in a template definition allows a single Smart Template to be applied across multiple kinds of appliances (see Figure 10(c)).

The challenge for the creator of a Smart Template is to find the different combinations of states and commands that an appliance implementer is likely to use. This makes it easier for appliance specification writers to use the templates, because there is no need to modify the appliance's internal data representation in order to interface with the controller infrastructure. For example, some appliances may not be able to export their playback state, and thus would want to use the option of specifying each playback function with a command. Another example is for the time-duration template. Windows Media Player makes the duration of a song available as a single integer while our Sony DV Camcorder makes the playback counter available as a string (see Figure 9(b) and (c) for specifications). Both of these representations can be accommodated by the time-duration Smart Template, which allows the PUC to be implemented more cleanly with these appliances because no translation is needed from the underlying implementation to the requirements of the PUC. Note that the regular specification language is used to specify how the Smart Template is represented. No new techniques need to be learned by specification authors to use and customize Smart Templates.

Each interface generator can implement special generation rules for each template. This allows each template to be rendered appropriately for its

controller device using bits of hand-designed interfaces specifically created for the template and the controller device. Sometimes these hand-designed interfaces include platform-specific controls that are consistent with other user interfaces on the same device. In the case of our time-duration Smart Template implementation, each platform has a different standard control for manipulating time that our interface generators use. Unfortunately, none of our platforms has built-in controls for media playback, so our media-controls Smart Template uses renderings that were hand-designed to be appropriate for each of the different platforms. For example, the Smartphone media-controls implementation mimics the interface used by the Microsoft Smartphone version of Windows Media Player, and thus is consistent with another application on that device (see Figure 10(b)).

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

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

Smart Templates are related to "comets" [Calvary et al. 2004], a widget-level technology for supporting user interface plasticity. Comets seem to focus primarily on supporting runtime adaptation within a user interface, such as changing a widget's appearance when it is resized. The focus of our Smart Templates is not runtime adaptivity, but compile-time (or authoring-time) adaptivity. The wide range of different data formats that are supported by most Smart Templates allow the specification author to create a template using a combination of variables and commands that are best suited for the particular appliance being described.

6.1 Smart Template Library

The PUC research team has implemented 15 different Smart Templates. These templates collectively illustrate all of the features discussed in previous sections.[2]

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

—The PUC supports state variables with a raw binary data type. These variables could contain images, sounds, or other data that cannot be easily communicated through the PUC's text-based communication protocol. Smart Templates were chosen to handle this binary information for several reasons. First, it did not seem appropriate for all interface generators to handle all kinds of binary data. Smart Templates were attractive then because they are optional by design, although in this case the lack of a Smart Template for handling a particular binary data type means that the data cannot be presented in the interface. Second, the controls needed for displaying any binary data would need to be custom built for the particular platform, which Smart Templates are already capable of adding to the generated interface. Finally, binary-typed data requires extra communication to negotiate the particular subtypes of data (such as image formats) that a particular platform supports. This extra communication is likely to differ based on the type of binary data being managed, requiring different implementations for each type.

—List operations can also be handled by Smart Templates, allowing for special integration with the controls for handling list data that would not otherwise be possible. A challenge the PUC faced for handling lists was how to represent list operations. There are several different kinds of list operations, such as add item, delete, move, insert, etc., with so many different variations (e.g., add-first, add-last, add-after, add-before, and so on) that did not seem practical to include as first-class elements in the specification language. Ultimately, we decided that most list operations could be specified as separate commands outside of the list that they operated on. However, we found that user interfaces were easier to generate when the interface generator could identify the commands that operated upon the list. Furthermore, commands that added items to a list could not automatically display a dialog box to enable editing of the newly added item. The solution we chose was to create the list-commands Smart Template, which is a template that may contain multiple other templates. These constituent templates include list-add, list-remove, list-clear, list-move-before, and several others. The list templates allow all operations to be grouped appropriately next to the list control, add operations to perform the correct dialog box opening behavior, and for "move" commands to be handled directly inside the list control rather than being

[2]A full description of all Smart Templates is available on the Web at `http://www.pebbles.hcii.cmu .edu/puc/highlevel-types.html`.

displayed as separate commands away from the list. It is important to note that while the list templates allow for better list user interfaces, it is possible for PUC interface generators to produce an adequate list interface with no knowledge of these templates. Another result of this design decision is that it may limit specification designers from describing many direct manipulation operations. We believe it may be possible to support certain unrestricted direct manipulation operations, such as arbitrary moves, through the use of a Smart Template, though this has not been implemented in the current PUC system.

