Report on the INCITS/V2 AIAP-URC Standard

Jeffrey Nichols and Brad Myers
Human Computer Interaction Institute
Carnegie Mellon University
{jeffreyn | bam}@cs.cmu.edu
http://www.cs.cmu.edu/~pebbles/puc/
February 9, 2004

Introduction

INCITS/V2, the Information Technology Access Interfaces Technical Committee of the InterNational Committee for Information Technology Standards (INCITS), is about to formally propose a standard for "the discovery, selection, operation, and substitution of user interfaces and options" [6] for everyday appliances. Their approach is similar to that of our *personal universal controller* (PUC) project [9]: each appliance has an abstract specification of its functions and user interfaces for the appliances are presented on a remote control device that users have with them. The remote control device could be a personal digital assistant, (PDAs), mobile phone, or any other computerized device with communication capabilities (e.g. Tablet PC, wristwatch of the future, etc.). The interfaces presented on the remote control device could also be in several modalities, such as graphical or speech.

This report analyzes the V2 specification in detail, drawing on our experiences building the PUC system. We start by examining how the current version of the V2 specification addresses our comments on an earlier version, particularly by analyzing how the current standard meets our eight requirements for automatically generating user interfaces [10]. Then we contrast the current version of the V2 specification with our PUC system, and discuss the advantages and disadvantages of both approaches. The report concludes with an explanation for why we will not be using the V2 standard in our work, even though it provides most of the benefits of our PUC system.

Results of Previous Collaboration

At a V2 meeting in June of 2002, we presented our analysis of an early version of the V2 specification. Based on this analysis, we also wrote a paper describing eight requirements for automatically generating interfaces and analyzed how the PUC, V2 and other automatic generation systems met these requirements [10]. This section applies those eight requirements to the current version of the V2 specification, and looks at how some of our early comments were taken into account.

Requirements for Automatically Generating Interfaces

1. Two-Way Communication

Two-way communication is needed between the appliance and the controller device to allow an abstract appliance specification to be downloaded to the controller and control signals to be sent to the appliance. Two-way communication is also needed to keep state information synchronized between the appliance and the controller. The PUC and all versions of the V2 specification have satisfied this requirement.

2. Simultaneous Multiple Controllers

It is also important that multiple controllers can communicate with the same appliance simultaneously. Users will expect this feature, and it has the added benefit of allowing different interface modalities to be freely mixed together by using several different controller devices in tandem. For example, a user might combine a handheld controller with a headset to create a multi-modal graphical and speech interface. The PUC and all versions of the V2 specification have satisfied this requirement.

Supporting multiple controller devices simultaneously does introduce synchronization and conflict problems that need to be resolved. For example, a synchronization problem can arise when two users are manipulating the same list of information. Suppose one user moves the first item in the list to the end, and a split-second later another user deletes the item that was initially fifth in the list. This might cause the fifth element after the move to be deleted from the list, instead of the desired action (the fourth item after the move should have been deleted). At the moment, the PUC system ignores this kind of problem, though we have designed a list versioning system that may solve this problem if implemented. The V2 standard does not seem to have a system for resolving similar problems, and may require each appliance to develop its own solution. This could add greatly to the complexity of implanting an appliance within the standard.

Conflicts are also a problem when two different people are simultaneously controlling a single appliance. For example, one person might instruct the media player to go to the previous track at the same time as another user instructs the media player to go to the next track. Most conflict problems can be resolved with social solutions rather than technology, but it is important that users have enough information to solve the problem socially. The PUC system also does not address this problem, but the V2 standard might consider mechanisms for determining who has recently manipulated a state variable or command.

3. No Specific Layout Information

The appliance specification should include information about the functions of the appliance, but it should not contain any information about how those functions should be presented to the user. This includes the controls that should be used to manipulate a function, where the control should be placed on a screen, etc. With the PUC, we are purposely trying to take an extreme view of not allowing this kind of information to be included anywhere. Our challenge is to see how well we can incorporate design knowledge into our automatic interface generators, and to create high-quality interfaces automatically without needing a designer to tweak the result.

The new version of the V2 standard takes a less extreme approach by making pre-designed interfaces available but also requiring an abstract description without any kind of layout information. This makes sense for a system that must be adopted by real corporations that will want to ensure that the remote control interface that most people see contains the same branding as the actual appliance. It also allows corporations who value good interface design to ensure the quality of the interfaces that most users see.

