Chapter 1

UIML.NET: AN OPEN UIML RENDERER FOR THE .NET FRAMEWORK

Kris Luyten, Karin Coninx
{kris.luyten,karin.coninx}@luc.ac.be
Limburgs Universitair Centrum
Expertise Centre for Digital Media
Universitaire Campus, B-3590 Diepenbeek, Belgium
http://www.edm.luc.ac.be

Abstract As the diversity of available computing devices increases it becomes more difficult to adapt User Interface development to support the full range of available devices. One of the difficulties are the different GUI libraries: to use an alternative library or device one is forced to re-develop the interface completely for the alternative GUI library. To overcome these problems the User Interface Markup Language (UIML) specification has been proposed, as a way of glueing the interface design to different GUI libraries (in different environments) without further efforts. In contrast with other similar approaches UIML has matured and has some implementations proving its usefulness. We introduce the first UIML renderer for the .Net framework; a framework that can be accessed by different kinds of programming languages and can use different kinds of widget sets. We show that its properties, among them its reflection mechanism, are suitable for the development of a reusable and portable UIML renderer. Furthermore the suitability for multi-device rendering is discussed in comparison with our own multi-device User Interface framework Dygimes. The focus is on how layout management can be generalized in the specification to allow the GUI to adapt to different screen sizes.

Keywords: UIML, user-interface design and specification methods and languages, multi- and multiple-device User Interfaces, Automatic User Interface Generation

1. Introduction

It is a known fact that all computing environments become more heterogeneous every day. Instead of emerging to a common set of hardware and software platforms, computing gains at diversity. Nevertheless, a lot of attention is given to open standards supporting interoperability between different devices and software platforms.

The diversity raises the opportunity for new methodologies and techniques to support multi- (and multiple-) device User Interfaces (UIs). Several initiatives exist in the academic world as well as in the industry. Managing the reuse of interactive software components over several different kinds of devices is one of the problems tackled in this paper. One of the noticeable methodologies is the use of Model-Based User Interface Development (MBUID). Another one is the use of High-level User Interface Descriptions (HLUID), nowadays mostly based on the XML syntax). This work concentrates on the latter: the goal is to develop an adaptive, flexible HLUID renderer so it can be deployed easily in MBUID for multiple devices. In the existing literature there are several publications describing the usage of the HLUID within MBUID to support the design of multi-device UIs. XIML [7, 18], XWeb [15], XForms [6], XUL [8] and TERESA XML [13, 16] are especially worth mentioning here. They provide solutions for multi-device UI design and creation on different levels of abstraction. Another initiative is the Dygimes framework [5]: it combines several techniques like task modeling, HLUID, web services and constraint-based layout management to generate UIs for mobile and embedded systems *at runtime*.

Unfortunately, once the design reaches the presentation level it remains difficult to specify this in a device-independent manner. Very few HLUID succeed in being generic enough to be really independent of the widget set (e.g. some can only be used with Java widgets, or are only suitable with webbrowser support). The User Interface Markup Language (UIML) [2, 17] is a specification that is independent of a widget set and claims to be device-independent as well. Because the specification has matured over the years and efforts are emerging to submit it as a World Wide Web Consortium (w3c) specification, it is beneficial to develop renderers for the specification. Some of the current research efforts include creating better support for multi-platform UIs [3, 17]. Targetting multi-device environments implies the UIML renderer has to be very flexible: on different device there could be different widget sets, or the widget set API can be slightly different due to the different device profiles. This work also targets to create a UIML renderer that can man-

age and support evolution in widget set APIs and differences in widget set vocabularies without the need for changing the renderer itself.

The remainder of this paper is structured as follows: section 1.2 gives a short introduction into the UIML language. It provides the necessary details of the specification to understand the following sections. Next, in section 1.3 some related work and underlying technologies are discussed evaluating the use of UIML to illustrate the context of the work. This is followed by a description of the implemented renderer in section 1.4. Several aspects will be highlighted with the emphasis on the flexibility of the renderer. Section 1.5 identifies the layout management problem and proposes a solution for future HLUID renderers. The paper is concluded with an example in section 1.6 and with conclusions in the last section.

2. UIML Overview

The UIML specification is currently under revision for submission as a w3c standard. Consequently this means some changes in the specification can be expected and current renderer software design should support easy refactoring to adopt these changes.

