Chapter 22

BUILDING MULTI-PLATFORM USER INTERFACES WITH UIML

Mir Farooq Ali[1], Manuel A. Pérez-Quiñones[1], Marc Abrams[2], Eric Shell[1]
[1]*Department of Computer Science, Virginia Tech,*
660 McBryde Hall, Blacksburg, VA 24061, USA
Email: { mfali, perez, ershell }@vt.edu

[2]*Harmonia, Inc., PO Box 11282,*
Blacksburg, VA 24062, USA
Email: marc@harmonia.com

Abstract There has been a widespread emergence of computing devices in the past few years that go beyond the capabilities of traditional desktop computers. However, users want to use the same kinds of applications and access the same data and information on these appliances that they can access on their desktop computers. The user interfaces for these platforms go beyond the traditional interaction metaphors. It is a challenge to build User Interfaces (UIs) for these devices of differing capabilities that allow the end users to perform the same kinds of tasks. The User Interface Markup Language (UIML) is an XML-based language that allows the canonical description of UIs for different platforms. We describe the key aspects of our approach that makes UIML successful in building multi-platform UIs, namely the division in the representation of a UI, the use of a generic vocabulary, a process of transformations and an integrated development environment specifically designed for transformation-based UI development. Finally we describe the initial details of a multi-step usability engineering process for building multi-platform UI using UIML.

Keywords: Multi-Platform User Interfaces, Transformations, UIML, Physical Model, Logical Model, Usability Engineering.

1. INTRODUCTION

Advances in electronics, communications, and the fast growth of the Internet have made the use of a wide variety of computing devices an everyday occurrence. These computing devices have different interaction styles, input/output techniques, modalities, characteristics, and contexts of use. Furthermore, users expect to access their data and run the same application from any of these devices. Two of the problems we encountered in our own work [2] in building UIs for different platforms were the different layout features and screen sizes associated with each platform and device. Dan Olsen [13], Peter Johnson [9], and Stephen Brewster, et al. [4] all talk about problems in interaction due to the diversity of interactive platforms, devices, network services and applications. They also talk about the problems associated with the small screen size of hand-held devices. In comparison to desktop computers, hand-held devices will always suffer from a lack of screen real estate, so new metaphors of interaction have to be devised for such devices. It is difficult to develop a multi-platform UI without duplicating development effort. Developers now face the daunting task to build UIs that must work across multiple devices. There have been some approaches towards solving this problem of multi-platform UI development including XWeb [14]. Building "plastic interfaces" [5, 20] is one such method in which the UIs are designed to "withstand variations of context of use while preserving usability".

We developed the User Interface Markup Language (UIML) to address the need for a uniform language for building multi-platform applications. We discuss our research in creating such language, some of the support tools available, and describe our approach towards creating a new user interface design methodology to build multi-platform user interfaces.

UIML is a single language for building user interfaces for any device. UIML emphasizes the separation of concerns of an interactive application in such a way that moving one program from one platform to another (see our definition of platform later) is relatively easy. Furthermore, because it is based on XML, it is easy to write transformations using a tool like XSLT [6] that take the language from one abstract representation to a more concrete representation. The tools built around UIML extend the language with the use of transformations that afford the developer the creation of user interfaces with a single language that executes on multiple platforms.

The approach taken with UIML is that of building an application using a *generic vocabulary* that could then be *rendered* for multiple platforms. Using a generic vocabulary for desktop applications, for example, the developer can write a program in UIML only once and have it be rendered for Java or HTML. And, within HTML, it can be rendered to different

versions of HTML and/or browsers. We present some details of the generic vocabulary and the advantages of this approach in this paper.

Based on our experience with UIML, we have learned that one generic vocabulary is not sufficient to build interfaces for widely varying platforms, such as VoiceXML, handhelds (e.g. Palm PDAs) and desktops. In this paper, we describe the preliminary results of a multi-step process of building multi-platform user interfaces. It involves programming in UIML at different levels of abstractions with the support of transformations that move the developer's work from one representation to the next. This paper presents our initial results on this area.