We are a little worried that being able to include hand-designed user interfaces in a V2 specification will decrease the motivation of corporations to include complete or valid abstract descriptions of the user interface. For example, the HAVi standard supports two levels of interface: the level one Data-Driven Interface (DDI) that is somewhat abstract, and the level two interface which consists of a Java applet. The HAVi digital VCR that we have supports limited functionality through its DDI description (play, stop, pause, etc.) but the level two Java applet has controls for the counter, playback speed, etc. As we understand the V2 standard, it would be possible for a corporation to write a limited user interface socket and standard presentation template, but include a much more complete hand-designed interface for platforms the corporation is interested in. Presumably such a corporation could design their device to respond to communication regarding state variables and commands that are not included in the user interface socket. End-users could still reverse engineer the product and produce a better standard template for it, but this would only happen for the most common appliances. V2 will need to have enforcement measures similar to the UPnP licensing process to ensure that each appliance is fully specified. Without such measures, the standard is unlikely to achieve its goal of making appliances accessible to all kinds of users.

4. Hierarchical Grouping

A fundamental requirement of any user interface is good organization, because users must be able to intuitively find a particular function. An appliance specification can easily define organization using a tree to group similar functions. This makes automatically generating interfaces easier, because most concrete user interfaces can be described as a tree. The utility of trees for grouping seems to be universally accepted; the PUC, all versions of the V2 standard, and many other systems use trees for grouping functions.

5. Actions as State Variables and Commands

Each function of the appliance must be represented in the appliance specification. We found, as have many others, that state variables and commands are a succinct way to represent the manipulable elements of an appliance. There are at least two ways state variables and commands can be used however. The method used by systems like UPnP is to specify commands for every allowable manipulation of a state variable. In this case, the variable is only used for data storage and its value cannot be set directly. This means that a specification for a radio might include a station variable, and also tune up, tune down, seek up, and seek down commands associated with the variable. The PUC system infers as many functions from the state variable as possible, but still uses commands for those functions that cannot be inferred such as seek up and seek down

The new version of the V2 standard [5] is more like the PUC system. State variables have types and some commands are inferred from the type of the variable. The V2 standard also uses two other primitive elements beyond state variables and commands to represent the functions of an appliance: statics and notifications. A static is an element that represents fixed or constant information in the interface description. The PUC system has a similar construct called an “explanation,” which seems to have the same behavior as a V2 static, though explanations are implicitly limited to having a string type.

The other primitive element is the notification, which is used to explicitly describe alert messages that may be used by the appliance. This is an interesting idea, as it gives the interface generator information about when the appliance may want to interrupt the user, and also information about what actions may result in an error. In the PUC system, it is possible for an appliance to send an alert message to a controller device, but this feature is implemented strictly through the communication protocol. The PUC specification language does not define when an alert message will be sent or any actions that should be taken following an acknowledgement of the alert. No message is sent back to the appliance when an alert dialog has been acknowledged by the user.

Notifications in the V2 standard are somewhat complicated, and support many of the features that the PUC system's alert messages do not. Notifications have a category attribute that specifies whether they are information, an alert message, or the result of an error. The V2 standard can require explicit user acknowledgement of a notification, and a special *acknowledge* dependency must be satisfied before the user can successfully acknowledge a notification. There is also an explicit process for dealing with multiple notifications simultaneously. In this case, notifications become stacked with only the newest notification visible to the user.

One worry that we have about notifications is the possibility for creating deadlocks that the user cannot escape from. Given the current specification it is possible for one notification to require that an earlier one has been acknowledged. If this does not happen for some reason, the current standard would not allow the user to go back and acknowledge the earlier notification. An improperly written acknowledge dependency could also create a deadlock situation, by creating a dependency on a state variable that the user cannot modify. Deadlocks can be prevented by careful authoring, but should authors need to be thinking about deadlocks when writing an abstract specification?

6. Dependency Information

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

In an early version of the V2 standard, dependency information was specified with ECMAScript expressions in structures called "guards." Guards were sufficient for determining whether a function was active, but it was difficult to do any analysis of the dependency information because it was written in a scripting language. This would preclude the dependency information from being used for graphical layout, speech disambiguation, etc. The PUC system avoids this problem by specifying dependency information as a concise set of five relations joined by logical AND and OR operations.

