Improving Automatic Interface Generation with Smart Templates

Jeffrey Nichols, Brad A. Myers and Kevin Litwack
Human Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
{jeffreyn, bam, klitwack}@cs.cmu.edu
http://www.pebbles.hcii.cmu.edu/puc/

ABSTRACT
One of the challenges of using mobile devices for ubiquitous remote control is the creation of the user interface. If automatically generated designs are used, then they must be close in quality to hand-designed interfaces. Automatically generated interfaces can be dramatically improved if they use standard conventions to which users are accustomed, such as the arrangement of buttons on a telephone dial-pad or the conventional play, stop, and pause icons on a media player. Unfortunately, it can be difficult for a system to determine where to apply design conventions because each appliance may represent its functionality differently. *Smart Templates* is a technique that uses parameterized templates in the appliance model to specify when such conventions might be automatically applied in the user interface. We show that our templates easily adapt to existing appliance models and that interface generators on different platforms can apply appropriate design conventions using templates.

Keywords
Automatic interface generation, Pebbles, handheld computers, appliances, personal digital assistants (PDAs), personal universal controller (PUC)

INTRODUCTION

Everyday home and office appliances, including photocopiers, televisions, DVD players, and telephones, are being designed with increasingly many complex functions. Unfortunately, the user interfaces for these appliances often get harder to use as more computerized features are added [1]. One solution to this problem is to move the interface on every appliance to an intermediary intelligent "user interface" device. Such a device could improve interfaces for users by creating similar interfaces for appliances that have similar functions, and by combining the interfaces for a group of related appliances into a single user interface.

Submitted for Publication.

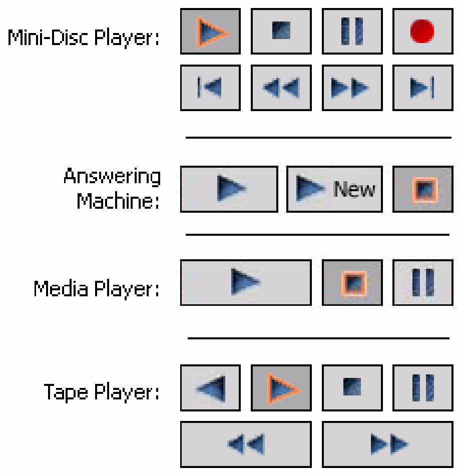

Figure 1. Different arrangements of media playback controls automatically generated for several different appliances from a single Smart Template (`media-controls`).

We are building a system called the *personal universal controller* (PUC) that supports UI devices with these features. The system relies on intelligent automatic user interface generators that build interfaces from abstract descriptions of the appliances. A PUC controller device, which might be a personal digital assistant (PDA), a mobile phone, or even a wristwatch, engages in two-way communication with each appliance that it controls, first downloading the appliance's description, automatically generating a user interface, and then sending control signals and receiving feedback on the appliance's state. The PUC appliance specification language includes only abstract information about the functionality of the appliance and no specific information about which user interface controls should be used or where they should be placed on the screen.

A well-known problem for automatic interface generators has been their inability to recognize when to apply user interface conventions because of the vast differences in appliances. We have added a new feature to the PUC called *Smart Templates*, which is novel because it simultaneously

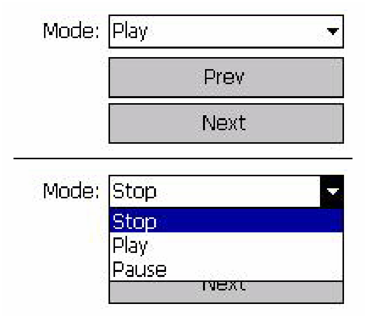

Figure 2. A rendering of a play control in an early PUC that represents the play, stop, and pause functions with a pull-down selection list, and previous and next track with separate buttons. The bottom view shows the selection list pulled down.

gives the interface generator semantic information and the appliance implementer flexibility to choose a representation for that semantic information. Interface generators are free to interpret the semantic information and apply design conventions as necessary. If an interface generator does not recognize the template, it can still be rendered because all templates are defined using the primitive elements of our specification language [7]. An important innovation is that our 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 1). We have used Smart Templates to represent complex features such as various kinds of media playback controls and to extend our built-in type system with basic elements such dates and times.