Some terminology that is used in the paper is defined next. A *device* is a physical object with which an end-user interacts using a user interface, such as a Personal Computer (PC), a hand-held computer (like Palm), a cell-phone, an ordinary desktop telephone or a pager. A *toolkit* is the library or markup language used by the application program to build its UI. Unlike its traditional definition, the term *toolkit* implies both markup languages and more traditional APIs for imperative or object-oriented languages. A *Vocabulary* is the set of names, properties and associated behaviour for UI elements available in any toolkit. A *platform* is a combination of device, operating system and toolkit. An example of a platform is a PC running Windows 2000 on which applications use the Java Swing toolkit. This definition has to be expanded in the case of HTML to include the version of HTML and the particular web browser being used. For example, Internet Explorer 5.5 running on a PC with Windows NT would be a different platform than Netscape Communicator 6.0 running on the same PC with the same OS since they implement various features of HTML differently. A related term we use in this paper is *family*, which indicates a group of platforms that have similar layout features. A *Generic Vocabulary* is the vocabulary shared by all platforms of a family.

2. RELATED WORK

Transcoding [3, 8] is a technique used in the World Wide Web for adaptively converting web-content i.e. HTML page, for the increasingly diverse kinds of devices that are being used these days to access web pages. This process also requires some kind of transformation that occurs on the HTML web page to convert it to the desired format. In its simplest form, semantic meaning is inferred from the structure of the web page and the page is transformed using this semantic information. A more sophisticated version of transcoding annotates the structural elements of the web page and the transformation occurs based on these annotations. The transcoding approach

is limited in the amount of semantic inferencing that can be made automatically from structural elements of web pages and because annotations are done by hand and are often tied to the structure of existing web pages. If these pages are modified, the annotations have to be modified.

It is useful to revisit some of the concepts behind model-based UI development tools since these concepts have to be utilized for generating multi-platform UIs [12]. Model-based user interface development tools use high-level specifications of different aspects of the UI and automatically generate some parts or the complete UI [11]. One of the central ideas behind model-based tools is to achieve balance between the detailed control of the design of the UI and automation. Many model-based tools built in the late 80s and early 90s include UIDE [18], HUMANOID [10], MASTERMIND [19], ITS [21], Mecano [17], and Mobi-D [16].

The central component in a model-based tool is an abstract model representing the UI. Different types of models have been used in different systems including task models, dialogue models, domain models, and application models. All these models represent the UI at a higher level of abstraction than what is possible with a more concrete representation. The UI developer builds these models that are transformed either automatically or semi-automatically to generate the final UI. The generation of the interface was often done via an automated tool that included a set of rules for generation of the interface components and the interface style.

Although model-based tools provide many advantages over other user interface development tools, one of the main drawbacks of these systems was that the automatically generated interfaces were not of very good quality. One more limitation of some of the earlier systems was the lack of developer control over the process of UI generation. Having more developer-control over the UI development process and having more usable models could rectify some of the deficiencies of the model-based approaches.

3. UIML, GENERIC VOCABULARY AND TRANSFORMATIONS

UIML is a declarative XML-based language that can be used to define user interfaces. More details about the language itself can be found in other references [1, 15]. The mechanism that is currently employed for creating UIs with UIML is to render a UIML document created by a UI developer using a platform-specific vocabulary for the target platform. These renderers can be downloaded from http://www.harmonia.com. Currently, there are platform-specific renderers available for UIML for a number of different platforms. These include Java, HTML, WML, and VoiceXML. Each of these

renderers has a platform-specific vocabulary associated with it to describe the UI elements, their behavior and layout. The formal definition of each vocabulary is available at http://www.uiml.org/toolkits. Table 1 shows five out of eleven vocabularies publicly available at the time of publication.

Table 1: UIML Vocabularies available from http://www.uiml.org/toolkits as of August 2001

Generic Vocabulary Available for Platform
W3C's HTML v4.01 with the frameset DTD and CSS Level 1
Java™ 2 SDK (J2SE) v1.3, specifying AWT and Swing toolkits
A single, generic vocabulary for creating Java *and* HTML user interfaces
VoiceXML Forum's VoiceXML v1.0
WAP Forum's Wireless Markup Language (WML) v1.3

The platform-specific vocabulary for Java, HTML, WML, and VoiceXML uses AWT/Swing, HTML, WML, and VoiceXML tags as UIML part names, respectively. This enables the UIML author to create a UI in UIML that is equivalent to a UI that is possible in Java, HTML, WML, and VoiceXML. However, the platform-specific vocabularies are not suitable for a UIML documents to be mapped to multiple target platforms. For this a *generic* vocabulary is needed. To date, one generic vocabulary has been defined, *GenericJH*, which maps to both Java Swing and HTML 4.0. The next section describes how a generic vocabulary is used with UIML