7. EVALUATION OF THE PUC

There are several important questions to ask about the PUC specification language as a whole.

—Is the language sufficiently complete to specify the functionality of most appliances?
—Is the language easy to learn and use?
—Can high-quality interfaces be generated from the language?

7.1 Completeness

Members of the PUC research group, including the first author, several undergrads, a masters student, and two staff members, have used the specification language to author specifications for 33 different appliances (see Table I). We have tried to cover a large range of appliance types and to write specifications for several highly complex appliances, including a high-end Mitsubishi DVCR, a Samsung DVD-VCR combo player, all-in-one printers from HP and Canon, and the navigation system from a GMC vehicle. Table II shows some statistics for the specifications that have been written so far. The table shows that PUC specifications on average are quite complex, particularly the GM navigation system specification, which is nearly twice as complex as any other specification. All of these specifications cover *all* of the functions of their appliance, giving us confidence that the language is capable of representing both the most common and most obscure functions of any appliance.

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

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

The main difficulty in the language design came from supporting complex data structures, particularly the lists that are found on many appliances. The design of these elements of the language was driven primarily by the GM

Table I. Complete List of Appliance Specifications Authored by the PUC Research Team

Home Entertainment Appliances:	*Office Appliances:*
Audiophase Shelf Stereo	AT&T 1825 Phone/Answering Machine
DirecTV D10-300 Receiver	Canon PIXMA 750 All-In-One Printer
Gefen 2:1 HDMI Switchbox	Epson PowerLite 770 Projector
InFocus 61MD10 Television	HP Photosmart 2610 All-In-One Printer
JVC 3-Disc DVD Player	Complex Copier
Mitsubishi HD-H2000U DVCR	Simple Copier
Panasonic PV-V4525S VCR	*Desktop Applications:*
Philips DVD Player DVDP642	Laptop Video Output Controls
Samsung DVD-V1000 DVD-VCR	Microsoft PowerPoint
Sony A/V Receiver	Microsoft Windows Media Player 9
Sony Camcorder	PUC Photo Browser
Lighting Controls:	PUC To-Do List
Intel UPnP Light	Task Manager
Lutron RadioRA Lighting	*Alarm Clocks:*
X10 Lighting	Equity Industries 31006 Alarm Clock
GMC 2003 Yukon Denali Systems:	Timex T150G Weather Alarm Clock
Driver Information Console	*Other:*
Climate Control System	Axis UPnP Pan-Tilt-Zoom Camera
Navigation System	Simulated Elevator

Table II. Maximum and Average Counts of Various Aspects of the PUC Specifications Written by Our Group

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

*Not including GM navigation system specification.

navigation system specification, which contains many lists of complex data, such as destinations. Several iterations on this specification led to our current language design, which combined grouping with list and union features and is capable of representing all forms of structured data. The design has since been used without modification on many other specifications, including those for many of the home entertainment and office appliances.

7.2 Learnability and Ease of Use

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

The formal study was conducted with three subjects who learned the language from reading a tutorial document (see the online appendix or the first author's dissertation [Nichols 2006]) and doing exercises on their own for approximately 1.5 hours. Subjects were then asked to write a specification for a low-end Panasonic VCR, which took on average 6 hours to complete. The focus

of this study was on the consistency of the resulting specifications and not learnability per se, so the details of the study preparation and its specific results are discussed elsewhere [Nichols et al. 2006a]. The subjects were able to learn the language sufficiently in the short 1.5 hour period to write valid specifications for the VCR. This suggests that the language is very easy to learn.