A key difference between the PUC system in general and most previous model-based work is that our automatically generated interfaces are expected to be complete and usable without further modification. We do not expect that a user will want to walk up to their office photocopier with an important document, pull out their PUC controller device, and then spend time fixing the organization and layout of their copier user interface. In order to meet this requirement, the PUC system has several novel features. Our appliance specification language includes multiple versions of every label and the interface generators intelligently choose which label to use based on available space and other factors. The language also includes dependency information, which describes whether features of the appliance are available with respect to the state of the appliance. Dependency information is not only useful for enabling and disabling user interface controls, but also for determining how a user interface can be structured.

We have built three different interface generators. Two generate graphical user interfaces for Microsoft's PocketPC and Microsoft's Smartphone. Another generates speech user interfaces using the Universal Speech Interfaces framework [12]. Each of the graphical generators builds interfaces of a different style; the PocketPC generator creates standard graphical interfaces that can also be used on desktop computers. The Smartphone generator creates list-based interfaces that are suitable for a device with only four navigation buttons and no touch-screen. Our specification language and interface generators are described in more detail elsewhere [7].

The first version of the PUC system [7] had several problems. First of all, the type system used in our specification language supported only types similar to those that are built into programming languages, and thus could not express complex semantics. Even relatively basic types such as dates and times could not be expressed. Clearly our system must be able to deal with these complex types if it is to generate high quality user interfaces. Unlike for programming languages however, it is unreasonable to allow specification designers to extend the set of types in their specification because the interface generators must be able to properly deal with each type. In practice, this often requires that new rules be added to each interface generator.

One solution to this problem would be to add some new types for basic cases like date and time. This presents a problem for building the PUC system into appliances however, because appliances may use different representations, even for common types. For example, one appliance might implement time as an integer representing the number of seconds since some epoch, whereas another appliance might use a human-readable string. We believe it is important to support the wide-range of possible appliance implementations while still providing consistency to users.

Another problem with our first system was that our interface generators did not use conventional layouts in their automatic designs. For example, a set of play controls on a stereo are often represented as several buttons on hand-designed interfaces, whereas our automatic designs used a pull-down selection list (see Figure 2). There are many situations where conventional layouts or designs could be applied in everyday appliance interfaces (e.g., the dialpad on a telephone, parallel vertical sliders for a graphic equalizer, a power button with standard icon, a volume slider with standard right triangle icon, proper left/right orientation of a control for audio balance, etc.). Creating interfaces that are consistent with what users expect will increase the usability of our automatically generated interfaces.

In order to solve these problems with extensible types and conventional layouts, we developed Smart Templates, which are pre-defined groupings of state variables and commands, the primary elements of the PUC specification language. Each Smart Template supports a number of different representations. In order to represent a date, for example, a template will allow the appliance to use a single state with a string type, two states with integer types representing the day and month, or three states with integer types representing the day, month, and year. The ability to have different representations gives the specification writer flexibility to pick one that matches the appliance implementation.

Since the templates are implemented using the primitive elements of the language, interface generators can choose not to recognize some Smart Templates. In some cases it may even be preferable for an interface generator to ignore a Smart Template. For example, many of the conventions that can be applied to graphical interfaces do not have similar analogs for speech interfaces. The time and date Smart Templates would be used by a speech interface, but it might not be necessary to use the media controls template because the rendering from primitive elements would be the same. For those templates that are not recognized, the primitive type information is available and sufficient to generate a full user interface for the same functions.

When the interface generator does recognize a Smart Template, it can take a number of actions to improve the generated interface for the user. For many templates an interface generator will want to use special layout rules. For example, the PocketPC generator uses special rules to create the different arrangements of media controls shown in Figure 1. For certain templates, the interface generator is able to use platform-specific controls to improve interface consistency for the user. For example, each of our interface generators chooses the time manipulation control appropriate for its platform when a time Smart Template is encountered (see Figure 3).

As noted above, the personal universal controller (PUC) has been described elsewhere [7], including the requirements for our specification language and infrastructure [8] and the early user studies that form the basis for our work [6]. All of these papers mention the need to describe high-level semantic information in the specification language, but here we present the first discussion of the solution to that problem.

