Describing Appliance User Interfaces Abstractly with XML

Jeffrey Nichols[*], Brad A. Myers[*], Kevin Litwack[*], Michael Higgins[†], Joseph Hughes[†], Thomas K. Harris[*]

[*]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{jeffreyn, bam, klitwack, tkharris}@cs.cmu.edu
http://www.pebbles.hcii.cmu.edu/puc/

[†]MAYA Design, Inc.
Suite 702
2100 Wharton Street
Pittsburgh, PA 15203
{higgins, hughes}@maya.com

ABSTRACT

This paper describes an XML-based language for describing the functions of appliances, such as televisions, VCRs, copiers, microwave ovens, and even manufacturing equipment. Our description language is designed to be concise, easy to use, and contain no presentation information. It has been used to describe more than twenty diverse appliances. The functional descriptions written in our language are used to automatically generate remote control interfaces for appliances. We have used these descriptions to generate both graphical and speech interfaces on handheld computers, mobile phones, and desktop computers.

Keywords

User interface description language (UIDL), automatic interface generation, remote control, appliances, personal digital assistants (PDAs), handheld computers, Pebbles, personal universal controller (PUC)

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. In this case, a UIDL is particularly beneficial because most of the code and implementation time is spent on the user interfaces in today's desktop applications. Without a multi-platform UIDL solution, even more will be required as applications become more distributed and user interfaces to those applications are needed on multiple platforms.

We are developing a UIDL for describing appliance user interfaces as part of our work on the *personal universal controller* (PUC) project [7]. The goal of the 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 use 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.

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. Not only must it describe the appliance in sufficient detail for the interface generators to create a high-quality interface, but it must be concise and short enough to be efficiently transmitted across wireless networks and parsed by embedded computing devices. Our specification language must also be descriptive enough to cover the complete functionality of any appliance, so that a PUC device can generate a complete user interface. Finally, the language needs to be abstract enough so that interfaces can be generated on multiple platforms from the same appliance specification.

This paper starts by discussing related work, both to our UIDL and the PUC system as a whole. Then we elaborate on the design principles for our UIDL, followed by a description of a study we conducted to inform our design. Then the language is described in detail, followed by some brief analysis of the strengths and weaknesses we have found in the current design.[‡]

RELATED WORK

A number of research groups are working on controlling appliances from handheld devices. Hodes, *et. al.* propose a similar idea to our PUC, which they call a "universal interactor" that can adapt itself to control many devices [4]. However, their research focuses on the system and infra-

Submitted for Publication

[‡] Portions of this paper are adapted from previously published material in [7] and [9].

structure issues rather than how to create the user interfaces. An IBM project [3] describes a "Universal Information Appliance" (UIA) that might be implemented on a PDA. The UIA uses an XML-based language called MoDAL from which it creates a user interface panel for accessing information. However, the MoDAL processor apparently only handles simple layouts and its only type of input control is text strings. The Stanford ICrafter [12] is a framework for distributing appliance interfaces to many different controlling devices. While their framework supports the automatic generation of interfaces, their paper focuses on hand-generated interfaces and shows only one simple automatically generated interface. They also mention the difficulty of generating speech interfaces.

The Xweb system [10] is an infrastructure that supports automatic generation of user interfaces from abstract descriptions, and supports multiple generation platforms, including speech. The PUC extends these ideas by adding more detail in the specification language and supporting more features in the automatic interface generation process.

The INCITS/V2 [20] standardization effort is developing the Alternative Interface Access Protocol (AIAP) to help disabled people use everyday appliances with an approach similar to the PUC. AIAP contains a description language for appliances that different interface generators use to create interfaces for both common devices, like the PocketPC, and specialized devices, such as an interactive braille pad. We are collaborating with the V2 group and they have incorporated many of our language ideas into their standard.

A number of research systems have looked at automatic design of user interfaces for conventional applications. These sometimes went under the name of "model-based" techniques [16]. Here, the programmer provides a specification ("model") of the properties of the application, along with specifications of the user and the display. Of particular note are the layout algorithms in the DON [5] and TRIDENT [18] systems that achieved pleasing, compact, and logical placements of controls. We extend these results to create panels of controls on significantly different handhelds, without requiring designer intervention after the interfaces are generated.

