26

Markup Languages in HCI

26.1 Introduction 26-409
26.2 What Can Be Specified with Markup Languages 26-410
Abstract Descriptions • Context of Use • Guidelines • Markup Languages for Implementation of Interactive Applications • Transformations
26.3 Existing Markup Languages for Multidevice UIs 26-416
XIML • UIML • TERESA XML • UsiXML • Pebbles • XForms
26.4 Transforming Markup-Language-Based Specifications 26-425
26.5 Conclusions 26-427
References 26-427

Fabio Paternò and
Carmen Santoro

26.1 Introduction

Markup languages are increasingly used in HCI to represent relevant information and process it. Interest in them has been stimulated, in particular, by the growing availability of many potential interaction devices and the need for supporting a wide variety of users, including the disabled. The XML metalanguage has become the *de facto* standard for creating markup languages, and a variety of XML-based languages have been proposed to address various aspects relevant to HCI. A markup language is a set of words and symbols useful for identifying and describing the different parts of a document. It combines text and related extra information, expressed using markup symbols intermingled with the primary text in a hierarchical structure of elements and attributes.

Some reasons can be identified at the basis of XML's popularity. First, different from some markup languages that are purpose-specific (e.g., HTML for describing document appearance) and that cannot be reused for a different goal, XML is a self-describing format in which the markup elements represent the information content. Thus, XML completely leaves the interpretation of such data to the application that reads them, and information content is separated from information rendering, making it easy to provide multiple views of the same data. By leaving the names, hierarchy, and meanings of elements/attributes open and definable, XML lays the foundation for creating custom and modular (new formats can be defined by combining and reusing other formats) XML-based markup languages. Also, XML has a plain text format, which means that it is both human and machine-readable. The wide availability of tools for text file authoring software facilitates rapid XML document authoring and maintenance, and cross-platform interoperability. This was not so easy before XML's advent when most data interchange formats were proprietary "binary" formats, and therefore not easily shared by different software applications or across different computing platforms. Moreover, the strict syntax and parsing requirements allow the appropriate parsing algorithms to remain simple, efficient, and consistent. XML is a robust, logically verifiable format based on international standards and is unencumbered by licenses or restrictions. Lastly, it is well supported. Thus, by choosing XML, it is possible to access a large and growing community of tools, services, and technologies based on it (XLink , XPointer, XSLT, but also RDF and the semantic web).

XML-based languages have been considered to address various aspects relevant to HCI. For example, as pervasive computing evolves, interactive application developers should cope with the problem of providing solutions for simultaneous deployment on a growing number of platforms for disparate users in a wide variety of contexts. Because developing ad hoc solutions might result in a significant overhead, one feasible solution is abstracting from the presentation details of the specific medium used for the interaction, and focusing more properly on aspects related to the semantic of the interaction, namely what is the expected result that an interaction should reach. Thus, capturing and describing the essence of what a user interface should be can be obtained by identifying logical descriptions that contain semantic information and are able to highlight the main aspects to consider.

In addition, applications for mobile users force designers to take into account a number of related aspects that could affect the way in which interactive applications might change not only their rendering but also the functionality provided to the users, depending on the context in which the interaction takes place. Such aspects might include the different categories of users representing the expected target population, the environmental conditions in which interaction might occur, the particular

device that might be used, and so on. Therefore, such different aspects should be described by appropriate specifications. Such logical specifications are usually described using XML-based languages.

The first issue connected with the adoption of XML-based languages for specifying user interface-related aspects at different abstraction levels is providing specific automatic tools able to understand such specifications, and consequently supporting the designers in deriving effective user interfaces. Another issue is semantically correlating such specifications to each other, so as to provide designers with different views of the same interactive system, along with automatic tools able to transform such different specifications.

This chapter is intended to extensively address and discuss the aforementioned topics. After a brief introduction about the usefulness of markup languages in computing, the chapter discusses what aspects relevant to HCI can be addressed through the XML-based languages that have been proposed (abstract/concrete user interface descriptions, tasks, users, devices, environments, etc.). Next, it is discussed how guidelines can be formalized in markup languages and how such information can be exploited to support user interface designers and evaluators, in particular in the accessibility area. A section is dedicated to markup languages for multidevice interfaces, which is a particularly important and timely topic. This section includes reviewing and discussing existing approaches that exploit XML-based languages for describing information useful for designing interactive systems. Furthermore, a subsequent section is dedicated to the motivations and usefulness for transforming the different specifications (i.e., for forward/reverse engineering, redesign for different platforms, multidevice support, etc.). Last, some conclusions are drawn and indications are provided of current challenges in the area of markup languages specifically used for supporting the development of future pervasive interactive systems exploiting ambient intelligence.

26.2 What Can Be Specified with Markup Languages

Markup languages are a very general tool to represent any type of information; they have been found particularly useful to represent several types of aspects in HCI:

- *Abstract descriptions of interactive systems*: There are several possible views on an interactive system that differ depending on the abstraction level considered. The purpose of such levels is to highlight the semantic aspects, removing low-level details that are not particularly important.
- *Context of use*: People can interact with systems in a wide variety of contexts of use, which differ in terms of the user characteristics, the technology available (in terms of interaction resources and modalities, connectivity, etc.), the surrounding environment (level of noise, luminosity, etc.), and the social context.
- *Guidelines*: An increasing number of guidelines have been proposed to help in accessibility and usability evaluation.
- *Implementation languages*: In particular for web environments, many types of implementation languages based on XML have been proposed, which also support different modalities: form-based interfaces are described by XHTML and XUL, while vectorial graphics is supported through SVG (scalable vector graphics, see http://www.w3.org/Graphics/SVG), multimodal and multimedia applications can be obtained by X+V (http://www.voicexml.org/specs/multimodal/x+v/12) and SMIL (http://www.w3.org/AudioVideo).
- *Transformations*: With so many XML-based languages used for describing both models and implementation languages, and used in an interoperable way among different tools, the problem emerges to define transformations able to translate such languages from one to another.

Figure 26.1 shows the different ways in which XML-based descriptions have been used in HCI (see boxes with bold labels). One of the most common uses is identifying XML-based languages for specifying different descriptions that are relevant for context-dependent interactive applications. This includes not only different descriptions of a user interface according to various abstraction levels (task and object level, abstract level, and the concrete one), but also conditions regarding the current context that are deemed relevant for the design and evaluation of the interactive application (since they can have an impact on both of them). Another goal for which XML-based languages are more and more used is the specification of final implementation languages for user interfaces.

Also, the mechanism underlying XML, which allows tagging the different parts of a description of a user interface (at various levels, from the most abstract one to the final level) provides twofold benefits. On the one hand, XML tagging-based languages allow providing a logical meaning to the different parts of the description. On the other hand, the use of such XML-based languages is a natural stimulus for developing XML-based languages to specify the transformations that can relate such various descriptions with each other. Last, XML-based descriptions have also been used to describe guidelines, which are meant to be provided as input to tools for evaluating interactive applications.