An UIML document exists of several parts [1] that are shown in figure 1.1. Together they form the Meta-Interface Model (MIM):

Interface describes four parts of the UI:

- **Structure** : describes the "widget hierarchy" of the UI. It defines the different parts that are contained in the UI, and the abstract widget name of each part.
- **Style** : describes properties of the parts defined in the structure. This allows to change properties of the widgets like color, font, text,... The layout is also defined as a style of the parts in structure. Unfortunately the current way of defining a layout is not suitable for multi-device UIs, section 1.5 will elaborate further on this.
- **Content** : separates the content of the interface (e.g. the list that has to appear in a list presentation) of the other parts.
- **Behavior** : defines rules with actions that are triggered when some condition is met. Some kind of event mechanism is offered to the UI designer this way.

Vocabularies are referred to as *peers* in the UIML specification: this contains the mapping with the concrete UI toolkit. To allow the use of different devices and different GUI libraries, one can define several peers for the same UIML document while choosing the

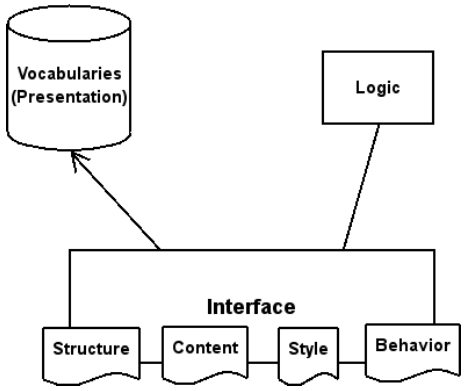


Figure 1.1. The UIML Meta-Interface Model

appropriate peer at runtime. The renderer described in this paper is limited to 2D widget sets.

Logic defines how to bind the UI with the application logic. More precise it describes the mappings with the software interface to communicate with the application logic.

3. Related Work

Until now, we are not aware of any previous work describing the actual implementation of an UIML renderer and releasing the source code. There were some initiatives in the past, but most of these projects only implemented parts of an obsolete specification version or are no longer supported. [4] describes how UIML can be converted into program code. Harmonia [3] offers a Java-based UIML renderer that implements most of the specification. Several other implementations are gathered on `http://www.uiml.org`, but most of them are deprecated and/or no source code has been released.

It is clear that UIML was designed with Object-Oriented programming languages in mind. Most mappings on the UI toolkit and the relations with the application logic rely on the existence of "classes" in the OO sense. The most mature implementation of the UIML renderer is the one provided by Harmonia, and is implemented in Java. However, the .Net framework offers some new possibilities to develop a UIML renderer. For example it supports on-the-fly executable code generation and better integration with web services. This is the first attempt to write a UIML renderer for the .Net Framework.

The widget set that is used to generate the UI is the GTK# (`http://gtk-sharp.sourceforge.net`) widget set. Although it is still in a development stage, and only few applications are implemented using this widget set, it is mature enough to use as a basis. The most well-known widget set is the Windows Forms library. This one is not chosen for two reasons: the first one is that it is more complex to use than most other .Net widget sets, the second reason is to keep the renderer independent of the Microsoft implementation. Portability to different platforms and availability are important issues, so the renderer was not implemented in the Microsoft .Net Framework, but in the Mono (`http://www.go-mono.com`) implementation of .Net. Both .Net Frameworks implement the same ECMA standard, so the implementation should be reusable as is.

4. The Renderer

4.1 Overall Design

A HLUID can be processed in two different ways: either it can be *compiled* or *rendered*. The former transforms the specification into program code, the latter provides a rendering engine that can interpret the UIML document. When the UIML document is transformed into source code ("compiled"), on its turn the resulting source code needs to be compiled. Transforming the UIML document into program code is advantageous when the code still needs to be manually changed afterwards.

The rendering approach is more complex to implement, but is more flexible: it allows fast prototyping because an UIML document can be tested directly, it can offer dynamic changes in the UI and a transparent mechanism for connecting the rendered UI with the application logic.

Several parts of the renderer can be distinguished:

Vocabulary Generator One of the most cumbersome and tedious tasks is to create a vocabulary for a particular widget set. When the vocabularies are manually edited this often results in different incompatible vocabularies and incomplete mappings. When the widget set API gets updated, often the vocabulary has to be updated manually if one wants to support the latest version of the widget set. This process can be automated when the implementation language supports reflection, e.g. Java and C# have reflection support. Reflection allows software to inspect implementation code and APIs at runtime.

Interface reader In the initial stage the UIML document has to be processed. The Interface reader processes the document and stores

it in an appropriate data structure. Notice that it is recommended to keep this data structure in memory during the lifespan of the UI: dynamic changes in the style and the UI structure can be supported better this way.