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

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

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

7.3 Quality of the Generated Interfaces

A user study was conducted to examine the usability of user interfaces generated from the PUC specification language. This study compared the generated interfaces to existing human-designed interfaces for the same functionality, with the hypothesis that interface quality is no longer a limiting factor for automatically generated interfaces. Many more details of this study are presented elsewhere [Nichols et al. 2007].

Our user study compared interfaces for two different off-the-shelf all-in-one printer appliances: a Hewlett-Packard (HP) Photosmart 2610 with a high-quality interface including a color LCD, and a Canon PIXMA MP780 with a few more features and an interface that turned out to be harder to learn than the HP. These two represented the top-of-the-line consumer models from these manufacturers and the most complex all-in-printers available for home use at the time of their purchase (mid-2006). All-in-one printers were chosen as the appliances in this study because they were challenging to specify, resulted in complicated generated user interfaces, and a convincing simulation of remote controlling the appliances was easy to implement. The existing manufacturers'

(a) HP Printer (b) Canon Printer

Fig. 11. One screen from each of the PocketPC interfaces generated by the Personal Universal Controller (PUC) for the two all-in-one printers.

interfaces from both printers were used for the comparisons conducted in the study. The PUC-generated interfaces were presented on a Microsoft PocketPC device (see Figure 11). In total there were four user interfaces tested in the study: the existing user interfaces for the two printers, and the automatically generated PocketPC user interfaces for the same printers.

PUC specifications of both all-in-one printers were needed in order for the PUC to generate interfaces. Different writers from the PUC development team were used for the two specifications to create a realistic scenario where the specifications were written separately by different manufacturers. The specifications were written using an approach that we would expect actual specification writers to take. Writers were generally faithful to the design of the actual appliances, but also took advantage of the features of the PUC specification language, such as including extra labels for functions with further detail where necessary. Both specifications included *all* of the features of their appliances. The initial specifications were tested with the interface generators to ensure correctness and went through several iterations before they were deemed of high enough quality to be used for the study. Note that this testing is similar to debugging a program or iteratively testing a user interface and is necessary to ensure that no functions are forgotten, understandable labels are used, etc.

The procedure used for this study was similar to that used in the preliminary user studies described in Section 4. Eight tasks were designed for subjects to perform during the study, and were chosen to be realistic for an all-in-one printer, covering a wide range of difficulties, and as independent from each other as possible (so success or failure on one task would not affect subsequent tasks). Subjects performed the eight tasks twice, once for each all-in-one printer. Subjects either used only the existing interfaces on the appliances, or only the PUC-generated PocketPC interfaces. For each task, we recorded the time necessary to complete the task and whether the subject was able to complete

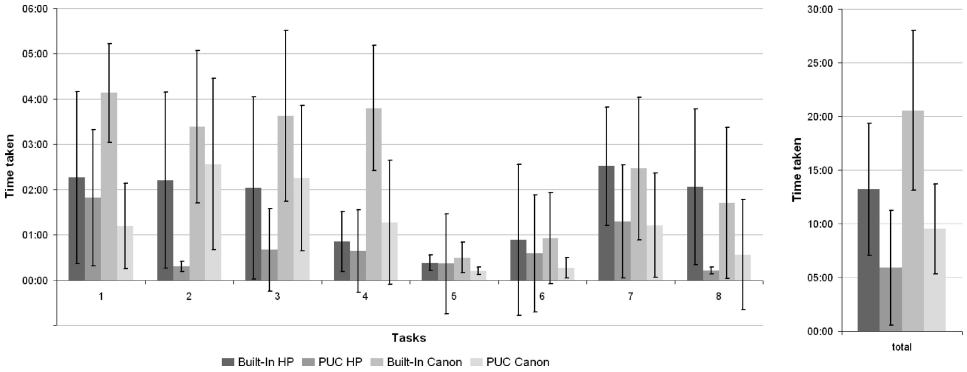

Fig. 12. The results of the study, showing performance time with the existing appliance interfaces compared with the PUC-generated user interfaces for each of the 8 tasks and the total time for all tasks.