UIML [1] is an XML language that is designed to provide a highly-device independent method for user interface design. UIML differs from the PUC in its tight coupling with the interface. Designers using UIML can define the types of components to use in an interface and the code to execute when events occur. The PUC specification language leaves these decisions up to each platform's interface generator.

Recently a new general purpose language has been introduced for storing and manipulating interaction data. The eXtensible Interface Markup Language (XIML) [13] is an XML-based language that is capable of storing most kinds of interaction data, including the types of data stored in the application, task, and presentation models of other model-based systems. XIML was developed by RedWhale Software and is being used to support that company's user interface consulting work. They have shown that the language is useful for porting applications across different platforms and storing information from all aspects of a user interface design project.

It is common to find task information included in abstract UIDLs, though the PUC language does not yet include such information. Paterno's ConcurTaskTrees [11] is one such language, which has a graphical notation and allows the specification of concurrent tasks (not possible in earlier languages). ConcurTaskTrees also allows the specification of who/what is performing the task, whether it be the user, the system, or an interaction between the two.

DESIGN PRINCIPLES

Before and during the design of the specification language, we developed a set of requirements and principles on which to base our design [8]. Some of the principles are:

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

Sufficient detail to generate a high-quality interface. We conducted a user study 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 readable labels or adequate grouping information that is needed for generating a good user interface. For example, the Universal Plug and Play (UPnP) standard [17] includes an appliance description language that does not include sufficient detail for generating a good user interface.

No specific layout information should be included in the specification language. We wanted to ensure that our language would be general enough to work for interface generators running on a wide-variety of platforms. Another solution for addressing the multi-platform problem is to include multiple concrete interface descriptions in the appliance specification (the INCITS/V2 standard [20] and ICrafter [12] support this approach). 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.

Short and concise are very important principles for the design of our language. Appliance specifications must be sent over wireless networks and processed by computing devices that lack the power of today's desktop machines.

To ensure performance is adequate, the specification language must be concise. Why then choose a verbose format like XML as the basis for our language? We chose XML because it was easy to parse and there were several available parsers. XML is also a very compressible format, which can reduce the cost of sending specifications over the network, though our system does not use any compression.

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

PRELIMINARY USER STUDIES

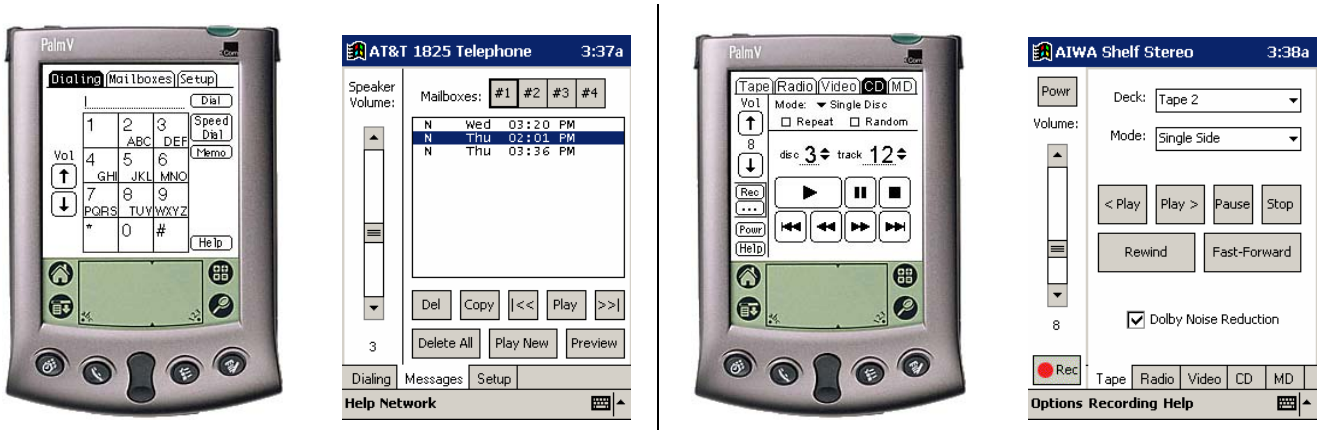
These principles guide the design our language, but do not suggest what information should be included or what level of detail is needed to automatically generate a high-quality interface. In order to determine what content should be included in a specification, we hand-designed several remote control interfaces for existing appliances. Then user studies were conducted to compare the hand-designed interfaces to the manufacturers' interfaces (described in more detail in [6]). This approach allowed us to concentrate on the functional 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 is the standard unit installed in many offices at Carnegie Mellon. Aiwa-brand stereos seem to be particularly common (at least among our subject population) because ten of our twenty-five subjects owned Aiwa systems.