This paper begins by discussing related work and then gives more background on the PUC specification language. The following section discusses Smart Templates in more detail, examining how the templates are defined and how we have implemented them in our interface generators. We finish with future work and some concluding thoughts.

RELATED WORK

A number of systems exist for controlling appliances, many of which support the automatic generation of user interfaces. Hodes et al. propose an idea similar to the PUC called a "universal interactor" [2], but their work focuses more on the infrastructure issues and less on the generated interfaces. The XWeb [10] project automatically generates an interface from an XML-based specification language. XWeb includes less information in its specification language however, and does not deal well with appliance modes. The Stanford ICrafter [11] is a system for distributing appliance interfaces to a number of different clients. While it does support automatic generation, it is typically

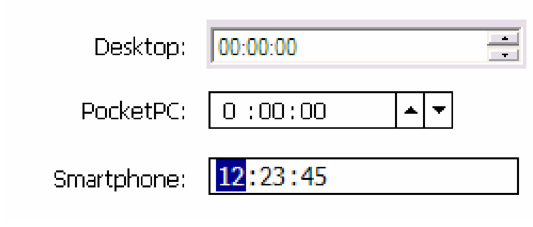

Figure 3. The standard controls used for manipulating durations of time, as generated by our system, on all three of our platforms.

used for distributing hand-designed user interfaces. SUPPLE [15] uses a decision-theoretic approach to find the optimal layout and choice of controls for a user interface. It does not use design conventions in its layouts however, though that might be possible with extensions to their description language, which seems to be similar to ours. The Ubiquitous Interactor (UBI) [9] creates automatic designs that may apply rules from the appliance manufacturer to add branding or other qualities to the user interface. It is possible that the UBI's customization forms could cause the interface generator to use design conventions, but it is not clear whether these would be portable between different appliances. With the possible exception of UBI, none of these systems have any support for applying conventional layouts in their automatic designs, though all could probably be extended with the Smart Template idea in the future.

Earlier model-based systems had some support for using conventional layouts in their automatic designs. ITS [16] allows the specification of style rules, which could be shared across multiple applications. A style rule could create a conventional layout if it found particular relationships in ITS's action or dialog layers. In practice, however, it was found that it was difficult to write and share style rules. HUMANOID [13] has the ability to apply custom displays to application-specific types. These displays are specific to the application however, and it is not clear whether they can be easily shared among applications. The display would also need to be modified if the representation of the type changed between applications, which is not necessary with Smart Templates. Most other model-based systems offered interface designers the ability to modify the interfaces after they were generated [3, 14]. An important principle of Smart Templates is that no modifications are necessary since they allow conventional layouts to be applied automatically during interface generation.

PUC SPECIFICATION LANGUAGE

Before describing Smart Templates, we first provide a brief overview of the PUC's XML-based specification language and the features of the language that are discussed later in this paper. Full documentation is available on our project web site [5].

The functions of an appliance are represented by *state variables* and *commands*. State variables have primitive types that define the data they contain, such as integer, string, or enumeration. The interface generators infer from the type the type the operations that are possible on the state variable, so it is not required that a command be supplied for the common manipulations of the state variable. Commands can be used to specify manipulations that cannot be inferred directly from a variable's type. One example of a command is the seek function for a radio station. The station itself might be represented as a variable, but seek cannot be inferred from the variable because the controller cannot know in advance what the next radio station with good reception will be. After the seek command is invoked, the appliance can change the radio station variable's value as appropriate.

Organization is specified in our language via a hierarchy of groups. Variables and commands can be placed anywhere in the hierarchy, not just at the leaves. The hierarchy is used for structuring the interface and making layout decisions. We encourage specification designers to make the hierarchy deep so that space-constrained interface generators have as much information as possible about how controls can be grouped. Interface generators for larger screens can ignore the deeper branches and group more controls onto a single panel.

Another important feature of our specification language is dependency information. This information describes whether a state variable or command is available for use based on the values of the appliance's other state variables. Dependency information defines when an interface element should be enabled or disabled and is also used by our interface generators to determine which branches of the hierarchy are more important. Dependencies are defined using several different relations (e.g., equals, greater-than, less-than, defined, etc.) that are grouped with the logical operators AND and OR. Not only does dependency information allow graphical interfaces to display an indicator of whether the function is available (such as graying out unavailable controls), but it can also be useful for inferring information about the panel structure and layout of the interface. Appliances with modes especially benefit from this approach, because each mode is typically associated with several functions that are active only in that mode. In such cases, an interface generator can decide to place the functions for different modes on different overlapping panels.