Table III. Average Completion Time and Total Failure Data for the First Block of Tasks

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

the task. Thus, we used a between-subjects design to compare performance between the existing appliance interfaces and the PUC-generated interfaces.

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

Figure 12 shows the average completion time for each of the tasks on each appliance, comparing the existing interfaces with the automatically generated interfaces. Table III shows the data in more detail. To evaluate the performance time data, we performed a mixed model analysis using log(time) as the dependent variable. Many more details of this analysis are available elsewhere [Nichols 2007]. We found significant differences in performance time and failures between the existing interfaces and the automatically generated interfaces. Overall, we found that users of the automatically generated interfaces were twice as fast and four times more successful than users of the existing interfaces for the set of eight tasks. This suggests that our specification language is capable of describing a user interface sufficiently for an interface generator to create a high-quality user interface.

8. ANALYZING OUR DESIGN PROCESS

The steps in our design process were as follows.

(1) *Create two interfaces for representative applications in our domain.* It was necessary to create interfaces ourselves because there were few existing remote control user interfaces for appliances that were designed for handheld devices. Those that existed did not have all of the features that we hoped to support with the PUC system. In other domains, it might be possible to start with existing interfaces instead of creating new ones.

(2) *Analyze these interfaces and develop a set of principles, and an initial language that adequately describes the basic features of these interfaces.* Our initial language design did not include every feature of the preliminary interfaces, such as the telephone number pad or the list of messages on the AT&T phone. We felt it was important to get a basic framework in place rather than support all the complexity of the interfaces in our initial language design.

(3) *Iterate on the design with the initial interfaces and new example interfaces in the same domain.* We iterated on our design several times with our hand-designed interfaces before we moved on to new appliances. As we became comfortable with our basic design, we gradually added in more complex features, such as Smart Templates and support for complex data structures. Over time we found increasingly complex appliances to specify, some of which inspired additional modifications to the language. Simultaneously with our work on the language, we were also building our interface generation software. As we better understood the needs of our interface generator, we were able to make some modifications to the language to better support the generation of user interfaces. Periodically, new people joined our research team and learned the language. Their experiences helped us modify the language to make it more usable and learnable.

Overall, we found our design process to be very effective. Building example interfaces allowed us to get deep into the problems involved with creating the types of interfaces that we hoped to automatically generate, and helped us understand the information that was necessary in order to create them. Focusing on the basic design features of the language first allowed us to build a strong base for our language, on top of which we could layer more complex features. Periodically introducing the language to new users increased the usability of the language. Working in parallel with development of the user interface generation software ensured that our language supported our overall goal of generating high-quality user interfaces. We strongly recommend this approach to the designers of future UIDLs.

In retrospect, our process might have worked even better if we had done two things differently. First, it would have been helpful to push certain complex features into the language more quickly. For example, we held off on adding support for lists to the language for far too long. Lists were part of the initial interfaces that we created, and we knew from the beginning that a list feature would need to be a part of the language. Early in the process, we had developed

several design alternatives for how lists might be included, but we found none of them particularly satisfying and there was little evidence to support choosing one over another. As a result we focused on other aspects of the research and none of the alternatives was implemented. Finally, when the feature was required for specifying the key features of a new appliance, we found that the language needed substantial modification to support lists and other complex data structures. It is worth noting that it was easy to add *other* complex languages features later in the process, however. Smart Templates, for example, required little modification to the basics of the language and could have easily been added even later in the design process.

Second, we should have more quickly ramped up the complexity of the appliances for which we wrote specifications. Our most complicated specification was written towards the end of the project, and it led to the addition of several new language features. Had we written this specification earlier, then the design of our language might have matured much faster. This problem may be less of an issue in a different domain. A particular focus of the PUC project was to control real appliances, and thus much of our early work focused on appliances that we could actually control remotely. Although some of these appliances were complex, few of their complex features could be controlled by our system. The specifications we wrote were for the features we could control, and thus early on we were not always exploring the full functionality of our appliances.