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 [6]. 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 [6].

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.



(a) (b) (c) (d)

Figure 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.

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 ignore some deeper branches.

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 1c-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 most of the functions of an appliance were manipulating some data in a definable way, but some were not. For example, the tuning function of a radio is manipulating the current value of the radio station by a 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 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 functions. For example, the Palm stereo interface (see Figure 1c) used the label "Vol" to refer to volume, whereas the PocketPC stereo interface (see Figure 1d) 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 that an interface generator could choose between 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 recently developed a solution for addressing this problem called Smart Templates [9], which will be discussed in the next section.

SPECIFICATION LANGUAGE

The PUC specification language is XML-based and includes all of the information that we found in our analysis of the hand-designed interfaces. This section describes the features of our language and shows examples of how each feature is used. A language reference can be downloaded from our project web site:
http://www.cs.cmu.edu/~pebbles/puc/specification.html

State Variables, Commands, and Explanations

Our specification language supports three primitive elements for describing the functions of an appliance. We discovered from our PocketPC implementations that most appliance functions could be represented as state variables. Each state variable has a given type that tells the interface generator how it can be manipulated. For example, the radio station state variable has a numeric type, and the interface generator can infer the tuning function because it knows how to manipulate a numeric type. Other state variables include the current track of the CD player and the status of the tape player (stop, play, fast-fwd, etc.).

As mentioned above, we discovered that state variables are not sufficient for describing all of the functions of an appliance, such as the seek button on a radio. The seek function must be represented as a *command*, a function whose result cannot be described easily in the specification. Figure 2 shows examples of both state variables and commands.

Commands are also useful when an appliance is unable to provide state information to the controller, either by manufacturer choice or a hardware limitation of the appliance. For example, up and down commands might be used for volume if the appliance cannot support an integer-typed state variable. In fact, the remote control technology of today can be simulated on the PUC by writing a specification that includes only commands. Any state information must then be shown on the appliance's front panel.

Our specification language also has a feature called an explanation. Explanations are static labels that are important enough to be explicitly mentioned in the user interface, but are not related to any existing state variable or command. For example, an explanation is used in one specification of a shelf stereo to explain the Auxiliary audio mode to the user. The mode has no user controls, and the explanation is used to explain this. Explanations are used very rarely in the specifications that we have written.

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 Controls.Mode state in Figure 2 has

an enumerated 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.

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

Label Information

The interface generator must also have information about how to label the interface components that represent state variables and commands. Providing this information is difficult because different form factors and interface modalities require different kinds of label information. An interface for a mobile web-enabled phone will probably require smaller labels than an interface for a PocketPC with a larger screen. A speech interface may also need phonetic mappings and audio recordings of each label for text-to-speech output. We have chosen to provide this information with a generic structure that we call the *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 [2]) and text-to-speech labels that may contain text using SABLE mark-up tags [15] 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. Thus the interface generator can use any label within a label dictionary interchangeably. For example, this allows a graphical interface generator to use a longer, more precise label if there is lots of available screen space, but still have a reasonable label to use if space is tight. Figure 2 shows the label dictionary, represented by the `<labels>` element, for a number of states. The dictionary for the Controls group has two text labels and a text-to-speech label.