From Section 1, we recall that the definition of family refers to multiple platforms that share common layout capabilities. Different platforms within a family often differ on the toolkit used to build the interface. Consider for example, a Windows OS machine capable of displaying HTML using some browser and capable also of running Java applications. Both of these use different toolkits, thus making it impossible to write an application in one and have it execute in the other, even though they both run on the same hardware device using the same operating system. For these particular cases, we have built into UIML support for generic vocabularies. A generic vocabulary of UI elements for UIML, can describe any UI for any platform within its family. The vocabulary has two objectives: first, to be powerful enough to accommodate a family of devices, and second, to be generic enough to be used without having expertise in all the various platforms and toolkits within the family.

We have identified and defined a set of generic UI elements including their properties and events for desktop applications. Ali, *et al.* [2] provides a more detailed description of this generic vocabulary. Table 2 shows some of the part classes in this vocabulary for the desktop family that includes HTML 4 and Java Swing.

Table 2: Example of a Generic Vocabulary

Generic Part	**UIML Class Name**	**Generic Part**	**UIML Class Name**
Generic top container	G:TopContainer	Generic Label	G:Label
Generic area	G:Area	Generic Button	G:Button
Generic Internal	G:InternalFrame	Generic Icon	G:Icon
Generic Menu Item	G:Menu	Generic Radio Button	G:RadioButton
Generic Menubar	G:MenuBar	Generic File Chooser	G:FileChooser

3.1 Transformations

A generic vocabulary is useless without a corresponding set of transformations that convert the user interface from the generic terms to a particular toolkit. These transformations are basically a conversion from generic UIML to platform-specific UIML. Both of these representations are be represented as trees since they are XML-based. The platform specific UIML is then rendered using an existing UIML renderer. There are several types of transformations that are performed:

- Map a generic class name to one or more parts in the target platform. For example, a G:TopContainer is mapped to the following sequence of parts in HTML:

```
<html>
  <head>...
    <title>...
    <base>...
    <style>...
    <link>...
    <meta>...
  <body>...
```

  In contrast, G:TopContainer is mapped simply to one part, JFrame, in Java.
- Map properties of the generic part to the proper platform-specific part.
- Map generic events to the proper platform-specific events.

In order to allow a UI designer to fine-tune the UI to a particular platform, the generic vocabulary contains platform-specific properties. These are used when one platform has a property that has no equivalent on another platform. In the generic vocabulary, these property names are prefixed by J: or H: for mapping to Java or HTML only. The transform engine automatically identifies which target part to associate the property with, in the event that a generic part (e.g., G:TopContainer) maps to several parts (e.g., 7 parts for HTML). This is also done for events that are specific to one platform. In this way, the generic vocabulary is not a lowest-common-denominator approach. The generic UIML file would then contain three <style> elements. One is for cross-platform style, one for HTML, and one for Java UIs:

```
<uiml>
...
 <style id="allPlatforms">
   <property id="g:title">My User Interface</property>
 </style>
 <style id="onlyHTML" source="allPlatforms">
   <property id="h:link-color">red</property>
 </style>
 <style id="onlyJava" source="allPlatforms">
   <property id="j:resizable">red</property>
 </style>
</uiml>
```

In the example above, both a web browser and a Java frame have a title, which is "`My User Interface`". However, only web browsers can have the color of their links set, so property h:link-color is used only for HTML UIs. Similarly, only Java UIs can make themselves non-resizable, so the j:resizable property applies only to Java UIs. When the UI is rendered, the renderer chooses exactly one <style> element. For example, an HTML UI would use *onlyHTML*. This <style> element specifies in its *source* attribute the name of the common, shared *allPlatforms* style, so the *allPlatforms* style is shared by both the HTML and Java style elements.

Figure 1 illustrates two interfaces generated from generic UIML for Java Swing and HTML through a process of transformation.

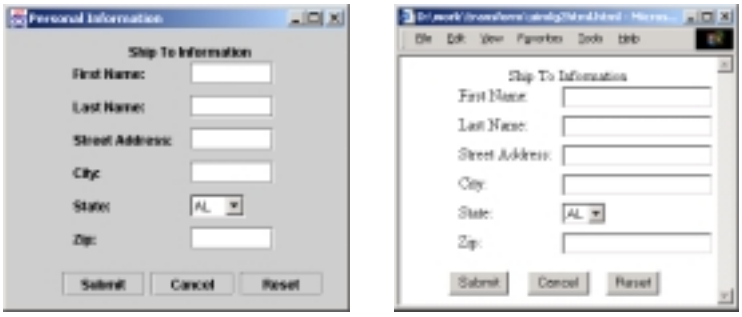

Figure 1: Screen-shots of a sample form in Java Swing (left) and HTML (right).