Style repository The style properties included in a UIML document are implemented in a repository-like manner. On the one hand the part that is specified beforehand is queried using XPath expressions. On the other hand there is support for properties that are added at runtime by an internal data structure.

Rendering Backends The specification allows different peers to coexist for the same interface specification. A peer defines the language bindings for the interface, thus which widget set is being used and how it can interact with the application logic.

System Glue The system glue connects the concrete interface with the application logic. There are different ways to do this; by means of direct method invocation, remote method invocation or through web services. All three ways are supported by the .Net framework making it a powerful choice for implementing the renderer.

Figure 1.2 gives an overview of the architecture of the renderer. Figure 1.3 illustrates the rendering process of the uiml.net renderer.

4.2 Dynamic Core

Roughly spoken there are two ways of implementing a renderer for a UI markup language:

Static renderer The implementation relies on specific knowledge of the widget set. The types offered by the target GUI library are loaded and used at compile-time.

Dynamic renderer The implementation does not rely on specific knowledge of the widget set. The types offered by the target GUI library are loaded and used at runtime.

The former is more robust but less flexible and requires more program code. The latter takes full advantage of the information offered by the peer descriptions (vocabularies); it requires a detailed mapping description in the vocabulary.

Reflection is a very powerful tool to use when mapping the Abstract Interface Objects (AIO) to Concrete Interface Objects [19]. AIOs are abstract representation of widgets, and CIOs are the concrete representation; e.g. a “range indicator” is an AIO and can be mapped to a