26.2.1 Abstract Descriptions

In the research community in model-based design of user interfaces there is a general consensus on what useful logical descriptions are (Szekely, 1996; Paternò, 1999; Calvary et al., 2003; see also Chapter 25, "Model-based Tools: A User-Centered Design for All Approach"):

- The task and object level, which reflects the user view of the interactive system in terms of logical activities and objects manipulated to accomplish them
- The abstract user interface, which provides a modality-independent description of the user interface

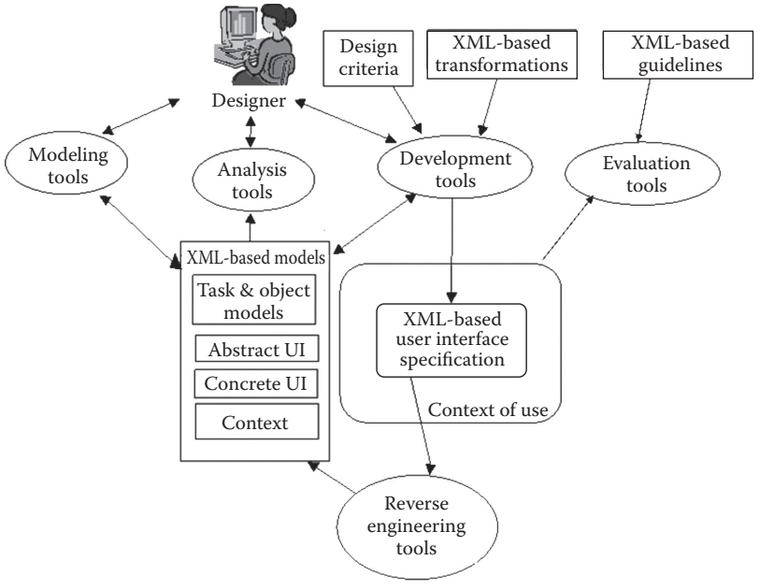

FIGURE 26.1 Using XML-based specifications in HCI.

- The concrete user interface, which provides a modality-dependent but implementation-language-independent description of the user interface
- The final implementation, in an implementation language for user interfaces

Thus, for example, the task "select vegetarian menu" implies the need for a selection object at the abstract level, which indicates nothing regarding the platform and modality in which the selection will be performed (it could be through a gesture or a vocal command or a graphical interaction). When moving to the concrete description, a specific platform has to be assumed, for example, the graphical PDA, and a specific modality-dependent interaction technique needs to be indicated to support the interaction in question (e.g., selection could be through a radio-button or a dropdown menu), but nothing is indicated in terms of a specific implementation language. After choosing an implementation language, the last transformation from the concrete description into the syntax of a specific user interface implementation language is performed. The advantage of this type of approach is that it allows designers to focus on logical aspects and take into account the user's view right from the earliest stages of the design process.

Figure 26.1 shows how various XML-based descriptions can be used at different stages in the life cycle of the interactive software application. Figure 26.2 shows how data contained in different XML-based descriptions can be used both at run-time and design time in the process of generating user interfaces.

At design time, the idea is to precompute once the different versions of a UI for the various platforms the designers want to address. To this aim, various levels of abstractions are considered to obtain a refinement mechanism able to transform high-level abstraction descriptions to more concrete specifications, up to the final implementation. Such transformations are carried out by taking into account information provided in various relevant models (user, platform, environment, etc.) and should also be specified to transform concrete user interface descriptions to abstract ones to maintain model consistency.

Furthermore, some data contained in such models can be more or less relevant, depending on the transformation (between two different abstraction levels) considered. For instance, depending on the characteristics specified in the model of the platform currently considered, an appropriate transformation will be supported for obtaining a concrete user interface from an abstract one, so as to take into account the features and resources of the device at hand.

The data contained in such models are also useful at runtime, since they are considered when a dynamic reconfiguration is needed as a consequence of some occurring events. For example, if the user changes a device, the reconfiguration phase has to calculate again the effects of the changes occurred, and then trigger a dynamic generation of the user interface so as to adapt the UI to the new device. Section 26.3 provides a description of different markup languages for multidevice user interfaces.

26.2.2 Context of Use

The context of use is defined by a number of aspects (user, device, environment, social context), which can be formalized at various extents. In particular, the user and the device have been the subject of various proposals aiming to represent the associated important information. In general, in user modeling the goal is to represent the level of knowledge or interest of a user regarding a number of topics (see also Chapter 24, "User Modeling: a Universal Access Perspective"). This information

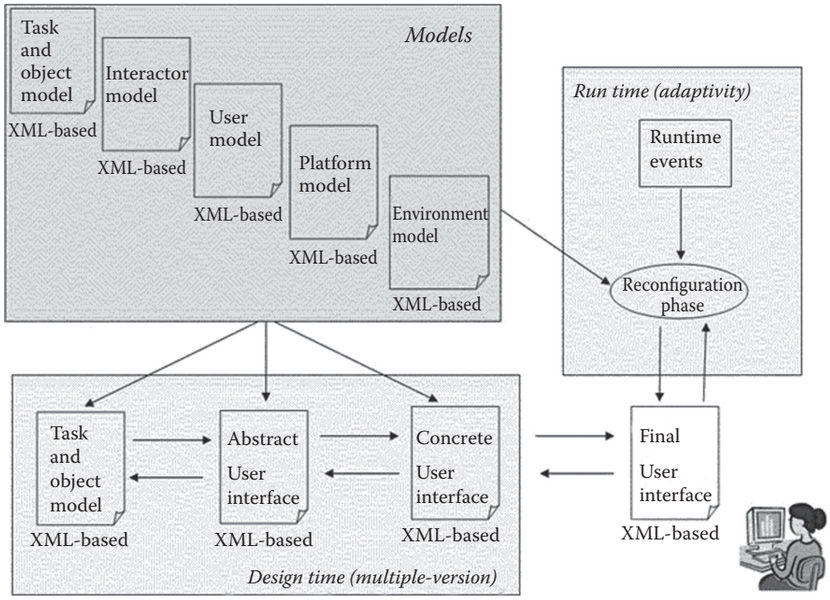

FIGURE 26.2 Generating user interfaces by using XML-based descriptions at design time and run-time.

can be represented at different levels of detail: binary (known/unknown), qualitative (good/medium/poor), and quantitative (i.e., probability that the user knows a piece of information). The user model can be used to adapt the user interface. Several aspects of the user interface can adapt depending on the user: presentation (e.g., in the choice of layout, colors, fonts, etc.), content (depending on the level of knowledge of the user), and navigation (links can be enabled or disabled depending on user interests). Ubiquitous computing offers new chances and challenges to the field of user modeling. With the markup language UserML, Heckmann and Krueger (2003) propose a language for combining partial user models in a ubiquitous and comprehensive computing environment, where all different kinds of systems work together to satisfy the user's needs.