9. ANALYZING OUR FINAL DESIGN

We start by considering which aspects of our language worked well and which did not. Then we discuss the overall design of our language in terms of issues that may be applicable to other UIDLs.

9.1 What Worked Well?

One of our design principles was to keep the language short and concise. Keeping the design simple benefited us in a number of ways.

—The language is easy to learn, as shown by our user studies.
—Specifications can be written relatively quickly, even for complicated appliances.
—We found that basic representations, such as trees and logical formulas, were the most concise means to represent certain information. As a result we were able to apply existing algorithms, such as schema matching algorithms and AI planners, with little modification.

We also found that it was not always necessary to build every needed feature into the language directly, but instead it was valuable to build in a workaround (e.g., Smart Templates) that we could fall back on for a few features that were otherwise difficult to describe. This kept our language from becoming unnecessarily complicated. For example, our language might have included a large set of list operations to describe how any particular list might be manipulated by the user. Instead we chose to represent these operations using Smart Templates, which required no change to our language design. Because our design of

Smart Templates made use of the basic elements of our language, interface generators could always fall back to the basic language if they did not understand the template. Of course, such a workaround could be designed in a number of ways. In some UIDLs, it might be appropriate to introduce small amounts of executable code into the description as a workaround.

9.2 What Did Not Work?

The "only one way to specify anything" guideline was helpful during design, but ultimately we found that it should not be thought of as a requirement. Usability may be better served in some cases by allowing multiple means of specifying certain features; each of the different options should be designed to be preferable in a particular situation. For example, in the current language design the union group is a more explicit means of creating mutually exclusive groups of functions. We found the concept of a union group to be more useful when thinking about a data structure than the somewhat abstract notion of groups with mutually exclusive dependencies.

Many of the other difficulties we encountered in the design of our language are shared with any large software or user interface project.

—We added several features to our language early on that turned out to be not very useful, such as explanations and a notion of the priority of an appliance object. These features remained in the system, however, and caused confusion for people learning the language. We probably should have been more aggressive about removing these relatively useless and confusing features. Instead we simply made a point of noting that the features were unused in our documentation.

—Versioning was an issue. As we modified the language, we attempted to maintain backward compatibility so that our old specifications could still be processed without modification. Twice we found that backward compatibility was not easily possible, however, and we were forced to rewrite the old specifications.

9.3 General Discussion

One of the key decisions to be made in any UIDL is to decide where the language falls along the continuum from an application's functions to a final user interface design. The functional representation used by the PUC avoids including presentation information and thus is much closer to the application's functions than a final interface design. The presentation models in some previous systems and an initial version of the INCITS/V2 specification were closer to the final interface, because they directly specify the type of control to be used for each function (e.g., button, text box, etc.) and in some cases describe concrete layout information. Our choice to stay closer to the application functions prevents the specification from biasing towards any one interface modality and was effective for interfaces in our domain.

It is not clear, however, in which other domains our representation would be useful. For example, our language cannot describe applications that feature

objects that are created and directly manipulated by the user, such as drawing applications, circuit design applications, etc. Modeling behaviors in direct manipulation applications seems much more complex than what can be represented in the PUC language. The PUC representation is also not object-oriented nor does it make use of a task model, both of which might be needed features in some domains.

A related design issue when creating a UIDL is determining the desired balance between work done by the specification author and the interface generation software. If the specification designer must do a lot of work to write a highly detailed specification, then a simple interface generator may be used because little analysis and extrapolation of the specification will be necessary to build a final user interface. Alternatively, if the specification designer only specifies the basic outline of the user interface, then a powerful interface generator will be needed to generate high-quality user interfaces. Our goal was to push as much work as possible into the interface generator, so that specifications could be written quickly and easily. We found it necessary to iterate on this balance over time as we explored what was possible for our interface generator.

Another important balance in the design of our language was between subjective and objective information. For example, we consider dependency information to be objective because it is determined by the actual operation of the appliance and, in most cases, would be described identically by different authors. The group tree and label information are subjective, however, because both require the specification author to interpret the functions of the appliance and make her own decisions. Subjective information in a specification is likely to vary, sometimes widely, between different authors. Our preference was to include objective information in the specification and this information was given a greater weight within our interface generators.