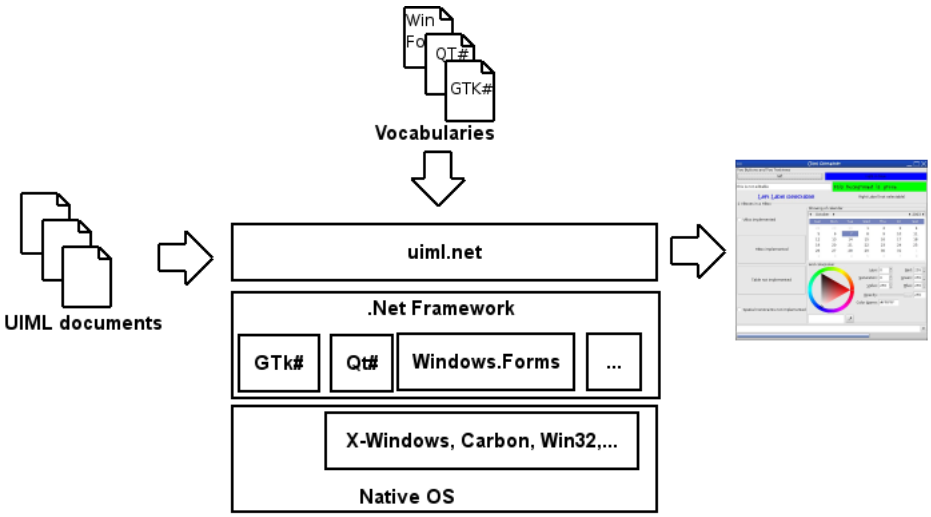


Figure 1.2. A rough sketch of the uiml.net architecture

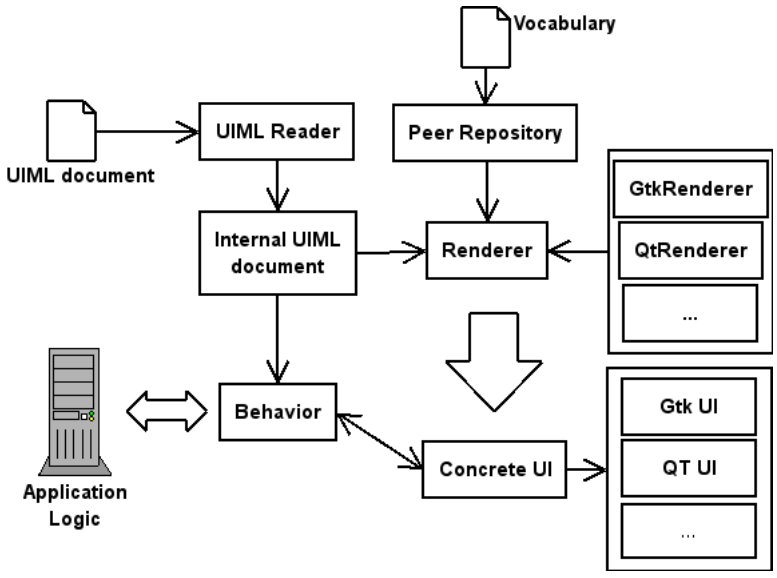


Figure 1.3. Processing an UIML file with uiml.net

slider widget (CIO). The rendering engine itself has no notion of concrete widgets, but will be guided by the vocabulary to search for the appropriate mapping. Even when the concrete widgets are found (including its class name and properties), the renderer will avoid using the explicit class names. Instead it queries the available libraries containing possible widget sets with the information retrieved from the vocabulary. The reflection mechanism allows to construct new objects using solely this information. This has several advantages:

- The rendering engine is **reusable** for other widget sets, since it does not explicitly creates the concrete widgets directly.
- The vocabulary can be extended **independently** of the renderer. When the widget set is updated, only the vocabulary has to be updated. New entries in the vocabulary can be used without further adaption of the program code.
- The renderer is more **portable** across devices. E.g.: it can be ported to platforms that only offer a limited version of the same widget set.

5. The Layout Problem

One of the pitfalls making UIML less suitable for multi-device interfaces is the lack of support for uniform layout management. We propose to use spatial hierarchical layout constraints to overcome this problem. [14] rightfully argues that constraint-based systems have not caught on for UIs, nevertheless *simple* constraints can be succesfull for specifying the layout of a system. Since the very beginning of GUI creation constraints are investigated to obtain better layout management. [11] gives an overview of different techniques using constraints for the layout of graphical interfaces. Thinglab used constraints for graphical simulation [9]. In [20] some techniques are discussed that a renderer could implement to obtain a visually pleasing result.

Typically the layout of the UI is influenced by the interface and style parts of the UIML document. Our approach differs with traditional approaches in the sense that we also use the hierarchy as described by UIML in the structure element instead of directly referring to the individual parts. Most available vocabularies have the layout specified as parts of the properties that can be defined in the style section of a UIML document.

In the way we implemented the renderer, the interface part determines how the concrete UI will be nested and the style part specifies the more widget-set related possibilities using layout managers. Using