The main idea of UserML is to enable communication about partial user models via the Internet. Therefore, one goal of UserML is the representation of partial user models. Using XML as a knowledge representation language has the advantage that it can be used directly in the Internet environment. One disadvantage is that the nested structures of the XML tags only represent a tree, while often the structure of a graph is needed. The approach used for UserML is, therefore, a modularized approach (see Figure 26.3) in which several categories will be connected via identifiers (IDs) and references to identifiers (ID-Refs). The content of a UserML document is divided into MetaData, UserModel, InferenceExplanations, ContextModel, and EnvironmentModel. In this way, the tree structure of XML can be extended to represent graphs. Therefore, a two-level approach is proposed. At the first level, a simple XML structure is offered for all entries of the partial user model. These UserData elements consist of the elements category, range, and value. At the second level, there is the ontology that defines the categories. The advantage of this approach is that different ontologies can be used with the same UserML tools, thus different user modeling applications can use the same framework and keep their individual user model elements. In the following excerpt, an example is presented of a partial user model that uses categories from the ontology "UserOL"

The UserModel consists of an unbounded list of UserData entries. Each one defines the category, the range, and the value. The reference to the ontology can also be set by default in the

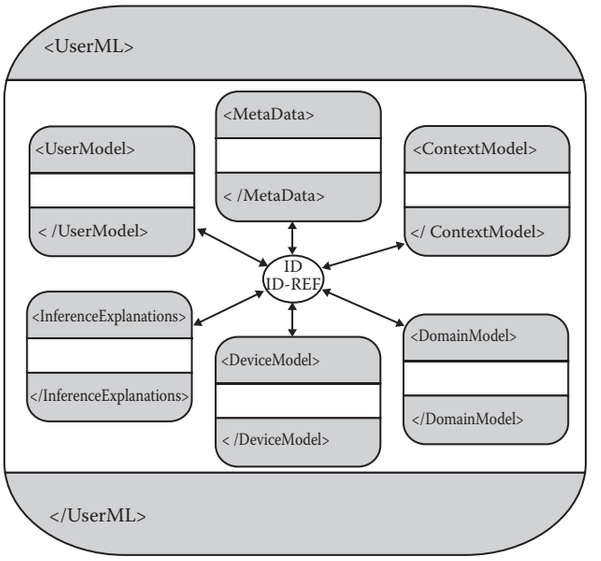

FIGURE 26.3 UserML containers, connected via IDs and ID-REFs.

```
<UserModel>
  <UserData id="231">
    <category>userproperty.timepressure</category>
    <range>low-medium-high</range>
    <value>high</value>
    <ontology>"http://www.u2m.org/UserOL/"</ontology>
  </UserData>
  <UserData id="224">
    <category>userproperty.walkingspeed</category>
    <range>slow-normal-fast</range>
    <value>fast</value>
    <ontology>"http://www.u2m.org/UserOL/"</ontology>
  </UserData>
  <UserData id="122">
    <category>usercontext.location</category>
    <range>airport.location</range>
    <value>X35Y12</value>
    <ontology>"http://www.u2m.org/UserOL/"</ontology>
  </UserData>
</UserModel>
```

UserModel element. An alternative approach would have been to encode the user modeling knowledge into the XML elements <timepressure>, <psychological-state>, <typing-behavior>.

An implementation architecture has also been elaborated of a general user model editor, which is based on UserML. The current implementation of this editing tool transforms UserML into XForms with XSLT.

Regarding the devices, the need for describing their features derives from the increasing availability of various types of interactive devices. Thus, to have applications able to adapt to their features, there should be a way to represent them. This is particularly important in the area of mobile phones, in which the possible characteristics are the most variable ones. The generic Composite Capabilities/Preference Profiles (CC/PP) framework (http://www.w3.org/2001/di) provides a mechanism through which a mobile user agent—a client, such as a browser, that performs rendering within a mobile device—can transmit information about the mobile device. The user agent profile (UAProf, 2001; http://www.openmobilealliance.org/release_program/uap_v20.html) is based on the CC/PP framework. It includes device hardware and software characteristics, information about the network the device is connected to, and other attributes. It is possible to identify a device through the header of HTTP requests. All the devices complying with UAProf have a CC/PP description of their characteristics in a repository server, which can be queried for obtaining the related information. The description of the devices is in the *Resource Description Framework* (see http://www.w3.org/RDF), another XML-based language. When a mobile device sends a request, it also informs about the URL where its profile is located through a specific field in the request called *X-Wap-Profile*. For example, the *x_wap_profile* for a Nokia N9500 is:

x_wap_profile:http://nds1.nds.nokia.com/uaprof/N9500r100.xml

The textbox that follows shows an excerpt of a profile for a mobile device: AQ1

Another proposal in this area is WURFL (wireless universal resource file) (Passani, 2005), which is an XML configuration file that contains information about capabilities and features of several wireless devices. The main objective is to collect as much information as possible about all the existing wireless devices that access WAP pages, so that developers will be able to build better applications and better services for the users. This proposal aims to support web applications for mobile devices. The goal is to programmatically abstract away devices' differences, avoid the current situation wherein we need to modify applications whenever a new device ships, and avoid the current situation that we need to track new devices that ship (particularly those in uninteresting markets). The basic idea is to provide a global database of all devices and their capabilities. The underlying assumption is that browsers are different, but they also have many features in common, and that browsers/devices coming from the same manufacturer are most often an evolution of the same hardware/software. In other words, differences between, for example, a Nokia 7110 and a Nokia 6210 are minimal; devices from different manufacturers may run the same software. For example, Openwave Systems provides the browser to Siemens, Motorola, Alcatel, Mitsubishi, Samsung, Panasonic, Telit, and Sagem. WURFL has created a compact, small, and easy way to update a matrix. WURFL is based on the concept of a family of devices. All devices are descendents of a generic device, but they may also descend from more specialized families (such as those who use the same browser). Its goal is to overcome some limitations of the UAProf standard developed in the OMA. UAProf

```
<?xml version="1.0"?>

  <rdf:RDF
     xmlns:rdf= "http://www.w3.org/..."
     xmlns:prf="http://www.openmobilealliance.org/..."
     xmlns:mms="http://www.wapforum.org/..."
     xmlns:pss5="http://www.3gpp.org/...">
     <rdf:Description rdf:ID="Profile">
       ......
       <prf:component>
         <rdf:Description rdf:ID="HardwarePlatform">
           ......
           <prf:PixelAspectRatio>1x1</prf:PixelAspectRatio>
           <prf:PointingResolution>Pixel</prf:PointingResolution>
           <prf:ScreenSize>640x200</prf:ScreenSize>
           <prf:ScreenSizeChar>29x5</prf:ScreenSizeChar>
     <prf:StandardFontProportional>Yes</prf:StandardFontProportional>
           <prf:SoundOutputCapable>Yes</prf:SoundOutputCapable>
           <prf:TextInputCapable>Yes</prf:TextInputCapable>
           <prf:Vendor>Nokia</prf:Vendor>
           <prf:VoiceInputCapable>No</prf:VoiceInputCapable>
       ......
       </rdf:Description>
    </prf:component>
    ......
  </rdf:Description>
</rdf:RDF>
```

seems to rely too much on someone else setting up the infrastructure to request profiles. There are cases of manufacturers just associating the profile of different phones into a new one. WURFL can be installed in any site and does not need to access device profiles from a repository on the Net.