Although we do believe our language design is closer to the application functions than the final interface design, we made an explicit choice not to allow a specification designer to include any snippets of code within our specifications. For example, languages like XUL and MXML allow JavaScript code to be included along with the declarative specification of the user interface. Instead, we chose to use structures that would be easy to inspect and analyze, which is often much more difficult with raw code. In some ways this made more work for our interface generators, which had to analyze the structures instead of blindly executing code, but this also had many benefits. We were able to find patterns in the specification, as in the case of mutually exclusive dependency formulas, which allowed our interface generators to create better user interfaces than they might have otherwise. Additionally, we later found that existing algorithms could be applied to gain additional understanding about the specifications, which would not have been possible, or at least more difficult, if we had allowed snippets of code in some existing programming language to be included in our language.

We have shown that the PUC system can generate high-quality interfaces. An important question to ask then is what properties of our UIDL contributed to our ability to generate these high-quality interfaces? Which

properties contributed the most? These questions are difficult to answer because every piece of information in the specification has a purpose. Without the group tree, the interfaces would be disorganized. Without human-readable labels, every function would have a mysterious name that would be difficult to interpret. If we compare the interfaces generated by the PUC to the existing appliance interfaces, then some features of the language stand out more clearly. Many appliance interfaces do not clearly indicate which functions are currently available, whereas the PUC is able to show this because of dependency information. Dependency information is also used to make some layout decisions, such as placing more commonly available functions on side panels so that they may be more universally accessed. Interfaces for lists and other complex data structures are also often easier to understand and interact with when displayed in a PUC interface. This is partly due to the larger screen that PUC devices commonly have, but it is also due to the use of more common and consistent interactions for list operations. Existing appliance user interfaces use a variety of different, often confusing, interactions to implement list operations, whereas all PUC interfaces on the same device use the same interactions.

A particularly noticeable improvement in interface quality occurred when we added Smart Templates to the language. Prior to this addition, the interface generator was able to create reasonably organized and understandable interfaces, but the lack of standard conventions for some functions was noticeable, such as missing the standard icons and arrangements of the play, stop, and pause buttons in media player interfaces. Smart Templates do not affect the structure or overall layout of the user interface, but the detail that they add seems to have a large impact on usability.

10. CONCLUSION

We have discussed the design of a UIDL for describing appliance user interfaces, and the design process that resulted in this design. We have used this language to automatically generate high-quality remote-control interfaces that are easier to use than many existing appliance interfaces [Nichols et al. 2007]. We have also used the language as a basis to explore the possibility of personalizing interfaces, such as by ensuring consistency with interfaces the user has previously seen [Nichols et al. 2006a], and combining interfaces of appliances that are used together [Nichols et al. 2006b].

Going forward, we hope that the lessons we have learned in designing the PUC UIDL will be instructive to the designers of future UIDLs. We hope to see UIDLs become a pervasive feature of future UI technologies, which will allow personalization features, like those we developed for the PUC, to be applied in a wider variety of domains. There is some hope that UIDLs will become more common, especially with the emergence of commercial UIDLs such as XAML, XUL, and MXML. These languages are still very close to specifying concrete interface designs, however, and there is further research needed to design more abstract UIDLs that can gain a foothold in the commercial space.

ACKNOWLEDGMENTS

We would like to thank the many students and staff members who worked on the Personal Universal Controller project over the years, especially H. H. Aung, D. Horng Chau, T. K. Harris, M. Higgins, J. Hughes, M. Khadpe, K. Litwack, M. Pignol, R. Rosenfeld, B. Rothrock, and R. Seenichamy.

REFERENCES

ABRAMS, M., PHANOURIOU, C., BATONGBACAL, A. L., WILLIAMS, S. M., AND SHUSTER, J. E. 1999. UIML: An appliance-independent XML user interface language. In *Proceedings of the 8th International World Wide Web Conference*. http://www8.org/ and http://www.uiml.org/.