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 play button for the CD player might be placed next to the stop button for the Tape player, creating an unusable interface. We avoid this by explicitly specifying grouping information using a group tree.

We specify the group tree as an *n*-ary tree that has a state variable or command at every leaf node (see Figure 3).

```
<?xml version="1.0" encoding="utf-8"?>
<spec name="MediaPlayer" version="PUC/2.0">
  <labels>
    <label>Media Player</label>
  </labels>

  <groupings>
    <group name="Controls" is-a="media-controls">
      <labels>
        <label>Play Controls</label>
        <label>Play Mode</label>
        <text-to-speech text="Play Mode"
                        recording="playmode.au"/>
      </labels>

      <state name="Mode">
        <type>
          <enumerated>
            <item-count>3</item-count>
          </enumerated>
          <valueLabels>
            <map index="1">
              <label>Stop</label>
            </map>
            <map index="2">
              <label>Play</label>
            </map>
            <map index="3">
              <label>Pause</label>
            </map>
          </valueLabels>
        </type>

        <labels><label>Mode</label></labels>
      </state>

      <group name="TrackControls">
        <command name="PrevTrack">
          <labels><label>Prev</label></labels>

          <active-if>
            <greater-than state="PList.Selection">
              0
            </greater-than>
          </active-if>
        </command>

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

          <active-if>
            <less-than state="PList.Selection">
              <ref-value state="PList.Length"/>
            </less-than>
          </active-if>
        </command>
      </group>
    </group>

    <list-group name="PList">
      <state name="Title">
        <type><string/></type>
        <labels><label>Title</label></labels>
      </state>

      <state name="Duration" is-a="time-duration">
        <type><integer/></type>
        <labels><label>Duration</label></labels>
      </state>
    </list-group>
  </groupings>
</spec>
```

Figure 2. A sample specification for a media player with a few basic functions and a play list.

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. We encourage designers to make the group tree as deep as possible, in order to help space-constrained interface generators. These generators can use the extra detail in the group tree to decide how to split a small number of controls across two screens. Interface generators for larger screens can ignore the deeper branches in the group tree and put all of the controls on one panel.

Complex Types

Our specification language uses the group tree to specify complex type structures often seen in programming languages, such as records, lists, and unions. We chose this approach because we felt it simplified our language, and followed the principle of "one way to specify anything." If we had chosen to specify complex types 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, we have added several special group tags. Figure 2 shows an example of the `list-group` tag that we added for specifying lists. List groups have two implicit variables to track the length of the list and the current selection(s). State variables that are specified within the list group will have multiple values associated with them, one for each item in the list. Multi-dimensional lists can be created by nesting list groups. We have also developed a special dependency operator for lists that can be true if all items, any items, or no items in the list match a dependency equation. The `union-group` is similar to the `list-group`, but acts like a union from the C programming language.

Dependency Information

The two-way communication feature of the PUC allows it to know when a particular state variable or command is unavailable. This can make interfaces easier to use, because the components representing those elements can be disabled. The specification contains formulas (see the `<active-if>` element in Figure 2) that specify when a state or command will be disabled depending on the values of other state variables, currently specified with several types of dependencies: equal-to, greater-than, less-than, defined, and others. Each state or command may have multiple dependencies associated with it, combined with the logical operations AND and OR. These formulas can be processed by the PUC to determine whether a component should be enabled when the appliance state changes.

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 interface. For example, dependency information can help the speech interfaces interpret phrases by eliminating all possibilities that are not currently available. The processing of these formulas is described elsewhere [7].