26.2.3 Guidelines

To assure a certain level of accessibility and usability of user interfaces (UIs), several design criteria and guidelines have been proposed in the literature. Several detailed guidelines were formulated for general UIs.

Guidelines are intended for developers and designers: they are general principles that can be followed to improve application accessibility and usability. Many countries have adopted legislation that imposes some level of compliance to accessibility guidelines. This has raised interest in tool support. The goal of tool support is not to completely replace human evaluators and designers, but to help them manage the complexity of many existing web sites, apply evaluation criteria in a consistent manner, and thereby make their work more efficient. Some tools for this purpose already exist, such as Bobby (http://www.watchfire.com/products/webxm/bobby.aspx), LIFT (http://www.usablenet.com), and A-Prompt (http://aprompt.snow.utoronto.ca/), but the increasing request for support poses new issues that deserve better answers. In particular, three important areas have been identified for accessibility tools: support for checking multiple sets of guidelines, code correction, and reporting.

The issue of multiple guidelines involves many factors. One is that there are various organizations that issue standard guidelines; some are international standards (such as the W3C/WAI, or the ISO FDIS 9241-171 and TS 16071), and others are national (such as Section 508). They often share the same body of knowledge but also have slight differences that designers have to consider in some contexts. Another reason is that, while accessibility guidelines usually provide an important step forward to make sites accessible to everyone, including the disabled, often their mere application does not provide usable results for some classes of users. Indeed, being able to access information is not enough. In fact, although a service may be accessible to certain types of users (such as the blind), it still may not be sufficiently usable to such users. While accessibility and usability are closely related to each other, they address slightly different issues. Accessibility aims to increase the number of users who can access a system, this means to remove any potential technical barrier that does not allow the user to access the information. Usability aims to make the user interaction more effective, efficient, and satisfactory. Thus, it is possible to have systems that are usable but not

accessible, which means that some users cannot access the information, but those who can access it are supported in such a way as to find it easily. It is also possible to have systems accessible but not usable, which means that all users can access the desired information, but this can only occur after long and tedious interactions. Consequently, there is also a need for integrating these two aspects to obtain interactive services for a wide variety of users, including people who are disabled.

This section mainly focuses on guidelines for web applications, because the web is currently the most common UI environment. Each guideline can include one or more checkpoints. Checkpoints are technical solutions that support the application/evaluation of the criteria and usually correspond to specific implementation constructs that guarantee the satisfaction of the associated guideline. For example, the guideline "Logical partition of interface elements" expresses the concept of well-structuring and organizing the page content. So, it provides a general principle that should be taken into account by developers during web site design. Then, developers can decide how to apply this criterion. Usually, several solutions can be adopted. For example, the web page content could be structured by using frames, or blocks customizable by CSS properties. Alternatively, the content within the page could be visualized by embedding it in the layout or data tables. Moreover, long page content could be partitioned through heading levels, paragraphs, or specific page parts could marked with "hidden labels." So, all these cases apply the same general guideline (i.e., partitioning the content), but use different technical solutions.

The analysis of web site accessibility and usability by means of guidelines, similar to other inspection methods used in usability/accessibility assessment, requires observing, analyzing, and interpreting the web site characteristics. Since these activities require high costs in terms of time and effort, there is great interest in developing tools that automate them in various phases. However, even the use of automatic tools has problems, such as the length and detailed nature of reports that make them difficult to interpret, accessibility guidelines require developers to fully understand their requirements, and often they still need some manual inspection. Besides, another issue has to be considered when automatic support is used is the "repair process." In fact, even if accessibility and usability problems are detected automatically, repairing them can require a lot of effort. A completely automatic repair process is not easy to implement; a semiautomatic support for this important phase is advisable. However, several common evaluation tools do not provide any support for repair functionality, and those tools that provide some support require working with the underlying HTML code, which can be tedious and difficult to do because of the many low-level details to handle. In this perspective, MAGENTA (Multi-Analysis of Guidelines by an ENhanced Tool for Accessibility) (Leporini, Paternò, and Scorcia, 2006) has been developed to assist developers in handling web pages and guidelines.

Many automatic analysis tools were developed to assist evaluators by automatically detecting and reporting guideline violations and in some cases making suggestions for fixing them. EvalIris (Abascal et al., 2004) is an example of a tool that provides designers and evaluators with a good support to easily incorporate new additional accessibility guidelines. This is obtained through a language for guidelines represented by an XML schema, which has been used to specify WAI guidelines. The result report of the single page evaluation is provided through a representation defined in XML as well. MAGENTA is another tool that supports the verification of guidelines expressed by an XML-based language (GAL: guidelines abstraction language). One advantage of this choice is that the guidelines are specified externally to the tool. Thus, if a new set of guidelines has to be checked, the tool can still be used without modifying its implementation, as long as the new guidelines are specified through such abstraction language. MAGENTA also aims at addressing other ways to support designers, such as providing interactive support for correcting the web pages violating the guidelines considered. Another contribution in this area is DESTINE (Beirekdar et al., 2005), which supports W3C and Section 508 guidelines specified through another language for guidelines and it provides reports that include statistics at different levels (site, page, guideline). However, even this tool does not address the possibility of supporting the repair of web pages. Imergo (Mohamad et al., 2004) is another interesting contribution in the area of tools for accessibility, which aims to provide integrated support for content management systems and engineered support for multiple guidelines. MAGENTA provides a different solution to such an issue, based on the definition of an abstract language for guidelines and support for automatic repair. It allows developers to visualize the reports without performing again the evaluation process because the reports are stored in external XML files.

26.2.4 Markup Languages for Implementation of Interactive Applications

The XML UI language (XUL) is an example of a markup language for creating UIs. It is a part of the Mozilla browser and related applications, and is available as part of Gecko (http://developer.mozilla.org/en/docs/Gecko). With XUL and other Gecko components, developers can create sophisticated applications without special tools. XUL was designed for creating the UI of the Mozilla application, including the web browser, mail client, and page editor. XUL may be used to create these types of applications. However, it may also be used for any web application, for instance, when developers need to be able to retrieve resources from the network and require a sophisticated UI. Like HTML, in XUL developers can create an interface using a markup language, use CSS style sheets to define appearance, and use JavaScript for behavior. Programming interfaces for reading and writing remote content over the network and for calling web services are also available. Unlike HTML, however, XUL provides a powerful set of UI widgets for creating menus, toolbars, tabbed panels, and hierarchical trees to give a few examples. This means that developers do not have to look for third-party code or include a large block of JavaScript in their application just to handle a popup menu. XUL has all of these elements built-in.