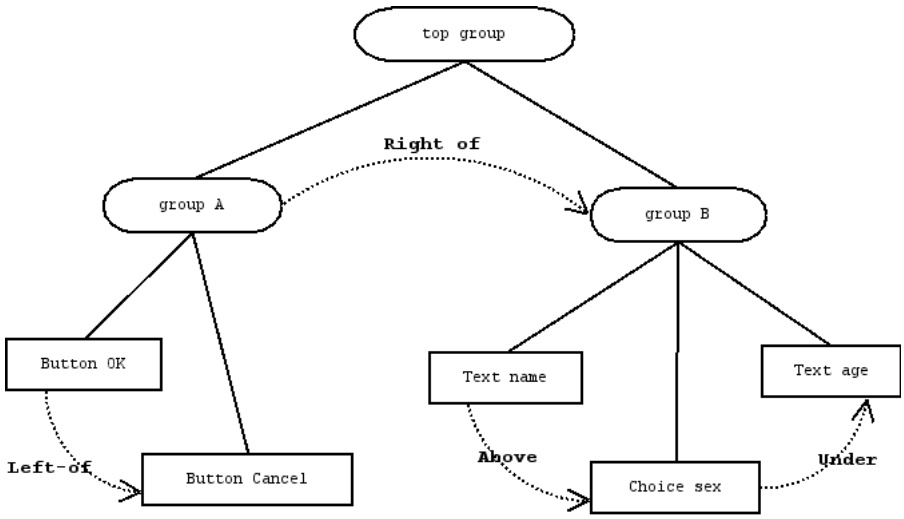


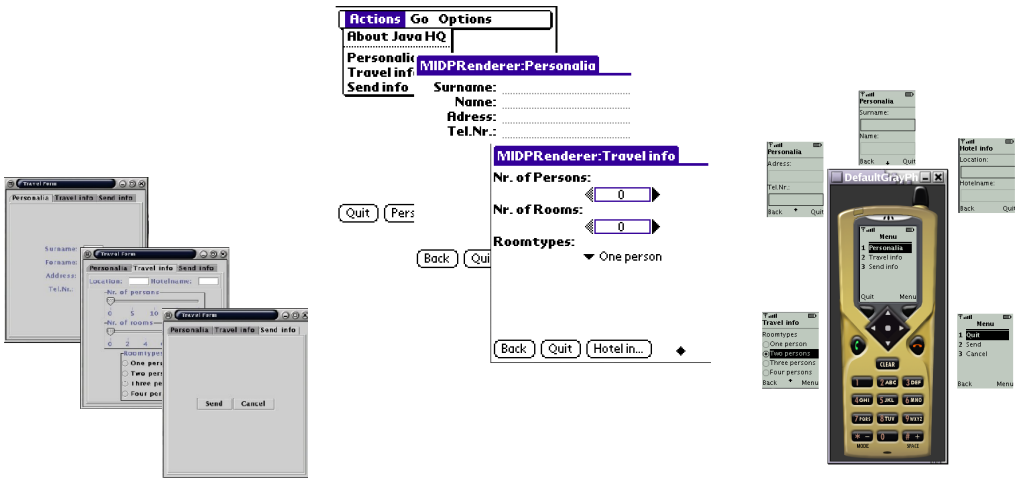
Figure 1.4. A visual representation of the constraint definition for a structure tree

spatial layout constraints this separation can be preserved, while adding adaptability when rendering the UI. Constraints are only defined between siblings in the structure tree. A visual representation of this related to the example in section 1.6 is shown in figure 1.4. The hierarchy divides the interface in groups. These groups can be subdivided in other groups and so on. All widgets that are part of the same group, have a logical relation with respect to each other. Some rules can be applied here:

- A group describes a set of **logically related** abstract interactors or groups of abstract interactors. The designer should decide which widgets are gathered in a group.
- A group can be specified **splittable** (as a UIML property). This specifier allows the layout manager to show the abstract interactors or groups of abstract interactors in separate spaces.
- The group specifier **non-splittable** (as a UIML property) forces the layout manager to show the children of the group as a whole to make sense to the user. Note that "non-splittable" is only valid for the direct children of the group, and does not affect the further offspring.

For now, we have not implemented this into the uiml.net renderer because it would break the current specification. The layout manage-

ment should be generic and not related to any widget set and modalities. By consequence this requires adding new elements into the UIML specification, e.g. tags to define constraints. The spatial constraints are implemented in the Dygimes framework [5, 12] for testing purposes and has proven to be a feasible solution for UIs containing a limited amount of widgets. Results obtained in this experiment to combine HLUID with spatial constraints can be seen in figure 1.5. The figure shows how a hotel registration form described in a HLUID can be rendered to different devices.



(a) IPaq 3970 (b) Palm IIIc (c) Cell phone

Figure 1.5. Hotel registration form

6. A Multi(ple)-Device Picture Browser

To provide the reader with a clear understanding we introduce a simple example here: the *multi(ple)*-device picture browser. We will show how to create a simple picture browser that can be rendered on multiple different devices, distributing the interface over several devices. Imagine the following scenario: nowadays most people have a digital camera. It would be nice to design one application for browsing your pictures that works on most devices. Sometimes it is even usefull to distribute several parts of the UI over several devices. For example suppose you want to give a presentation showing your pictures on a large screen. The controls for browsing the pictures (e.g. the “previous” and “next” button) can

(a) On a desktop

(b) The controls rendered separate

Figure 1.6. The Multi(ple)-device Picture Browser

be shown on a PDA, while the pictures can use the full screen space of the available monitor. An extract of the UIML document can be seen in figure 1.7.

When the UIML document shown in figure 1.7 is rendered for the desktop, the UI depicted in figure 1.6(a) is the result. Suppose we are only interested in a part of the UI being rendered for our PDA, more specific the controls section (thumbnails and buttons) of the picture browser. This results in only a selection of the structure tree being rendered as shown in figure 1.6(b). The interesting feature is that the pruning can be done at runtime, without manual intervention (apart from the user interaction to initiate this action).

7. Conclusion

We discussed the implementation of a User Interface Markup Language (UIML) renderer and a possible extension: better adaptability for multiple/multi-device environments throughout the usage of spatial constraints. Our goals are to compare the UIML specification with the HLUIDs supported by the Dygimes framework and to contribute to the development of UIML; the leading specification for multi-device UI de-

```
<uiml>
 <interface>
  <structure>
   <part class="Frame" id="picbrowser">
     <part class="Frame" id="theimg">
       <part class="Image" id="image"/>
     </part>
     <part class="Frame" id="remote">
      <part class="Frame" id="thumbnails">
       <part class="Image" id="prev2"/>
       ...
       <part class="Button" id="previous"/>
      </part>
    </part>
   </part>
  </structure>
  ...
```

Figure 1.7. Part of the UIML document

velopment. We explored how the .Net framework allows to create a renderer that is bound to a widget set at runtime (through reflection) instead of at compile-time. This results in an easy extensible rendering engine that supports evolution of widget set vocabularies.