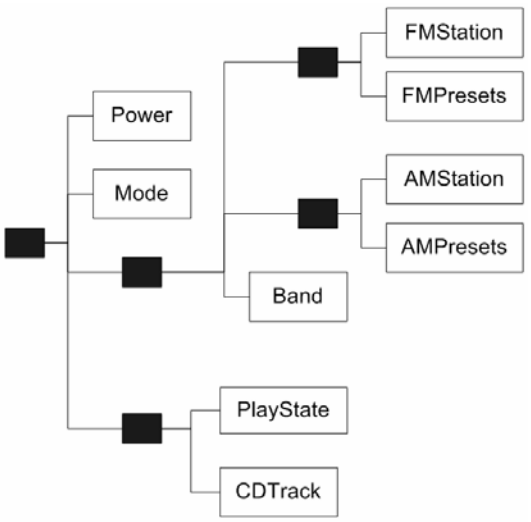


Figure 3. A sample group tree for a shelf stereo with both a CD player and radio tuner. The black boxes represents groups and the white boxes with text represent state variables. The mode variable indicates which source is being played through the speakers.

Smart Templates

A common problem for automatic interface generators has been that their interface designs do not conform to domain-specific design conventions 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 [19] have dealt with this problem by defining specific rules for each application that apply the appropriate design conventions. Other systems [5] rely on human designers to add design conventions to the interfaces after they are automatically generated. Neither of these solutions is acceptable for the PUC system. Defining specific rules for each appliance will not scale, and a PUC device cannot rely on user modifications because its user is not likely to be a trained interface designer. Even if the user was trained, he or she is unlikely to have the time or desire to modify each interface after it is generated, especially if the interface was generated in order to perform a specific task.

We have developed one solution to this problem called *Smart Templates* [9], where the PUC specification language's primitive type information is augmented with high-level semantic information. For example, the `media-controls` template defines that a state variable with particular attributes controls the playback of some media. Figure 2 shows how the `media-controls` Smart Template is indicated using the `is-a` attribute in our specification language. PUC interface generators can use the information added by a Smart Template to apply design conventions and make interfaces more usable. If an interface generator does not recognize a template however, a

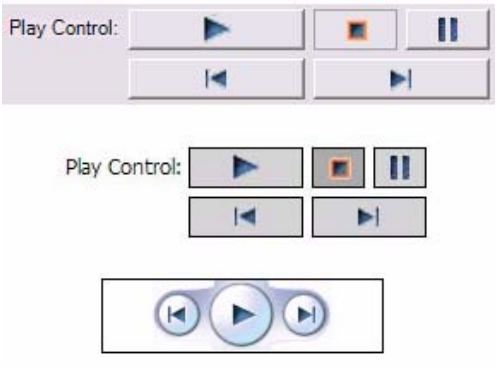


Figure 4. Media controls rendered for a Windows Media Player interface on each of our three platforms. At the top is the desktop, the middle is PocketPC, and the bottom shows Smartphone. The Smartphone control maintains consistency for the user by copying the layout for the Smartphone version of Windows Media Player, the only media player application we could find on that platform. This interface overloads pause and stop onto the play button.

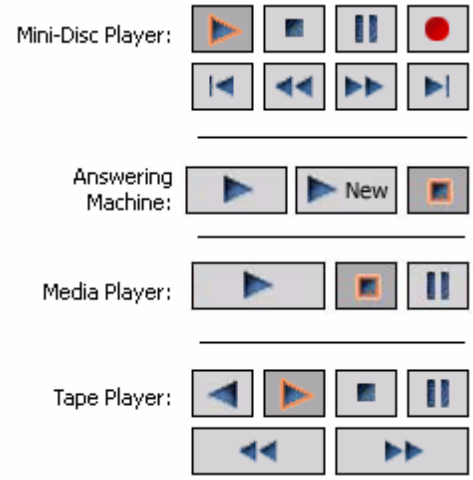


Figure 5. Different arrangements of media playback controls automatically generated for several different appliances from a single Smart Template (`media-controls`).

user interface can still be created because Smart Templates are constructed from the primitive elements of our specification language. Figure 4 shows the same instance of a Smart Template rendered on different platforms.

An important innovation is that Smart Templates are parameterized, which allows them to cover both the common and unique functions of an appliance. For example, the media playback template supports play and stop, but also optional related functions such as next track for CDs, fast-forward and reverse-play for tape players, and "play new" for phone answering machines (see Figure 5). Smart Templates also give appliances the flexibility to choose a functional representation that matches their internal implementation. For example, our `time-duration` Smart Template allows single state variables with integer or string types, or multiple state variables (e.g. a state for hours and another for minutes).