In addition, the elements are designed to look and feel just like those on the user's native platform, or designers can use standard CSS to create their own look.

A markup language for vectorial graphics is SVG, which, together with JavaScripts, supports interactive direct manipulation graphical interfaces. The vocal modality is supported through VoiceXML (http://www.w3.org/TR/voicexml20). X+V supports multimodal interfaces by combining XHTML and VoiceXML. SMIL supports multimedia presentations, including video or audio streams combined with texts and images.

26.2.5 Transformations

As previously discussed, XML can be used to formalize various aspects, concepts, and models relevant in HCI. This knowledge often needs to be transformed in design, development, and evaluation. For example, one model can be converted into another model of the same system. A transformation is a set of rules and techniques used for this modification. A description of a transformation may be a rule specified in natural language, an algorithm in an action language, or in a model mapping language. Therefore, there is the problem of transforming different XML-based specifications one into another. There are various types of transformations; for example, abstraction moves toward more abstract representations, and refinement moves toward more concrete representations. Mappings are a specific type of transformation that associates one element of a representation with one or multiple elements of another representation. Section 26.4 will provide a more detailed description of different markup-based approaches for supporting transformations among different XML-based descriptions.

26.3 Existing Markup Languages for Multidevice UIs

Looking at the historical evolution of development software languages (represented in Figure 26.4), one can notice that there is a tendency to increase the abstraction levels to make them more manageable and expressive. In particular, in recent years there has been a focus on device-independent markup languages to address the issues raised by multidevice environments and models (see, e.g., the evolution toward model-driven architectures of UML and related languages).

Souchon and Vanderdonckt (2003) provided an early discussion of various model-based approaches, mainly for multidevice UIs, considering the models supported, the method applied, the tools associated, the implementation languages and the platforms supported, along with the number of tags, the expressivity, and the openness of the environments. Since then many other proposals have been put forward in this area. For example, in TADEUS (Müller, Forbrig, and Cap, 2001) authors present a concept of device-independent UI design based on the XML-based technology. Specifically, it is an approach for integrating task and object knowledge into the development process and its underlying representations, with a special focus on mobile devices. The model-based approach of TADEUS is based on user, task, and business-object models. It allows the computer-based development of interactive systems, with the three models at the basis for the development of the application logic and the UI design.

A UI is considered a simplified version of the MVC model (Burbeck, 1981) and is separated into a model component and a presentation component. The model component (also called abstract interaction model) describes the feature of the UI on an abstract level, while the UI objects with their representation are specified in the presentation component. During the development, a mapping transforming the abstract interaction model to the specific interaction model is necessary. TADEUS uses a representation of both models. The transformation process is described by attribute grammars: for different platforms different grammars are necessary. The XML-based technology is used in this approach for specifying the description of the abstract interaction model, the description of specific characteristics of different devices, and the specification of the transformation process.

In this section it is not possible to mention all the XML-based languages that have been proposed to address the issues raised

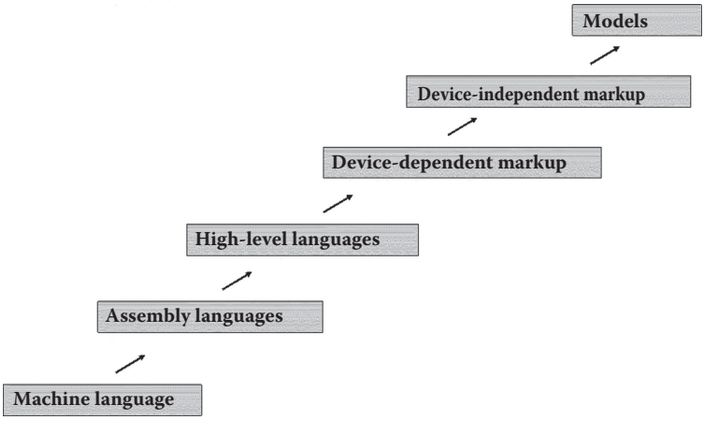

FIGURE 26.4 Historical evolution of software languages.

by multidevice interfaces. Thus, a subset is discussed that should be sufficient to illustrate the relevant issues and the possible solutions.

26.3.1 XIML

The extensible interface markup language (XIML) (http://www.ximl.org) is an extensible XML-based specification language for multiple facets of multiple models in a model-based approach, developed by a forum headed by RedWhale software. It was introduced as a solution that enables a framework for the definition and interrelation of interaction data items. As such, XIML can provide a standard mechanism for applications and tools to interchange interaction data and to interoperate within integrated UI engineering processes, from design, to operation, to evaluation.

The XIML language draws mainly from two foundations. One is the study of ontologies and their representation, and the other is the work on interface models (Szekely et al., 1995; Puerta, 1997; Puerta and Eisenstein, 2001). From the former, XIML draws the representation principles it uses; from the latter, it derives the types and nature of interaction data. The basic structure of XIML consists of components, relations, and attributes:

- *Components*: XIML is an organized collection of interface elements that are categorized into one or more major interface components. The language does not limit the number and types of components that can be defined or the number and types of elements under each component. However, it is expected that an XIML specification would support a relatively small number of components with one major type of element defined per component. In its first version (1.0), XIML predefines five basic interface components, namely task, domain, user, dialogue, and presentation. The first three of these can be characterized as contextual and abstract, while the last two can be described as implementational and concrete:
  - *Task*: The task component captures the user tasks that the interface supports. The component defines a hierarchical decomposition of tasks and subtasks, along with the expected flow among those tasks and the attributes of those tasks. The granularity of tasks is not set by XIML, so valid tasks can range from simple one-step actions (e.g., Enter Date, View Map) to complicated multistep processes (e.g., Perform Contract Analysis). An XML excerpt of a task specified in XIML is shown below:

```
<TASK_MODEL ID='tm1'>
  <TASK_ELEMENT ID='t1' name='Make
  annotation'>
  <TASK_ELEMENT ID='t1.1'
  name='Select location'/>
  <TASK_ELEMENT ID='t1.2'
  name='Enter note'/>
  <TASK_ELEMENT ID='t1.3'
  name='Confirm Annotation'/>
  </TASK_ELEMENT>
</TASK_MODEL>
```

- *Domain*: The domain component is an organized collection of data objects and classes of objects that is structured into a hierarchy. This hierarchy is similar in nature to that of an ontology, but at a very basic level. Objects are defined via attribute-value pairings. Objects to be included in this component are restricted to those that are viewed or manipulated by a user and can be either simple or complex types. For example, "Date," "Map," and "Contract" can all be domain objects. An excerpt of the domain component is shown here:

```
<DOMAIN_MODEL ID='dm1'>
  <DOMAIN_ELEMENT ID='d1.1'
  name='map annotation'>
  <DOMAIN_ELEMENT ID='d1.1.1'
  name='location'/>
  <DOMAIN_ELEMENT ID='d1.1.2'
  name='note'/>
  <DOMAIN_ELEMENT ID='d1.1.3'
  name='entered_by'/>
  <DOMAIN_ELEMENT ID='d1.1.4'
  name='timestamp'/>
  </DOMAIN_ELEMENT>
</DOMAIN_MODEL>
```