ABRAMS, M., SABOO, P., HELMS, J., SIMS, J., AND THOMAS, A. 2007. Using UIML to automate generation of usability prototypes and tactical software. In *Proceedings of the Human Systems Integration Symposium*.

ALI, M. F., PEREZ-QUINONES, M. A., ABRAMS, M., AND SHELL, E. 2002. Building multi-platform user interfaces with UIML. In *Proceedings of the Computer-Aided Design of User Interfaces Conference.* 255–266.

BOJANIC, P. 2006. The joy of XUL. http://developer.mozilla.org/en/docs/The_Joy_of_XUL.

CALVARY, G., COUTAZ, J., DAASSI, O., BALME, L., AND DEMEURE, A. 2004. Towards a new generation of widgets for supporting software plasticity: The "comet". In *Proceedings of the 11th International Workshop on Interactive Systems: Design, Specification and Verification*. 306–324.

CARD, S. K., MORAN, T. P., AND NEWELL, A. 1983. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ.

CMU. 1998. Carnegie Mellon pronuncing dictionary. http://www.speech.cs.cmu.edu/cgi-bin/ cmudict.

COENRAETS, C. 2004. An overview of MXML: The Flex markup language. http://www.adobe.com/devnet/flex/articles/paradigm.html.

FOLTZ, M. A. 2001. Ligature: Gesture-based configuration of the E21 intelligent environment. In *Proceedings of the MIT Student Oxygen Workshop*.

GAJOS, K., WOBBROCK, J. O., AND WELD, D. S. 2007. Automatically generating user interfaces adapted to users' motor and vision capabilities. In *Proceedings of the User Interface Software and Technology Conference (UIST)*. 231–240.

Incits/V2. 2003. Universal remote console specification. In *Alternate Interface Access Protocol*. Washington, D.C.

ISO. 1988. Information processing systems - Open systems interconnection - LOTOS - A formal description technique based on temporal ordering of observational behavior. In *ISO/IS 8807*, I. C. Secretariat Ed.

KIM, W. C. AND FOLEY, J. D. 1993. Providing high-level control and expert assistance in the user interface presentation design. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*. 430–437.

LIMBOURG, Q., VANDERDONCKT, J., MICHOTTE, B., BOUILLON, L., AND LOPEZ-JAQUERO, V. 2004. UsiXML: A language supporting multi-path development of user interfaces. In *Proceedings of the 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with 11th International Workshop on Design, Specification, and Verification of Interactive Systems (EHCI-DSVIS'04)*. 200–220.

Microsoft. 2006. XAML. http://windowssdk.msdn.microsoft.com/en-us/library/ms747122.aspx.

MORI, G., PATERNO, F., AND SANTORO, C. 2002. CTTE: Support for developing and analyzing task models for interaction system design. *IEEE Trans. Softw. Engin. 28*, 797–813.

MORI, G., PATERNO, F., AND SANTORO, C. 2004. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Trans. Softw. Engin. 30*, 1–14.

MYERS, B. A., HUDSON, S. E., AND PAUSCH, R. 2000. Past, present and future of user interface software tools. *ACM Trans. Comput. Hum. Interact. 7*, 3–28.

MYERS, B. A., NICHOLS, J., WOBBROCK, J. O., AND MILLER, R. C. 2004. Taking handheld devices to the next level. *IEEE Comput. 37*, 36–43.

NEWMAN, M. W., IZADI, S., EDWARDS, W. K., SEDIVY, J. Z., AND SMITH, T. F. 2002. User interfaces when and where they are needed: An infrastructure for recombinant computing. In *Proceedings of the User Interface Software and Technology Conference (UIST'02)*. 171–180.

NICHOLS, J. 2006. Automatically generating high-quality user interfaces for appliances. http://www.jeffreynichols.com/papers/dissertation-final-smaller.pdf.

NICHOLS, J., CHAU, D. H., AND MYERS, B. A. 2007. Demonstrating the viability of automatically generated interfaces. In *Proceedings of the Conference on Computer-Human Interaction (CHI)*. 1283–1292.