The new version of the V2 standard [5] uses XPath [1] expressions to specify dependency information. Unlike ECMAScript, XPath expressions have more constraints and are thus easier to analyze. XPath is still more flexible than the language we use for the PUC system however, and it is not clear how much more difficult this would make analysis. For example, XPath supports mathematical operations and a number of built-in method calls, including some for manipulating strings, none of which are supported by the PUC. It would be nice to see some support in the documentation that showed that XPath could be easily analyzed and knowledge from this analysis could be applied by an interface generator. If this is not true, then perhaps XPath is not a good choice for specifying dependency information.

The new version of the V2 standard also defines several different types of dependencies, including read and write dependencies for state variables, read dependencies for commands and statics, execute dependencies for commands, and acknowledge dependencies for notifications. The PUC system only defines one kind of dependency which can be applied to any appliance object. These dependencies specify when the object is active, which most closely matches V2's write dependencies for state variables and the read dependencies for statics, and the execute dependencies for commands. One important difference is that no dependency in the PUC system that places a restriction on whether a control will be visible, which is the function of the read dependency in the V2 standard. Read dependencies for commands are almost layout hints, which suggests that perhaps they should be defined in the standard presentation template [3] instead of the user interface socket. Read dependencies are a more interesting idea for variables, but I wonder whether the read dependency result will always be accurate. In the PUC system it is possible for variables to have an undefined value. This is always the case before a value has been downloaded to the controller device, and it can be true in other situations as well. Given the description in the standard, it seems that the read dependency must have some way of describing this situation, but the standard does not indicate how this would be done.

I also think that the definition of a read dependency for a static should not be shared with a state variable. Since the value of a static never changes, it does not make sense to define its read dependency as a formula describing when the variable will be defined. Either the value of a read dependency for a static should always be true, or be defined differently. It might make sense to define it in the same way as for a command: a static is presentable whenever its read dependency formula is true.

7. Sufficient Labels

Good labels are an important part of creating a high-quality user interface. Labels are even more important for speech user interfaces, because no graphical hints are available to assist the user's understanding of the interface. To give flexibility to the user interface generator, a label in an appliance specification should not be a single text string but instead a collection of text strings, pronunciation keys, and text-to-speech recordings. Multiple text strings give a graphical interface the flexibility to select the label with the most information that can fit in the allocated space.

An early version of the V2 standard used only a single string for each label. The new version of the standard goes far beyond this, by completely separating all labels into a resource description file. A resource is defined as “an identifiable object of a user interface that is used as an entity in the construction of a concrete user interface” [4]. In most cases, a resource can be thought of as a label in the PUC specification language. The V2 standard allows each resource to have multiple different descriptions, similar to the way the PUC allows a collection of multiple items for each label. The key difference between the V2 standard and the PUC approach is V2’s separation of the resource descriptions from the rest of the appliance specification. This allows a V2 appliance description to be easily internationalized by picking a resource description file for the appropriate country. The PUC system does not currently support internationalization, but if we choose to add that feature then we will probably use an approach similar to that of the V2 standard.

8. Shared High-Level Semantic Knowledge

We must concede that it is impossible to encode all the information into an appliance specification that a human would use to design an interface. In addition to functional information, a human designer will also use his knowledge of conventions when creating an interface. There are many such conventions, such as the arrangement of buttons on a telephone number pad or the country-specific format for specifying dates. Defining every such convention in an appliance specification is simply not possible.

We are solving this problem with a new feature called *Smart Templates*, which extend the primitive types in the PUC appliance specification language with high-level semantic information. For example, the `media-controls` template defines that a state variable with an enumerated type containing members labeled *Play* and *Stop,* which control the playback of some media. PUC interface generators can use the information added by a Smart Template to apply design conventions and make interfaces more usable. If an interface generator does not recognize a template however, a user interface can still be created because Smart Templates are constructed from the primitive elements of our specification language.

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 1). 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. one variable for hours and another for minutes).

The V2 standard takes a different approach by allowing hand-designed interfaces to be included in the appliance specification. Designers can explicitly include high-level domain-specific design conventions, like those that would be handled by Smart Templates, in the hand-designed interfaces. High-level information is not available in the standard presentation template however, which means that automatic interface generators will likely create lower quality interfaces from the V2 standard than they might have. It is hard to recommend that V2 include a feature like Smart Templates however, because standardizing design conventions seems to be a large project by itself and is on the fringe of what V2 is trying to accomplish.