- *User*: The user component defines a hierarchy—a tree—of users. A user in the hierarchy can represent a user group or an individual user. Therefore, an element of this component can be "Doctor" or can be "Doctor John Smith." Attribute-value pairs define the characteristics of these users. As defined today, the user component of XIML is expected to capture data and features that are relevant in the functions of design, operation, and evaluation. An XML excerpt for the user component is shown below:

```
<USER_MODEL ID='umode1'>
  <USER_ELEMENT ID='u1.1'
  NAME='field researcher'>
  <USER_ELEMENT ID='u1.1.1'
  NAME='field
  supervisor'/>
  <USER_ELEMENT ID='u1.1.2'
  NAME='field
  geologist'/>
  </USER_ELEMENT>
  <USER_ELEMENT ID='u1.2'
  NAME='analyst'>
</USER_MODEL>
```

  - *Presentation*: The presentation component defines a hierarchy of interaction elements that comprise the concrete objects that communicate with users in an interface. Examples of these are a window, a push button, a slider, or a complex widget such as an ActiveX control to visualize stock data. It is generally intended that the granularity of the elements in the presentation component will be relatively high, so that the logic and operation of an interaction element are separated from its definition. In this manner, the rendering of a specific interaction element can be left entirely to the corresponding target display system.
  - *Dialogue*: The dialogue component defines a structured collection of elements that determine the interaction actions that are available to the users of an interface. For example, a "Click," a "Voice response," and a "Gesture" are all types of interaction actions. The dialogue component also specifies the flow among the interaction actions that constitute the allowable navigation of the UI. This component is similar in nature to the Task component, but it operates at the concrete levels, as opposed to the abstract level of the Task component.
- *Relations*: The interaction data elements captured by the various XIML components constitute a body of knowledge that can support organization and knowledge-management functions for UIs. A relation in XIML is a definition or a statement that links any two or more XIML elements either within one component or across components. By capturing relations in an explicit manner, XIML creates a body of knowledge that can support design, operation, and evaluation functions for UIs. The run-time manipulation of those relations constitutes the operation of the UI. However, XIML does not specify the semantics of those relations, which is left up to the specific applications that utilize XIML.
- *Attributes*: In XIML, attributes are features or properties of elements that can be assigned a value. The value of an attribute can be one of a basic set of data types, or it can be an instance of another existing element. Multiple values are allowed, as well as enumerations and ranges. The basic mechanism in XIML for defining the properties of an element is to create a number of attribute-value pairs for that element. In addition, relations among elements can be expressed at the attribute level or at the element level. As in the case of relations, XIML supports definitions and statements for attributes.

One of the main disadvantages of XIML is that a rather simple notion of task models is supported by this approach, for which tool support is not currently available. Another drawback connected to XIML is the fact that, different from the majority of the other UI description languages, it is developed within a software company, and therefore its use is protected by copyright.

26.3.2 UIML

An example of a language that has addressed the multidevice interface issue is the UI markup language (UIML) (http://www.uiml.org; Abrams et al., 1999; VT Course), an XML-compliant language that supports a declarative description of a UI in a device-independent manner. This has been developed mainly by Harmonia.

In UIML, a UI is simply a set of interface elements with which the user interacts. These elements may be organized differently for different categories of users and families of appliances. Each interface element has data (e.g., text, sounds, images) used to communicate information to the user. Interface elements can also receive information from the user using interface artifacts (e.g., a scrollable selection list) from the underlying application. Since the artifacts vary from appliance to appliance, the actual mapping (rendering) between an interface element and the associated artifact (widget) is done using a style sheet. Run-time interaction is done using events. Events can be local (between interface elements) or global (between interface elements and objects that represent an application's internal program logic; i.e., the backend). Since the interface typically communicates with a backend to perform its work, a run-time engine provides for that communication. The run-time engine also facilitates a clean separation between the interface and the backend.

UIML describes a UI in the following sections (see Figure 26.5): *structure, style, content, behavior* (which compose the interface section) and *presentation* and *logic sections*.

```
<uiml>
    <interface>
      <structure> ...</structure>
      <style> ...</style>
      <content> ...</content>
      <behavior> ...</behavior>
    </interface>
    <peers>
      <logic> ...</logic>
      <presentation>...</presentation>
    </peers>
</uiml>
```

- *Structure*: The <structure> element contains a list of <part> elements. Each <part> element describes some abstract part of a UI and it has at least a class name. Here is an example:

```
<structure>
  <part id="TopLevel" class="JFrame">
    <part id="Label" class="JLabel"/>
    <part id="Button" class="JButton"/>
  </part>
</structure>
```

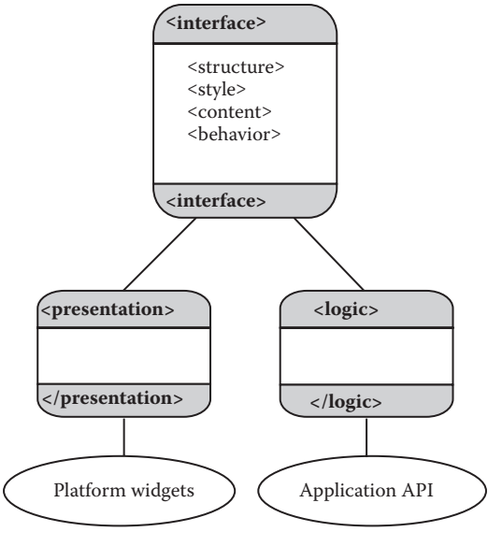

FIGURE 26.5 UIML structure.

The UI excerpt shown contains three parts: one named *TopLevel,* which is an instance of class *JFrame*, one named Label, which is an instance of class *JLabel*, and one named Button, which is an instance of class *JButton*. Note that Label and Button are nested inside *TopLevel*, which implies a relationship between the three parts. If two versions of a UI for two very different devices are considered (such as a mobile phone and a PC), two different <structure> elements will be used. That is because a mobile phone display can only show a few lines of text (e.g., one menu), while a desktop PC could display complex UIs (e.g., containing many pull-down menus).

- *Style*: The <style> element contains a list of <property> elements, giving presentation properties of the parts. Here's an example:

```
<style>
  <property part-name="TopLevel"
  name="background">blue</property>
</style>
```

This excerpt sets property *background* to the value *blue* in the part named *TopLevel*. It is worth noting that UIML itself does not define property names like *background* or values like *blue*—these are part of the vocabulary, whose name is given later in the <presentation> element. When designing a UI for multiple devices, multiple <style> elements can be included. One will represent <style> that is common to all devices; others will represent <style> information that is specific to certain devices. UIML uses a cascade mechanism similar to CSS to combine different <style> elements for a specific device.