SMART TEMPLATES

A Smart Template augments the grouping and primitive type information in the appliance specification language with knowledge about the semantics of the data. Smart Templates are *defined* in advance by template writers, who specify the fixed set of states and commands that can be included in a template. Some items are required and others are optional to allow for the different functions an appliance may support and their implementations (see Figure 4 for two supported examples for one particular template). Smart Templates are *used* in the appliance specification to add semantic information to either a group or state variable with the special `is-a` attribute (see Figure 4). Adding this

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

  <state name="Hours">
    <type>
      <integer>
        <min>0</min>
        <max>8</max>
      </integer>
    </type>
  </state>

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

```
<state name="SongLength" is-a="time-duration">
  <type>
    <integer>
      <min>0</min>
      <max>4440</max>
    </integer>
  </type>

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

Figure 4. Two examples of `time-duration` Smart Template instances from PUC appliance specifications.

Table 1. Smart templates defined for the PUC system.

Name	Represents…
`date`	Any date
`datetime`	The aggregate of the date and time-absolute types.
`image`	Any image, for use with our future binary primitive type.
`media-controls`	The interactions that control the playback of any audio/visual media
`mute-mic`	Mute for a microphone, as on a telephone
`mute-speaker`	Mute for speakers, as on a television or stereo.
`power`	The power button on any appliance.
`phone-dialpad`	The dialing pad on a telephone
`time-absolute`	The time-of-day.
`time-duration`	A duration of time, such as the length of a song or the counter on a VCR.
`volume`	The volume control on a stereo or telephone.

attribute requires the appliance specification to conform to the definition of that Smart Template. Finally, Smart Templates are *rendered* by interface generators based upon the chosen template and the content that was included within the template in the appliance specification.

This section divides the discussion of Smart Templates into two parts. First we discuss how templates are defined, elaborate on their flexibility for different representations of appliance data, and discuss the Smart Templates that we have defined so far. In the second sub-section we show how interface generators render Smart Templates with common controls and conventional layouts to ensure consistency for the user. Renderings are shown for several of our templates on each of our graphical interface generation platforms.

Definition

A template writer starts by selecting the state variables and commands that the template may contain, and then defines the names, types, values, and other properties that these elements must have. The challenge for the template writer 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, 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 5). Using our `time-duration` Smart Template, both of these representations can be handled appropriately by an interface generator, while still providing a consistent look and feel to users.

Allowing many combinations of states and commands in a template definition also allows a single Smart Template to be applied across multiple kinds of appliances. For example, the `media-controls` template optionally allows commands for next track and previous track for use with CD and MP3 players, a reverse play option for use with some tape players, and even a “play new” command for phone answering machines and voicemail systems (that plays only the messages that are tagged as being new).

We have defined a number of Smart Templates for use in the PUC system, shown in Table 1, and we intend to define many more in the future. In this paper we describe two representative templates, `time-duration` and `media-controls`, in detail and omit the others for brevity. All template definitions can be viewed on our web site [5].

The `time-duration` template allows many different state combinations to represent a length of time. One state may be used with any of the following primitive types:

- *Integer*, representing the number of seconds from zero. An upper bound may be added if appropriate, as is shown in the second example in Figure 4.
- *Fixed point number*, which uses the decimal portion of the number to represent milliseconds.
- *String*, with the string value provided in one of several formats (e.g., “xx:xx:xxx”).

A `time-duration` may also be represented with multiple states using the integer type, as shown in the first example in Figure 4. In this case, each state represents one unit of time and indicates this by using a pre-defined name, such as `Hours`, `Minutes`, or `Seconds`. Each state has bounds to match its unit. For example, if an `Hours` state is present, then the state representing minutes must have a value from 0 to 59. We believe these representations cover all of the common ways that an appliance might represent a length of time, but we can extend the definition if we find another representation.