Other Early Comments

Objective vs. Subjective Information

While designing the PUC specification language, we found that the information contained in each appliance description was either objective or subjective. Objective information can be extracted by manipulating the appliance and observing its output, whereas subjective information is biased by the designer of the specification language. In our language, we consider dependency information to be objective and the group tree to

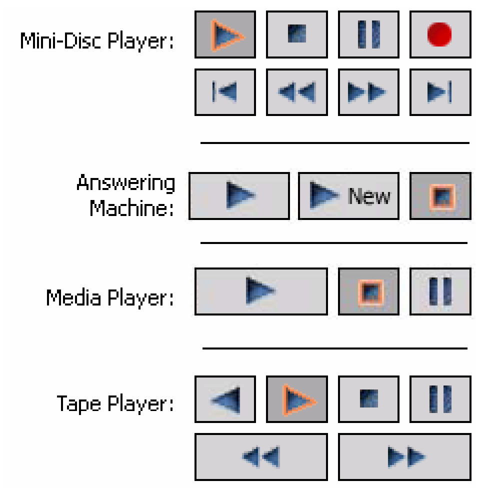

Figure 1. Different arrangements of media playback controls automatically generated for several different appliances from a single Smart Template (`media-controls`).

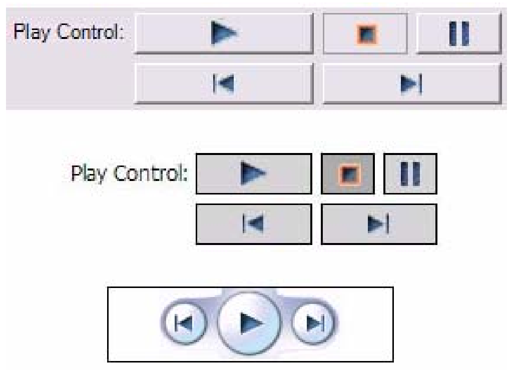

Figure 2. Media controls rendered for a Windows Media Player interface on each of our three platforms. At the top is the desktop, the middle is PocketPC, and the bottom shows Smartphone. The Smartphone control maintains consistency for the user by copying the layout for the Smartphone version of Windows Media Player, the only media player application we could fine on that platform. This interface overloads pause and stop with the play button.

be subjective. Our interface generators combine these pieces of information to get an accurate picture of the appliance.

Why is the distinction between objective and subjective information important? We have found that it is important to find a good balance between the amount of objective and subjective information that is included in a specification language. If too much objective information is included, then appliance descriptions become long, overly detailed, and difficult to author. If too much subjective information is included, then the designer of the specification may have too much control over the final result of any interface generator. Subjective information that adds little to a specification may even be harmful by taking design control away from the interface generator.

The early version of the V2 standard included a lot of subjective information that we believed would limit the quality of the interface that could be generated. The “*include* guards” from that version seemed particularly subjective and unnecessary. The new standard has read dependencies, which seem very similar to include guards. Although there may occasionally be benefits to this feature, there are clear liabilities as well. If a read dependency forces a control to be removed from the screen, how does an interface generator prevent this from being disruptive to users? If a large number of controls have the same read dependency then the interface generator may be able to do something that makes sense, but if only one control must be hidden then the number of design options becomes much smaller. Any information that specifies when a control can be shown seems more like a layout hint than a functional description, and we recommend that V2 be careful when including such features in their standard.

Additional Analysis

This section analyzes features that are new to the current version of the V2 standard.

Separation of Specification

The V2 standard separates the specification of an appliance into four parts: the target properties sheet, user interface socket, presentation templates, and resource descriptions. The target properties sheet describes properties of the appliance that are downloaded to the controller device when the appliance is discovered. The remaining files are downloaded when the appliance is opened, or as needed. The user interface socket describes the functions of the appliance, the presentation templates describe how user interfaces might look, and resource descriptions give labeling information for use in the concrete interface. This approach is very different from the PUC specification language, which uses only one file to describe the functions of an appliance. The PUC also does not have a feature analogous to the target properties sheet.