- *Content*: The <content> element contains the text, images, and other content that goes into the UI. The <content> element contains one or more scalar values (e.g., a string) or data models (e.g., a list of strings to populate a menu), each identified by a name and referenced in other parts of the UIML document by the <reference> element. One benefit to separating content from structure is that this facilitates internationalization. Another benefit is that different devices need different words in their messages. For instance, a phone with a small display may need a shorter message than a PC, while a voice interface might sound stilted if the same text as a PC's textual message is used. A UIML document could contain different <content> elements for the phone, PC, and voice UIs.
- *Behavior*: The <behavior> element contains a set of rules to define how the UI reacts to a stimulus, either from a user or from the external programming environment. For instance, a user might click a button. The external programming environment might send an asynchronous message that should be displayed in the UI. The following is a simple example:

```
<behavior>
  <rule>
    <condition>
      <event class="actionPerformed"
      part-name="Button"/>
    </condition>
    <action>
      <property part-name="Label"
      name="text">Button pressed.
      </property>
    </action>
  </rule>
</behavior>
```

When the <condition> is satisfied, the <action> is executed. Specifically, the <condition> is satisfied whenever the Java *actionPerformed* listener receives an *actionEvent* on a part whose name suggests that it is a button (*Button*). The previous example uses the Java AWT/Swing vocabulary for UIML. The <action> sets a property named *text* for the part *Label* and changes the text in label *Label* to a say "Button pressed" when the UI part named "Button" is clicked.

- *Logic*: The <logic> element defines the application programming interface (API) of business logic with which the UI interacts. Here is an example of the <*logic*> element:

```
<logic>
  <d-component id="counter" maps-
  to="org.something.simplecounter">
    <d-method id="increment" return-
    type="int" maps-to="count"/>
  </d-component>
</logic>
```

This UIML allows the other parts of UIML to call method *increment()* in a component named *Counter*.

- *Presentation*: The final part is the <presentation> element. In virtually all UIML documents, this simply names a vocabulary file.

  ```
  <presentation base="Java_1.3_
  Harmonia_1.0"/>
  ```

  A UIML implementation reads the UIML vocabulary file, and then interprets all the part class names (e.g., *JLabel, JButton, JFrame*) and property names (e.g., *text*) according to the mappings in the vocabulary file.

One of the main disadvantages of UIML is the fact that it does not support the task level. Some research on how to integrate task models with UIML started some years ago at Virginia Tech (Alí, Perez-Qinones, and Abrams, 2003), but the results have not been incorporated in the Liquid environment supporting UIML. Another shortcoming of UIML is that it provides a single language to define different types of UIs; therefore, there is the need to design separate UIs for each device.

26.3.3 TERESA XML

TERESA XML supports the various possible abstraction levels. The task level, which describes the activities that should be supported by the system, is specified in a hierarchical manner. The temporal relationships occurring between the different tasks are expressed using the CTT notation (Paternò, 1999; Berti et al., 2004). Then, at the abstract level, an abstract UI is composed of a number of presentations and connections among them. While each presentation defines a set of interaction techniques perceivable by the user at a given time, the connections define the dynamic behavior of the UI, by indicating what interactions trigger a change of presentation and what the next presentation is.

There are *abstract* interactors indicating the possible interactions in a platform-independent manner: for instance, the type of interaction to be performed (e.g., selection, editing, etc.) can be indicated without any reference to concrete ways to support such a performance (e.g., selecting an object through a radio button or a pull-down menu, etc.). This level also describes how to compose such basic elements through some composition operators. Such operators can involve one or two expressions, each of them can be composed of one or several interactors or, in turn, compositions of interactors. In particular, the composition operators have been defined taking into account the type of communication effects that designers aim to achieve when they create a presentation (Mullet and Sano, 1995). They are:

- *Grouping*: Indicates a set of interface elements logically connected to each other.
- *Relation*: Highlights a one-to-many relation among some elements, one element has some effects on a set of elements.
- *Ordering*: Some kind of ordering among a set of elements can be highlighted.
- *Hierarchy*: Different levels of importance can be defined among a set of elements.

It is also worth noting that, at the abstract level, some work has also been conducted regarding the issue of how to connect the interactive part of a software application with the functional core, namely the set of application functionalities independent of the media and the interaction techniques used to interact with the user. This aspect is very relevant in addressing the problem of generating dynamic pages with TERESA.

The *concrete* level is a refinement of the abstract UI: depending on the type of platform considered, there are different ways to render the various interactors and composition operators of the abstract UI. The concrete elements are obtained as a refinement of the abstract ones. For example, a navigator (an abstract interactor) can be implemented either through a textlink, or an imagelink, or a simple button, and in the same way, a single choice object can be implemented using either a radio button or a list box or a dropdown menu. The same holds for the operators; for example, the grouping operator in a desktop platform can be refined at the concrete level by a number of techniques, including both unordered lists by row and unordered lists by column (apart from classical grouping techniques such as fieldsets,

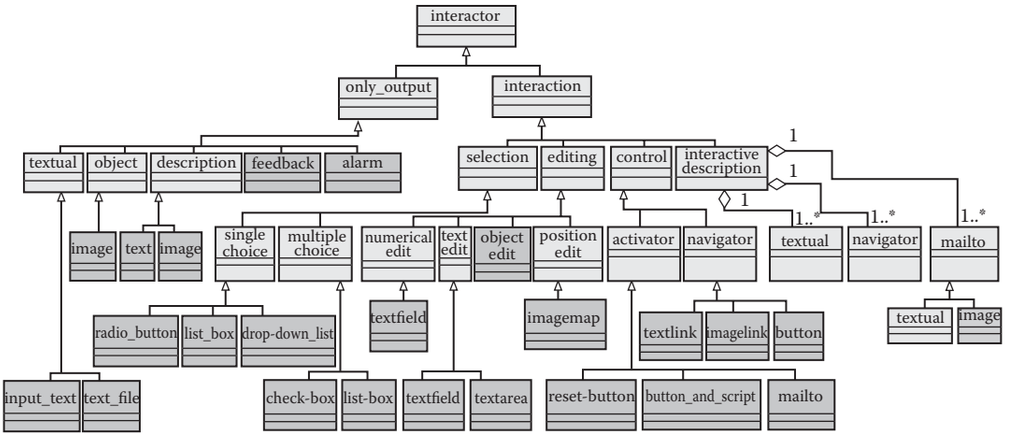

FIGURE 26.6 An excerpt of the concrete user interface for the graphical desktop.

bullets, and colors). The small capability of a mobile phone does not allow implementing the grouping operator by using an unordered list of elements by column; therefore, this technique is not available on this platform. In a vocal device, a grouping effect can be achieved through inserting specific sounds or pauses or using a specific volume or keywords.

In the following, excerpts are reported of the concrete UI languages specified in TERESAXML considering two platforms: desktop and mobile. They are both obtained as a refinement of the abstract language. Such refinements are added at the lower levels of the hierarchical structure of the languages. As it is possible to see from the following excerpts, the desktop CUI is defined by a number of presentations and settings, which are used for the page generation. The mobile devices need additional information concerning their capabilities.