We have built a preliminary implementation of Smart Templates into the existing PUC system. So far the PUC supports a few Smart Templates: `media-controls`, `time-duration`, `image`, and `image-list`. We plan to implement many more, including `date`, `mute`, `power`, and `volume`. We expect that some Smart Templates will naturally combine with others to create new templates. For example, date and time are often used together, as are volume and mute. We hope to implement Smart Templates in such a way that templates can be flexibly combined with less work than creating a new template from scratch.

EVALUATION

We have not yet conducted any formal evaluation of our specification language, but we have used it to specify more than twenty appliances ranging from stereos and telephones to elevators and car navigation systems. We have used those specifications as the basis for generating graphical user interfaces on PocketPCs, Microsoft Smartphones, and desktop computers, and speech user interfaces using the Universal Speech Interfaces framework [14]. This section informally discusses some strengths and weaknesses that we have found with the language.

The main strengths of the language come from the design principles that we started with. Appliance descriptions are often a reasonable size, even for our largest and most complicated appliances. Our specification for an Audiophase 5CD shelf stereo system, which has more than 50 states, is 25KB. The specification for the GM Yukon Denali navigation system, which has more than 80 states, is 41KB. These sizes are perfectly reasonable for transmission and processing on the devices we are targeting.

Our language also seems to be reasonably easy to learn. Four undergraduate students have learned the language over the course of the project. Each student picked up the basics of the language in a day and was proficient within about two weeks. The most difficult aspects of writing an appliance specification are determining the appliance's variables and commands, and designing the group tree structure. We believe these aspects are difficult in general, and do not represent weaknesses in our design.

The main weakness of the language is the lack of any task information. For many appliances this is not a problem because all of the tasks have only one step. For example, "play the tape" or "increase the volume." With complex appliances, such as a car navigation system, this is not always the case, and the lack of task information may lead to lower quality generated interfaces for these appliances.

FUTURE WORK

We are planning to conduct a formal evaluation of the specification language and interface generators as part of the first author's thesis work. This will involve specifying more complex appliances and further testing the descriptiveness of the language.

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

This feature will require some additions to our language to describe how appliances are connected together. Task information will also be required to support features like the "Play DVD" button, but the task information will be distributed among each of the different appliances. This is different from previous task languages [11] which have assumed that all task information is available in one location. Designing and building this distributed task language is a major area of future work for the PUC project.

CONCLUSION

We have discussed a language for describing appliances. The language is the basis for a system that automatically generates remote control user interfaces. We have used our specification language to describe more than twenty appliances from telephones to elevators to vehicle navigation systems. We have also written software that uses our language to automatically generate graphical user interfaces for handheld computers, mobile phone, and desktop computers, and speech interfaces using the Universal Speech Interfaces framework. We believe that the PUC specification language is at the appropriate level, and contains the right features to be successfully used for virtually all appliances and for many other tasks as well. We would welcome widespread adoption of the PUC specification language and collaboration with others.

ACKNOWLEDGMENTS

This work was conducted as a part of the Pebbles project, and the speech interface generator was implemented as a part of the Universal Speech Interfaces project. This work was funded in part by grants from NSF, Microsoft, General Motors, and the Pittsburgh Digital Greenhouse, and equipment grants from Mitsubishi Electric Research Laboratories, VividLogic, Lutron, and Lantronix. The National Science Foundation funded this work through a Graduate Research Fellowship for the first author and under Grant No. IIS-0117658. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