Separating the specification into several different pieces is a good idea, especially separating resources from the functional description. This makes it much easier for devices described using the V2 standard to be internationalized, even by a group not related to the manufacturer of the appliance. Separate presentation templates allow manufacturers to include hand-designed interfaces for common platforms, which could also be beneficial for users who have a common device (discussed further in the “3. No Specific Layout Information” section above). It is likely the PUC system will adopt a similar separation in the future if it suits our research goals.

One interesting feature of the V2 standard is the standard presentation template, which contains abstract information about how a user interface should be presented. This is different from other presentation templates, which are full descriptions of concrete interfaces. The PUC specification language could be seen as a combination of V2's user interface socket and standard presentation template. We have found it beneficial to keep these two pieces of information together in one file, because it keeps our descriptions concise and allows us to restrict the number of different ways that an appliance can be described. For example, the interactors described in V2's standard presentation template can usually be inferred from the type of the state variable that the interactor is associated with. The PUC specification language also uses the group tree to assist in the description of complex types. Groups are used to describe records, arbitrary combinations of fields with different types that are common in many programming language (called structs in the C language). As we understand it, the V2 standard could describe a record in two ways. Either multiple states could be created for each field in the UI socket and grouped in the standard presentation template, or a single state could be created with a complex record type. Ensuring that there is only one way to specify any particular feature has at least two benefits: the interface generator can easily recognize features in the specification and appliance descriptions are easier to author.

Target-URC Network

The V2 standard also separates concerns about the network from the rest of the specification. The standard suggests UPnP and JINI as possible network technologies for AIAP to run over, though other protocols could also be supported. This makes it easier for manufacturers to support the standard in their appliances.

Unfortunately, the standard does not make it clear how exactly AIAP will operate over technologies like UPnP or JINI. Each of these technologies is an application layer protocol, and does not have built-in operations that support the functionality that AIAP needs. In order for AIAP to use any particular protocol, it seems that a separate standard document will be required that defines how the networking technology is used by AIAP. No mention of this is made in the standard document.

For example, UPnP supports a description language that has state variables and parameterized function calls. Unlike AIAP, state variables can only be changed by invoking the appropriate command with the appropriate parameters. A standard set of these methods will need to be created in every UPnP device in order for AIAP to operate successfully over UPnP. Without some kind of pre-arranged standard, there is no guarantee that a URC will be able to control an appliance that supports AIAP over UPnP.

Interactors

The standard presentation template specifies a set of interactors that describe how state variables and commands should be manipulated in the user interface generator. The use of an interactor-like construct is strongly supported by research systems, though we have chosen not to use them for the PUC system. V2's interactors are much like the abstract interaction objects (AIOs) that Vanderdonckt proposed for his TRIDENT system [13, 14], and similar to the interactors Myers used in Garnet [7] and Amulet [8].

The PUC system does not currently support specifying an interactor for each state variable or command. Originally we chose not to include interactors because we believed that including interaction information would compromise the abstractness of our language. At that time we were also unaware of the AIO concept, which has proven to be portable across many different interface modalities [12]. We have not changed the PUC language to include AIOs for each state and command because the current system seems to work, and to keep our description files as short as possible. We have found interactor-like information to be useful however. For example, our new Smart Templates [11] apply abstract interaction information to groups of states and commands.

The nine interactors that V2 has chosen cover most common interactions and are based on the XForms standard [2]. XForms was designed to streamline the process of authoring web forms and fetching information from the user, in part by separating presentation from content. This is similar to what V2 and the PUC are trying to accomplish for remote control user interfaces. Basing the V2 standard on XForms may also benefit authors who have previous experience with XForms and can apply knowledge they already have. Our only criticism of this approach is the XForms interactors, which seem to include a few arbitrary design decisions. One question is whether to create a new interactor or parameterize an existing one. It seems that the `select` and `select1` interactors could be reduced to a single interactor with one parameter for the number of selections. The `input`, `secret`, and `textarea` interactors could also be combined into a single interactor through parameterization. Parameterization makes even more sense because the interactors that could be combined share many of the same attributes. Another question is is whether to infer some design decisions from the underlying state variables. For example, the `range` interactor seems to be the same as the `select1` interactor with specific requirements on the underlying type. Why not remove the `range` interactor and infer it from type information?