The source code of uiml.net can be obtained at `http://www.edm.luc.ac.be/software/uiml.net/`.

8. Acknowledgements

The authors would like to thank Bert Creemers for his help implementing the spatial layout constraints in Dygimes. The research at the Expertise Centre for Digital Media (EDM/LUC) is partly funded by the Flemish government and EFRO (European Fund for Regional Development).

References

[1] Marc Abrams and Jim Helms. User Interface Markup Language (UIML) Specification version 3.0. Technical report, Harmonia, 2002.

[2] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An appliance-independent XML user interface language. *WWW8 / Computer Networks*, 31(11-16):1695–1708, 1999.

[3] Mir Farooq Ali, Manuel A. Pérez-Quiõnes, Marc Abrams, and Eric Shel. Building Multi-Platform User Interfaces with UIML. In Kolski and Vanderdonckt [10], pages 255–266.

[4] Carsten Binnig and Andreas Schmidt. Development of a UIML Renderer for Different Target Languages: Experiences and Design Decisions. In Kolski and Vanderdonckt [10], pages 267–274.

[5] Karin Coninx, Kris Luyten, Chris Vandervelpen, Jan Van den Bergh, and Bert Creemers. Dygimes: Dynamically Generating Interfaces for Mobile Computing Devices and Embedded Systems. In *Human-Computer Interaction with Mobile Devices and Services, 5th International Symposium, Mobile HCI 2003*, pages 256–270, Udine, Italy, sept 8–11 2003. Springer.

[6] World Wide Web consortium. *XForms*. World Wide Web, `http://www.w3.org/TR/xforms/`, 2001.

[7] Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta. Applying Model-Based Techniques to the Development of UIs for Mobile Computers. In *IUI 2001 International Conference on Intelligent User Interfaces*, pages 69–76, 2001.

[8] David Hyatt, Ben Goodger, Ian Hickson, and Chris Waterson. *XML User Interface Language (XUL) Specification 1.0*. World Wide Web, `http://www.mozilla.org/projects/xul/`, 2001.

[9] Maloney J, Boming A, and Freeman-Benson BN. Constraint Technology for User Interface Construction in ThingLab II. In *OOPSLA*, 1989.

[10] Christophe Kolski and Jean Vanderdonckt, editors. *Computer-Aided Design of User Interfaces III*, volume 3. Kluwer Academic, 2002.

[11] Simon Lok and Steven Feiner. A Survey of Automated Layout Techniques for Information Presentations. In *Proceedings of SmartGraphics 2001*, March 2001.

[12] Kris Luyten, Bert Creemers, and Karin Coninx. Multi-device Layout Management for Mobile Computing Devices. Technical Report TR-LUC-EDM-0301, Limburgs Univeristair Centrum – Expertise Centre for Digital Media, September 2003. Available at `http://lumumba.luc.ac.be/kris/research`.

[13] Giullio Mori, Fabio Paternò, and Carmen Santoro. Tool Support for Designing Nomadic Applications. In *Intelligent User Interfaces*, pages 141–148, January 12–15 2003.

[14] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000.

[15] Dan R. Olsen, Sean Jefferies, Travis Nielsen, William Moyes, and Paul Fredrickson. Cross-modal interaction using XWeb. In *Proceedings of the 13th Annual Symposium on User Interface Software and Technology (UIST-00)*, pages 191–200, N.Y., November 5–8 2000. ACM Press.

[16] Fabio Paternò and Carmen Santoro. One model, many interfaces. In Kolski and Vanderdonckt [10], pages 143–154.

[17] Constantinos Phanouriou. *UIML: A Device-Independent User Interface Markup Language*. PhD thesis, Virginia Tech, 2000.

[18] Angel Puerta and Jacob Eisenstein. XiML: A Common Representation for Interaction Data. In *Sixth International Conference on Intelligent User Interface*, pages 214–215, 2002.

[19] Jean Vanderdonckt and François Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *ACM Conference on Human Aspects in Computing Systems InterCHI'93*, pages 424–429. Addison Wesley, 1993.

[20] Jean Vanderdonckt and Xavier Gillo. Visual Techniques for Traditional and Multimedia Layouts. In *Advanced Visual Interfaces*, pages 95–104, 1994.