Figure 2 illustrates a complex application for Java Swing. For economy of space, the corresponding HTML interface is not shown.

4. TRANSFORMATION-BASED DEVELOPMENT

A transformation-based UI development places the developer in unfamiliar territory. Developers are accustomed to having total control over the language and the specification of the user interface elements. A transformation-based environment asks the developer to provide a high-level

description of the interface and to "trust" the result. This is one of the common limitations of code-generators and model-based UI systems. To address this limitation, we have developed a Transformation-based Integrated Development Environment (TIDE) at Virginia Tech for UIML. In TIDE, the developer writes UIML code and the IDE generates the interface, as expected. However, the relationship between UIML code and its resulting interface component is explicitly shown. This section briefly shows how TIDE works.

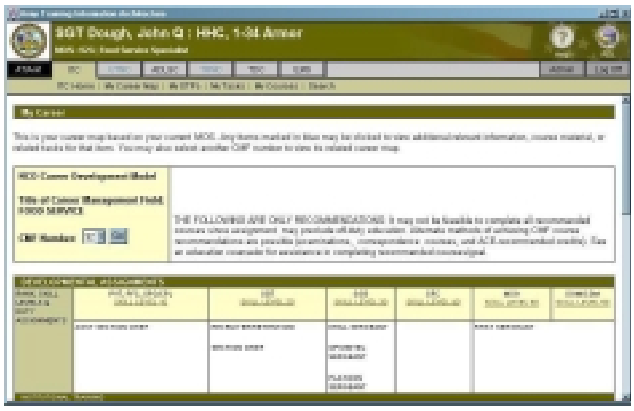

Figure 2: Screen-shots of a complex application for Java built with UIML

The TIDE application was built on the idea that developers creating an interface in an abstract language, such as UIML, which gets translated into one or more specific languages, undergo a process of trial and error. The developer builds what he or she thinks is appropriate in UIML, renders to the desired language(s), and then makes changes as appropriate. As an environment designed to help support this process, this application shows the developer three things: the original, abstract interface (in source code form), the resultant interface after rendering, and the relationship between elements in the two. Figure 3 shows two pictures of the TIDE environment.

The application uses Harmonia's LiquidUI product suite, version 1.0c. Harmonia's product was not changed in any way, and was used by the application to render from the original UIML to Java. The developer may open and close files, view the original UIML source code as plain text or as a tree, and make changes from the tree view. The developer can also re-render at any time (by pressing the red arrow in the center of the window).

The relationship between UIML code and the generated interface is made explicit as shown in Figure 3. The developer may click on a node in the UIML tree view and the corresponding element on the graphical user

interface is highlighted on the right side. The same is true if the developer clicks on a component of the graphical user interface. In the right hand side of Figure 3, the developer has clicked on the cancel button (middle of the three buttons) and the corresponding code is highlighted on the UIML tree-view.

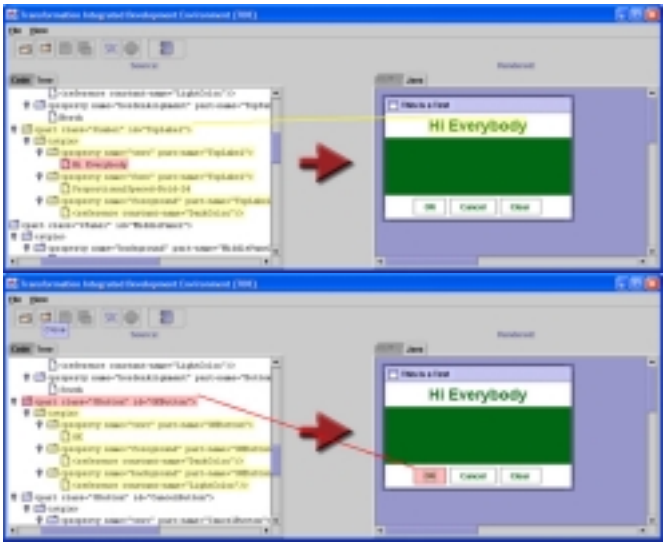

Figure 3: Selection in UIML Code

TIDE makes it very easy to explore the different UIML elements and the effect they have on the generation of the interface. A UIML element's property (e.g. the color of a button) can be edited in place within the tree view.