Adoption

The current version of the V2 standard is very similar to the PUC specification language, and seems to be about as descriptive. We believe that we could adapt our interface generator technology to the V2 standard without diminishing the quality of the resulting user interfaces, but we are leery of adopting the standard though for a number of reasons:

- Adopting the V2 standard will be a significant distraction from our research goals. We estimate that switching just the interface generators to V2 would be at least a month of effort, and that assumes that code libraries (such as parsers for V2 formats) are available on our platforms (Microsoft's .NET Compact Framework) to ease the transition. We would rather focus our efforts on future research directions, such as ensuring interface consistency and generating multi-appliance interfaces.

- It seems like we would only be able to interface to whatever targets V2 already has working. Porting our real and simulated targets over to the V2 protocol would also take many months of effort. Currently, we have developed about 15 targets through several man-years of effort.

- There is an added risk of going to a new code base, and in particular, of basing the success of our research on the success of V2's. Xerox PARC called this "Error 33: Predicating one research effort upon the success of another" -- http://info.astrian.net/jargon/terms/e/error_33.html
- From our initial analysis, it looks like most of the things that Jeff is proposing to do next for his thesis would require modification of the V2 standard. We doubt that Smart Templates [11] or multi-appliance interface generation could be implemented with the current V2 standard.

Although we do not plan to adopt the standard for our continuing research, it may be possible to write "adaptors" that bridge between the V2 standard and the PUC system. We have talked for several years about creating general adaptors to protocols such as UPnP and HAVi, but the V2 standard is first one that is descriptive enough to make the idea viable. Such as adaptor would need to automatically translate PUC specifications to V2 specifications and vice versa, and also translate between the PUC communication protocol and one or more of the V2 network links. We estimate that this work could be done independently of the current PUC research, though it would take at least several months to accomplish and require a new student to do the work. Some things would be lost in translation, since the two specifications do contain some different information. For example, a PUC specification created from a V2 specification could only include a few Smart Templates that can be recognized directly (date, time, and possibly others). A V2 specification created from a PUC specification would have default read dependencies for state variables, no notifications, etc.

Conclusion

This report analyzes the V2 standard in terms of our analysis of an earlier draft of the standard, our eight requirements for automatically generating user interfaces, and our experiences developing the PUC system. We think there is great promise for a standard such as V2, and hope that it becomes universally adopted. We look forward to collaborating with V2 in the future and attending the AIAP SIG at CHI 2004.

References

1. Clark, J., DeRose, S., "XML Path Language (XPath) Version 1.0," 1999. World Wide Web Consortium: http://www.w3.org/TR/1999/REC-xpath-19991116.
2. Dubinko, M., Klotz Jr., L.L., Merrick, R., Raman, T.V., "XForms 1.0," 2003. World Wide Web Consortium: http://www.w3.org/TR/2003/REC-xforms-20031014/.
3. INCITS/V2, "Standard Presentation Templates," in *Alternate Interface Access Protocol*2003.
4. INCITS/V2, "Standard Resource Descriptions," in *Alternate Interface Access Protocol*2003.
5. INCITS/V2, "Standard User Interface Socket Description," in *Alternate Interface Access Protocol*2003. Washington D.C. pp.
6. INCITS/V2, "Universal Remote Console Specification," in *Alternate Interface Access Protocol*2003. Washington D.C. pp.

7. Myers, B.A., “A New Model for Handling Input.” *ACM Transactions on Information Systems*, 1990. **8**(3): pp. 289-320.
8. Myers, B.A., McDaniel, R.G., Miller, R.C., Ferrency, A., Faulring, A., Kyle, B.D., Mickish, A., Klimovitski, A., and Doane, P., “The Amulet Environment: New Models for Effective User Interface Software Development.” *IEEE Transactions on Software Engineering*, 1997. **23**(6): pp. 347-365. June.
9. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Pignol, M. “Generating Remote Control Interfaces for Complex Appliances,” in *UIST 2002.* 2002. Paris, France: pp. 161-170. http://www.cs.cmu.edu/~pebbles/papers/PebblesPUCuist.pdf.
10. 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:
11. Nichols, J., Myers, B.A., Litwack, K. “Improving Automatic Interface Generation with Smart Templates,” in *Intelligent User Interfaces.* 2004. Funchal, Portugal: pp. 286-288.
12. 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.
13. 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:
14. Vanderdonckt, J., “Advice-Giving Systems for Selecting Interaction Objects.” *User Interfaces to Data Intensive Systems*, 1999. pp. 152-157.