NICHOLS, J. AND MYERS, B. 2004. Report on the INCITS/V2 AIAP-URC standard. http://www.jeffreynichols.com/papers/cmu-puc-v2-report.pdf.

NICHOLS, J., MYERS, B. A., HIGGINS, M., HUGHES, J., HARRIS, T. K., ROSENFELD, R., AND PIGNOL, M. 2002. Generating remote control interfaces for complex appliances. In *Proceedings of the User Interface Software and Technology Conference (UIST)*. 161–170.

NICHOLS, J., MYERS, B. A., AND LITWACK, K. 2004. Improving automatic interface generation with smart templates. In *Proceedings of the Conference on Intelligent User Interfaces*. 286–288.

NICHOLS, J., MYERS, B. A., AND ROTHROCK, B. 2006a. UNIFORM: Automatically generating consistent remote control user interfaces. In *Proceedings of the Conference on Computer-Human Interaction (CHI'06)*. 611–620.

NICHOLS, J. AND MYERS, B. A. 2003. Studying the use of handhelds to control smart appliances. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCS '03)*. 274–279.

NICHOLS, J., ROTHROCK, B., CHAU, D. H., AND MYERS, B. A. 2006b. Huddle: Automatically generating interfaces for systems of multiple appliances. In *Proceedings of User Interface Software and Technology Conference (UIST)*. 279–288.

PATERNO, F., MANCINI, C., AND MENICONI, S. 1997. ConcurTaskTrees: A diagrammatic notation for specifying task models. In *Proceedings of the INTERACT Conference*. 362–269.

PONNEKANTI, S. R., LEE, B., FOX, A., HANRAHAN, P., AND WINOGRAD, T. 2001. ICrafter: A service framework for ubiquitous computing environments. In *Proceedings of the UBICOMP Conference*. 56–75.

PUERTA, A. AND EISENSTEIN, J. 2002. XIML: A common representation for interaction data. In *Proceedings of the 7th International Conference on Intelligent User Interfaces*. 214–215.

PUERTA, A. R. 1997. A model-based interface development environment. *IEEE Softw. 14*, 41–47.

SCHAEFER, R., BLEUL, S., AND MUELLER, W. 2006. Dialog modelling for multiple devices and multiple interaction modalities. In *Proceedings of the 5th International Workshop on Task Models and Diagrams for User Interface Design*. 39–53.

SIMON, R., JANK, M., AND WEGSCHEIDER, F. 2004. A generic UIML vocabulary for device- and modality independent user interfaces. In *Proceedings of the World Wide Web Conference*. 434–435.

SPROAT, R., HUNT, A., OSTENDORF, P., TAYLOR, P., BLACK, A., LENZO, K., AND EDGINGTON, M. 1998. SABLE: A standard for TTS markup. In *Proceedings of the International Conference on Spoken Language Processing*.

SUKAVIRIYA, P., FOLEY, J. D., AND GRIFFITH, T. 1993. A second generation user interface design environment: The model and the runtime architecture. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems.* 375–382.

SZEKELY, P. 1996. Retrospective and challenges for model-based interface development. In *Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfac.* J. Vanderdonckt, Ed. Namur University Press, 1–27.

SZEKELY, P., LUO, P., AND NECHES, R. 1992. Facilitating the exploration of interface design alternatives: The HUMANOID model of interface design. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*. 507–515.

UPNP. 2005. Universal plug and play forum. http://www.upnp.org.

VANDER ZANDEN, B. AND MYERS, B. A. 1990. Automatic, look-and-feel independent dialog creation for graphical user interfaces. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*. 27–34.

WIECHA, C., BENNETT, W., BOIES, S., GOULD, J., AND GREENE, S. 1990. ITS: A tool for rapidly developing interactive applications. *ACM Trans. Inform. Syst. 8*, 204–236.

ZIMMERMANN, G., VANDERHEIDEN, G., AND GILMAN, A. 2002. Prototype implementations for a universal remote console specification. In *Proceedings of the Conference on Computer-Human Interaction.* 510–511.

Received January 2009; revised June 2009; accepted July 2009