5. FRAMEWORK FOR MULTI-PLATFORM UI DEVELOPMENT

UIML and the tools presented above are not sufficient to properly address the problem of interface development for multiple platforms. Therefore, we are adding a more abstract layer to support development of interfaces for a wide variety of platforms. This section describes preliminary work on how this higher-abstraction layer would work with UIML.

The concept of building multi-platform UIs is relatively new. To envision the development process, we followed an existing approach from the usability engineering (UE) literature. One such approach [7] identifies three different phases in the UI development process: *interaction design*, *interaction software design* and *interaction software implementation*.

Interaction design is the phase of the usability engineering cycle in which the "look and feel" and behaviour of an UI is designed. In current Usability Engineering practices, this phase is highly platform-specific. Once the interaction design is done, the interaction software is designed, which involves making decisions about the UI toolkit(s), widgets, etc. Once this design is done, the software is implemented Unfortunately, this traditional view of interaction design is highly platform-specific and works well for a single platform. The stage of interaction design has to be further split into two distinct phases for multiple platforms: a platform-independent interaction design and platform-dependent interaction design. This phase, illustrated in Figure 4, leads to different platform-specific interaction software designs, which in turn lead to platform-specific UIs.

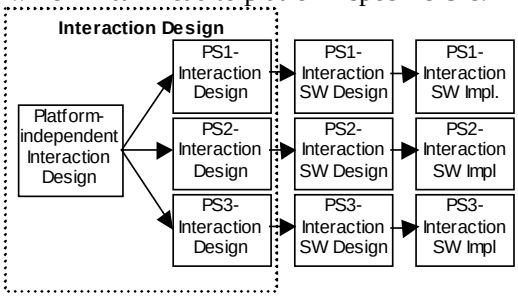

Figure 4: Usability Engineering process for multiple platforms

5.1 UIML-Based Usability Engineering Process

We have developed a framework that is very closely related to the UE process discussed above. The main building blocks within this framework are the *Logical Model*, *Family Model* and the *Platform-Specific Model*. The *logical model* is the new model that we are creating, and it is briefly described below. We described the *family* and *platform-specific models* in earlier sections. There is also a new set of transformations needed to convert the *logical model* to different *family models*. Thus, we have one new level of representation, the *logical model*, and one new process of transformations. The resulting diagram is shown in Figure 5.

The main objective of the *logical model* is to capture the UI description at a high level of abstraction. The logical model is a collection of logical constructs that represent the UI in some abstract fashion. We are considering using a hybrid task/domain model that can be transformed to the physical model. Research on this representation is underway.

The transformation from the *logical model* to a particular *family model* has to be developer-guided and cannot be fully automated. By allowing the UI developer to intervene in the transformation and mapping process, it is

easier to ensure usability. Once the developer had identified the mappings, the system generates a *family model* based on the target family mappings.

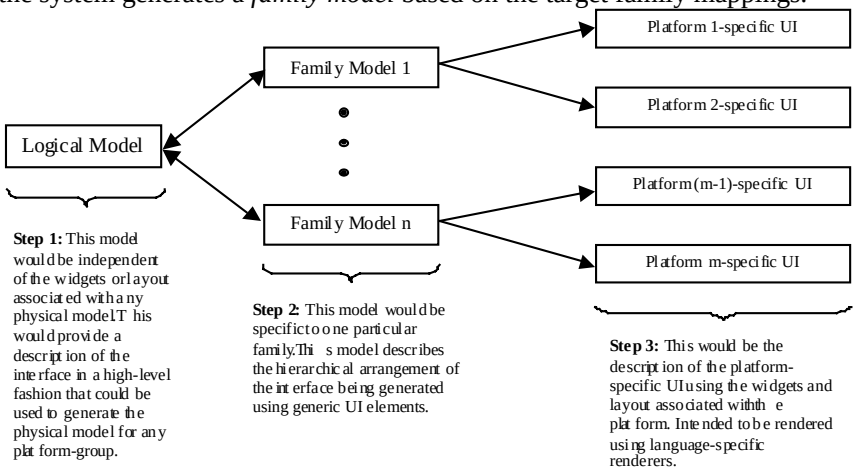

Figure 5: The overall framework for building multi-platform UIs using UIML

6. CONCLUSIONS

In this paper we have shown some of our research in extending and utilizing UIML to generate multi-platform user interfaces. We are using a single language, UIML, to provide the multi-platform development support needed. This language is extended via the use of a logical model, alternate vocabularies and transformation algorithms. Our approach allows the developer to build a single specification for a *family* of devices. UIML and its associated tools transform this single representation to multiple platform-specific representations that can then be rendered in each device. We have presented our current research in extending UIML to allow building of interfaces for very-different platforms, such as VoiceXML, WML and desktop computers.