The `media-controls` template allows interface generators to apply standard controls and icons to media playback. This template can be used on a variety of appliances, including CD players, audio tape players, VCRs, and answering machines. Two representations of play controls are allowed by this 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. Again, some functions must be represented by a command and others are optional. This template also allows three extra commands for functions that are commonly included in the same group as the play controls: the previous and next track functions for CD and MP3 players, and the play new function for answering machines.

Rendering

We have implemented the `time-duration` and `media-controls` Smart Templates in all of our graphical interface generators: PocketPC, Smartphone, and desktop. We have not yet implemented any Smart Templates for our speech interface generator, but we plan to do this in the near future. With these templates we have already seen many of the benefits we expected from our approach.

Smart Templates allow interface generators to use platform-specific controls that create consistency with other user interfaces on the same device. Using standard controls is a great improvement over the primitive controls that our interface generators would choose before Smart Templates (see Figure 2 and Figure 7). 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 (see Figure 3). Unfortunately, none of our platforms have built-in controls for media playback, so our `media-controls` Smart Template does not benefit in the same way. The Smartphone `media-controls` implementation does mimic the interface used by the Smartphone version of Windows Media Player however, so it still maintains some consistency with another application that is likely to be used on the Smartphone (Figure 8 shows examples on each platform).

Smart Templates are also able to intelligently choose a rendering based upon the contents of the template. For example, each implementation of the `time-duration` template only renders the time units that are meaningful, and each implementation of our `media-controls` Smart Template renders buttons for only the functions that are available. The `media-controls` implementations on the PocketPC and desktop take this further by intelligently laying out the buttons on one or more lines depending on space, enlarging buttons of greater importance such as Play, and using a grid to create aesthetically pleasing arrangements (see Figure 1).

An interface generator may also choose to use a different control based on the contents of a Smart Template. Our `time-duration` template implementations use different controls depending on whether the duration is bounded. For example, a slider is used by the PocketPC and desktop interface generators provided that the time has a minimum and maximum constraint. In Figure 5, the “Time” on the Sony Camcorder interface on the left does not have an upper bound and is read-only, so a label is used rather than a text field or a scrollbar. It is also possible for an appliance to specify that one variable has bounds in terms of another, as in a media player where the maximum value of the playback counter is the length of the currently playing song. Our implementations have a special-case rendering for these situations as shown in Figure 6 and also in the Windows Media Player interface shown in the right half of Figure 5.

Implementation

A large part of creating a Smart Template is implementing the rendering code for interface generators. As a Smart Template becomes more flexible for the appliance specification designer, more work is required for the implementer of the interface generator. Furthermore, this work must be repeated on every interface generation platform. We have created a framework for our interface generators to decrease the implementation cost, and have also found in practice that code can often be re-used across templates and even across platforms.

The code written for an interface generator to handle a Smart Template is typically made up of two parts:

- *Control Choice*, which analyzes the content that makes up a particular instance of a template and decides which control(s) should be rendered to represent it. If multiple controls are chosen, then the Smart Template code is also responsible for laying out the controls in an intelligent way. Often layouts are constructed using rules similar to those used throughout our interface generators (see [7]). In some

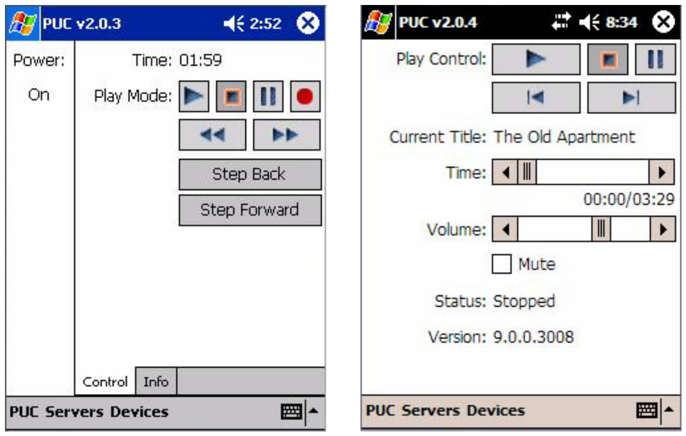