REFERENCES

1. Abrams, M., Phanouriou, C., Batongbacal, A.L., Williams, S.M., and Shuster, J.E. "UIML: An Appliance-Independent XML User Interface Language," in *The Eighth International World Wide Web Conference.* 1999. Toronto, Canada
2. CMU, "Carnegie Mellon Pronuncing Dictionary," 1998. http://www.speech.cs.cmu.edu/cgi-bin/cmudict.
3. Eustice, K.F., Lehman, T.J., Morales, A., Munson, M.C., Edlund, S., and Guillen, M., "A Universal Information Appliance." *IBM Systems Journal*, 1999. **38**(4): pp. 575-601.
4. Hodes, T.D., Katz, R.H., Servan-Schreiber, E., and Rowe, L. "Composable ad-hoc mobile services for universal interaction," in *Proceedings of the Third annual ACM/IEEE international Conference on Mobile computing and networking (ACM Mobicom'97).* 1997. Budapest Hungary: pp. 1-12.
5. Kim, W.C. and Foley, J.D. "Providing High-level Control and Expert Assistance in the User Interface Presentation Design," in *Proceedings INTERCHI'93: Human Factors in Computing Systems.* 1993. Amsterdam, The Netherlands: pp. 430-437.
6. Nichols, J., Myers, B.A. "Studying The Use Of Handhelds to Control Smart Appliances," in *23rd International Conference on Distributed Computing Systems Workshops (ICDCS '03).* 2003. Providence, RI: pp. 274-279.
7. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Pignol, M. "Generating Remote Control Interfaces for Complex Appliances," in *UIST 2002.* 2002. Paris, France: pp. 161-170.
8. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Shriver, S. "Requirements for Automatically Generating Multi-Modal Interfaces for Complex Appliances," in *ICMI.* 2002. Pittsburgh, PA:
9. Nichols, J., Myers, B.A., Litwack, K. "Improving Automatic Interface Generation with Smart Templates," in *Intelligent User Interfaces.* 2004. Funchal, Portugal: pp. 286-288.
10. Olsen Jr., D.R., Jefferies, S., Nielsen, T., Moyes, W., and Fredrickson, P. "Cross-modal Interaction using Xweb," in *Proceedings UIST'00: Symposium on User Interface Software and Technology.* San Diego, CA: pp. 191-200.
11. Paterno, F., Mancini, C., Meniconi, S. "ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models," in *INTERACT.* 1997. Sydney, Australia: pp. 362-269.
12. Ponnekanti, S.R., Lee, B., Fox, A., Hanrahan, P., and T.Winograd. "ICrafter: A service framework for ubiquitous computing environments," in *UBICOMP 2001.* 2001. Atlanta, Georgia: pp. 56-75.
13. Puerta, A., Eisenstein, J. "XIML: A Common Representation for Interaction Data," in *7th International Conference on Intelligent User Interfaces.* 2002. San Francisco: pp. 214-215.
14. Rosenfeld, R., Olsen, D., Rudnicky, A., "Universal Speech Interfaces." *interactions: New Visions of Human-Computer Interaction*, 2001. **VIII**(6): pp. 34-44.
15. Sproat, R., Hunt, A., Ostendorf, P., Taylor, P., Black, A., Lenzo, K., Edgington, M. "SABLE: A Standard for TTS Markup," in *International Conference on Spoken Language Processing.* 1998. Sydney, Australia:
16. Szekely, P. "Retrospective and Challenges for Model-Based Interface Development," in *2nd International Workshop on Computer-Aided Design of User Interfaces.* 1996. Namur: Namur University Press. pp. 1-27.
17. UPnP, "Universal Plug and Play Forum," 2003. http://www.upnp.org.
18. Vanderdonckt, J. "Knowledge-Based Systems for Automated User Interface Generation: the TRIDENT Experience," in *Technical Report RP-95-010.* 1995. Namur: Facultes Universitaires Notre-Dame de la Paix, Institut d' Informatique:
19. Wiecha, C., Bennett, W., Boies, S., Gould, J., and Greene, S., "ITS: A Tool for Rapidly Developing Interactive Applications." *ACM Transactions on Information Systems*, 1990. **8**(3): pp. 204-236.
20. Zimmermann, G., Vanderheiden, G., Gilman, A. "Prototype Implementations for a Universal Remote Console Specification," in *CHI'2002.* 2002. Minneapolis, MN: pp. 510-511.