REFERENCES

[1] Abrams, M., Phanouri, C., Batongbacal, A. and Shuster, J., *UIML: An Appliance-Independent XML User Interface Language*. in *8th World Wide Web*, (Toronto, 1999).

[2] Ali, M.F. and Abrams, M., *Simplifying Construction of Multi-Platform User Interfaces Using UIML*. in *UIML'2001*, (Paris, France, 2001).

[3] Asakawa, C. and Takagi, H., *Annotation-Based Transcoding for Nonvisual Web Access*. in *Assets'2000*, (Arlington, Virginia, USA, 2000), 172-179.

[4] Brewster, S., Leplâtre, G. and Crease, M., *Using Non-Speech Sounds in Mobile Computing Devices*. in *First Workshop on Human Computer Interaction of Mobile Devices*, (Glasgow, 1998).

[5] Calvary, G., Coutaz, J. and Thevenin, D., *Embedding Plasticity in the Development Process of Interactive Systems*. in *Sixth ERCIM Workshop "User Interfaces for All"*, (Florence, Italy, 2000).

[6] Clark, J. XSL Transformations (Version 1.0), 1999, http://www.w3.org/TR/xslt.

[7] Hix, D. and Hartson, R. *Developing User Interfaces: Ensuring Usability Through Product and Process*. John Wiley and Sons, Inc., 1993.

[8] Huang, A. and Sundaresan, N., *Aurora: A Conceptual Model for Web-Content Adaptation to Support the Universal Usability of Web-based Services*. in *Conference on Universal Usability, CUU 2000*, (Arlington, VA, USA, 2000), 124-131.

[9] Johnson, P., *Usability and Mobility: Interactions on the move*. in *First Workshop on Human Computer Interaction With Mobile Devices*, (Glasgow, 1998).

[10] Luo, P., Szekely, P. and Neches, R., *Management of Interface Design in Humanoid*. in *Interchi'93*, (1993), 107-114.

[11] Myers, B. User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*, *2* (1). 64-103.

[12] Myers, B., Hudson, S. and Pausch, R. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*, *7* (1). 3-28.

[13] Olsen, D. Interacting in Chaos. *Interactions*. 42-54.

[14] Olsen, D., Jefferies, S., Nielsen, T., Moyes, W. and Fredrickson, P., *Cross-modal Interaction using XWeb*. in *UIST'2000*, (CA, USA, 2000), 191-200.

[15] Phanouri, C., *UIML: An Appliance Independent XML User Interface Language*, Ph.D. Dissertaton, Computer Science, Virginia Tech, 2000.

[16] Puerta, A. and Eisenstein, J., *Towards a General Computational Framework for Model-Based Interface Development Systems*. in *IUI'99*, (CA, USA, 1999), 171-178.

[17] Puerta, A., Eriksson, H., Gennari, J.H. and Munsen, M.A., *Model-Based Automated Generation of User Interfaces*. in *National Conference on Artificial Intelligence*, (1994), 471-477.

[18] Sukaviriya, P.N. and Foley, J., *Supporting Adaptive Interfaces in a Knowledge-Based User Interface Environment*. in *Intelligent User Interfaces'93*, (1993).

[19] Szekely, P., Sukaviriya, P.N., Castells, P., Mukthukumarasamy, J. and Salcher, E., *Declarative Interface Models for User Interface Construction Tools: The MASTERMIND Approach*. in *6th IFIP Working Conference on Engineering for HCI*, (WY, USA, 1995).

[20] Thevenin, D. and Coutaz, J., *Plasticity of User Interfaces: Framework and Research Agenda*. in *INTERACT'99*, (1999), 1-8.

[21] Wiecha, C., Bennett, W., Boies, S., Gould, J. and Greene, S. ITS: A Tool for Rapidly Deveoping Interactive Applications. *ACM Transactions on Information Systems*, *8* (3). 204-236.