Figure 5. Two full user interfaces automatically generated on the PocketPC. Both interfaces show the `media-controls` and `time-duration` Smart Templates rendered together. On the left is an interface for a Sony DV Camcorder and on the right is an interface for Windows Media Player. The media player interface shows time as a scrollbar, because the play-back time is bounded by song length.

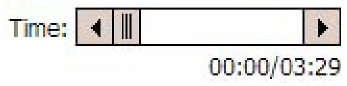

Figure 6. A rendering of a `time-duration` Smart Template with an upper bound equal to another `time-duration` template.

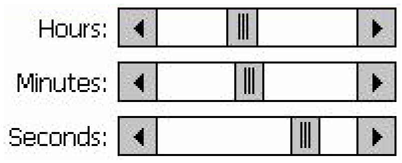

Figure 7. A rendering of an interface for manipulating time on an early PUC. Scrollbars were used in this rendering because all the units were bounded. In other cases, text fields would have been used.

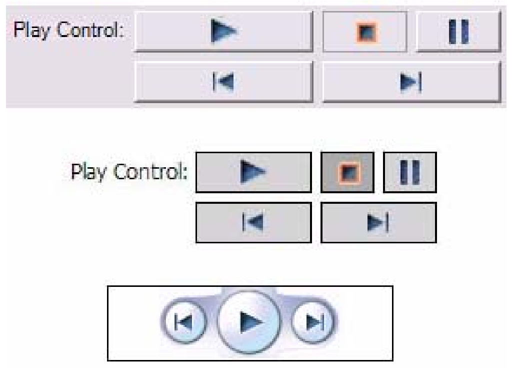

Figure 8. Media controls rendered for a Windows Media Player interface on each of our three platforms. At the top is the desktop, the middle is PocketPC, and the bottom shows Smartphone. The Smartphone control maintains consistency for the user by copying the layout for the Smartphone version of Windows Media Player, the only media player application we could find on that platform. This interface overloads pause and stop with the play button.

cases, special purpose rules are required, such as to always put the play button on the left.

- *Data Translation*, which translates the appliance state values into a format that can be understood by the chosen control(s), and then translates data from the control back into state values after the user has made a change.

The data translation code is usually the most time-consuming and tedious part of a template implementation, because each combination of state variables must be considered. Fortunately, we have found that it is easy to share the data translation code across platforms[1] because many controls use the same data representations. In some cases, it may be possible to share translation code between Smart Templates, such as for the `time-duration` and `time-absolute` templates.

The control choice code can rarely be shared across platforms or templates, because each implementation often includes layout rules that are platform and/or template-specific. Some of the tasks can be abstracted out however, such as extracting particular states from the template's content or certain kinds of analyses like determining if a state is bounded or if a state has a dependency on another state. We have created a framework that is used by all of our templates and reduces the amount of effort required to implement a Smart Template.

FUTURE WORK

In the immediate future we plan to define and implement many new Smart Templates in our system. As we connect our system to more appliances, we expect that we will find many more uses for Smart Templates. An eventual goal of our work is to create a comprehensive list of templates for common appliances and to make the list available to others.

Since Smart Templates are rendered differently on different platforms, it should be possible for a template renderer to take advantage of the special features of its platform. For example, on a phone with a jog-dial, the volume Smart Template could automatically use the jog-dial as its control rather than rendering a slider on the screen. In general, it is a difficult problem to decide how to allocate user interface features to the physical buttons and dials on a device, because the position of the buttons and the functions commonly assigned to the buttons must be taken into account. Smart Templates can help interface generators make these kinds of decisions by providing extra semantic information that is not normally available in the specification language.

As mentioned earlier, we also plan to implement Smart Templates in our speech interface generator. This promises to greatly improve certain aspects of the interfaces, especially for common types such as date and time. The currently speech generator does not speak a time properly (saying *one colon two four*, for 1:24) because it does not know that the value represents a time. With a Smart Template, the speech system will be able to use the extra semantic information to improve its speaking ability. We also hope to explore the use of Smart Templates for providing higher quality speech interactions to certain variables. For example, the `volume` Smart Template could suggest to the speech system that *louder* and *softer* should be used to manipulate a volume state variable, instead of generic commands such as *set volume to 10*. We believe there are many other situations where semantic information can benefit our speech interface generator.

In some cases, an interface generator may desire to render two or more Smart Templates as one element in a user interface. In these cases, it does not necessarily make sense to define another Smart Template that combines the others together. Instead, code could be written to find related instances of Smart Templates in an appliance specification and to render those templates as a combined control. This could be used for rendering mute and volume functions together, or even for combining date and time choices into a single control.

We believe that Smart Templates can also help make our generated interfaces localized for the culture of the user. Specifically, this should be done for the date and time Smart Templates, which should appear differently depending on the user's home locale. The interface generator should be able to localize interfaces transparently by extracting information from the user's device, and this is something we will implement in the future.

Finally, we intend to conduct user studies to compare our generated interfaces to the interfaces developed by manufacturers' for their own appliances. These studies would extend our earlier studies comparing hand-designed user interfaces to manufacturers' interfaces [6] and follow a similar format. Our goal is to show that users' performance with our generated interfaces matches or exceeds their performance with the manufacturers' interfaces.

CONCLUSION

We have described Smart Templates, a technique for improving automatically generated interfaces. This technique is novel because it uses parameterized templates to allow automatic interface generators to create more usable interfaces that are consistent for users. Parameterization of the templates makes appliance specifications easier to create because it does not require the appliance to be implemented in a particular way. Smart Templates can be used by interface generators to render basic elements such as time and more complex structures such as the playback controls for a media player. Automatic interface generators can use these templates to improve their interfaces by using conventional layouts and controls that are consistent with other interfaces on the same device. This technique was applied

[1] It is possible for us to re-use code across platforms because all of our implementations use Microsoft's .NET Compact Framework, which is supported on all of our platforms.

on three different platforms: Microsoft's PocketPC and Smartphone, and also on desktop computers.

ACKNOWLEDGEMENTS
This work was conducted as a part of the Pebbles [4] project, with the help of many other people including Michael Higgins and Joseph Hughes of MAYA Design, Thomas K. Harris, Roni Rosenfeld, Mathilde Pignol, Rajesh Seenichamy, and Stefanie Shriver of Carnegie Mellon University. The speech interface was implemented as a part of the Universal Speech Interfaces project [12]. We would also like to thank Krzysztof Gajos and Desney Tan for comments on an early version of this paper. This work was funded in part by grants from NSF, Microsoft, General Motors, DARPA, and the Pittsburgh Digital Greenhouse, and equipment grants from Mitsubishi Electric Research Laboratories, VividLogic, Lutron, Lantronix, IBM Canada, Symbol Technologies, Hewlett-Packard, and Lucent. 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. Brouwer-Janse, M.D., Bennett, R.W., Endo, T., van Nes, F.L., Strubbe, H.J., and Gentner, D.R. "Interfaces for consumer products: "how to camouflage the computer?"" in *CHI: Human factors in computing systems.* 1992. Monterey, CA: pp. 287-290.
2. 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.
3. 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.
4. Myers, B.A., "Using Hand-Held Devices and PCs Together." *Communications of the ACM*, 2001. **44**(11): pp. 34-41.
5. Nichols, J., Myers, B.A., "Personal Universal Controller Project Home Page," 2003. http://www.cs.cmu.edu/~pebbles/puc/.
6. Nichols, J., Myers, B.A. "Studying The Use Of Handhelds to Control Smart Appliances," in *23rd International Conference on Distributed Computing Systems Workshops.* 2003. Providence: 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.* 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
9. Nylander, S. "Different Approaches to Achieving Device Independence," in *Technical Report TR2003-XX.* 2003. Swedish Institute of Computer Science:
10. Olsen Jr., D.R., Jefferies, S., Nielsen, T., Moyes, W., and Fredrickson, P. "Cross-modal Interaction using Xweb," in *Proceedings UIST'00: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 2000. San Diego, CA: pp. 191-200.
11. 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.
12. Rosenfeld, R., Olsen, D., Rudnicky, A., "Universal Speech Interfaces." *interactions: New Visions of Human-Computer Interaction*, 2001. **VIII**(6): pp. 34-44.
13. Szekely, P., Luo, P., and Neches, R. "Beyond Interface Builders: Model-Based Interface Tools," in *Proceedings of INTERCHI'93: Human Factors in Computing Systems.* The Netherlands: pp. 383-390.
14. 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 Paix
15. Weld, D., Anderson, C., Domingos, P., Etzioni, O., Gajos, K., Lau, T., Wolfman, S. "Automatically Personalizing User Interfaces," in *Eighteenth International Joint Conference On Artificial Intelligence.* 2003. Acapulco, Mexico:
16. 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.