Each presentation contains interactors and interactor compositions, the connections to other presentations, and provides a description of specific properties, such as title, header, background, and so on.

Considering more in detail the interactors specification, one can see how it is possible to specify a "single selection" interactor in the CUI-desktop.

The main differences in the concrete languages are in the concrete interactors associated with an abstract one. For example, considering a "single selection," it can be observed that in a mobile device the "list boxes" are not suitable.

Each concrete language takes into account the interaction resources available. Thus, for example, the grouping composition can be supported in several ways in the desktop, but only a subset is supported in mobile devices.

One advantage of this approach based on multiple levels of abstraction is that all the concrete interface languages share the same structure and add concrete platform-dependent details to the abstract language regarding the possible attributes for implementing the logical interaction objects and the ways to compose them. All languages in this approach, for any abstraction level, are defined in terms of XML to make them more easily manageable and to allow for their export/import in different tools.

Another advantage of this approach is that maintaining links among the elements in the various abstraction levels allows for the possibility of linking semantic information (such as the activity that users intend to do) and implementation levels, which can be exploited in many ways. A further advantage is that designers of multidevice interfaces do not have to learn all the details of the many possible implementation languages, because the environment allows them to have full control over the design through the logical descriptions and leave the implementation to an automatic transformation from the concrete level to the target implementation language. In addition, if a new implementation language needs to be addressed, the entire structure of the environment does not change, but only the transformation from the associated concrete level to the new language has to be

```
<!ELEMENT concrete_desktop_interface    (default_settings, presentation+)>
<!ELEMENT default_settings     (background, font_settings, operators_settings)
<!ELEMENT concrete_mobile_interface    (device_type, default_settings, presentation+)>
<!ELEMENT device_type (big | medium | small)>
<!ELEMENT big EMPTY>
<!ATTLIST big
    graphic_support (%option;) #REQUIRED>
<!ELEMENT medium EMPTY>
<!ATTLIST medium
    graphic_support (%option;) #REQUIRED>
<!ELEMENT small EMPTY>
<!ATTLIST small
    graphic_support CDATA #FIXED "no">
```

```
<!ELEMENT presentation     (presentation_properties, connection*, (interactor |
                           interactor_composition))>
<!ELEMENT presentation_properties    (title, background, font_settings, top)>
```

```
<!ELEMENT interaction   (selection | editing | control | interactive_description)>
<!ELEMENT selection   (single | multiple)>
<!ELEMENT single   (radio_button | list_box | drop_down_list)>
<!ATTLIST single
    cardinality (%cardinality_value;) #REQUIRED>
<!ELEMENT radio_button (choice_element+)>
<!ATTLIST radio_button
    label CDATA #REQUIRED
    alignment (%element_selection_alignment;) #REQUIRED>
<!ELEMENT choice_element EMPTY>
<!ATTLIST choice_element
    label CDATA #REQUIRED
    value CDATA #REQUIRED>
```

```
<!ELEMENT single (radio_button | drop_down_list)>
```

```
CUI-desktop
<!ELEMENT grouping (fieldset?, bullet?, background_color?,   position)>
```

```
CUI-mobile
<!ELEMENT grouping ((fieldset | bullet), position)>
```

added. This is not difficult because the concrete level is already a detailed description of how the interface should be structured.

26.3.4 UsiXML

The semantics of the UsiXML models (Limbourg et al., 2004; Vanderdonckt, 2005) are based on meta-models expressed in terms of UML class diagrams, from which the XML schema definition are derived. Currently, there is no automation between the initial definition of the semantics and their derivation into XML schemas. Only a systematic method is used for each new release.

All model-to-model transformations are themselves specified in UsiXML to keep only one UIDL throughout the development life cycle. Model-to-code transformation is ensured by appropriate tools that produce code for the target context of use or platform. For reverse engineering, code-to-model transformations are mainly achieved by derivation rules that are based on the mapping between the meta-model of the source language (e.g., HTML) and the meta-model of the target language (i.e., UsiXML).

In UsiXML, a concrete UI model consists of a hierarchical decomposition of CIOs. A concrete interaction object (CIO) is defined as any UI entity that users can perceive, such as text, image, or animation, and/or manipulate, such as a push button, a list box, or a check box. A CIO is characterized by various attributes such as, but not limited to: ID, name, icon, content, defaultContent, defaultValue. Each CIO can be subtyped into sub-CIOs depending on the interaction modality chosen: graphicalCIO for GUIs, auditoryCIO for vocal interfaces, 3DCIO for 3D UIs, and so on. Each graphicalCIO inherits properties from its parent CIO and has specific attributes such as: isVisible, isEnabled, fgColor,

and bgColor to depict foreground and background colors, and so on. Each graphicalCIO is then subtyped into one of the two possible categories: graphicalContainer, for all widgets containing other widgets such as page, window, frame, dialogue box, table, box and their related decomposition, or graphicalIndividualComponent, for all other traditional widgets that are typically found in such containers. A graphical IndividualComponent cannot be further decomposed. The model supports a series of widgets defined as graphicalIndividualComponents such as: textComponent, videoComponent, imageComponent, imageZone, radioButton, toggleButton, icon, checkbox, item, comboBox, button, tree, menu, menuItem, drawingCanvas, colorPicker, hourPicker, datePicker, filePicker, progressionBar, slider, and cursor. Thanks to this progressive inheritance mechanism, each final element of the concrete UI inherits properties from these levels, depending on the category it belongs to. The properties populating the concrete level of UsiXML have been chosen because they belong to the intersection of the property sets of major UI toolkits, such as Windows GDI, Java AWT and Swing, HTML. In this way, a CIO can be specified independently from the fact that it will be further rendered in HTML, VRML, or Java (this quality is often referred to as the property of *implementation language independence*).

UsiXML aims to address the development of UIs for multiple contexts of use, and has the advantage of providing a graphical syntax for a majority of constituent models. Also, unlike XIML, UsiXML's language specifications are freely available and not protected by copyright. However, UsiXML renderers are still at a development stage, and cannot compete with the numerous implementations of UIML.

26.3.5 Pebbles

In the Pebbles project at CMU University in Pittsburgh (USA), a personal universal controller (PUC) environment has been developed, which supports the downloading of logical descriptions of appliances and the automatic generation of the corresponding UIs (Nichols et al., 2002).

A PUC engages in two-way communication with everyday appliances, first downloading a specification of the appliance's functions, and then automatically creating an interface for controlling them. The specification of each appliance includes a high-level description of every function, a hierarchical grouping of those functions, and a description of the availability of each function relative to the appliance's state. The PUC architecture has four parts: appliance adaptors, a specification language, a communication protocol, and interface generators.

The assumption is that each appliance has its own facility for accepting connections from a PUC. The peer-to-peer characteristic of the PUC architecture allows a more scalable solution than other systems, such as ICrafter (Ponnekanti et al., 2001) and UIA (Eustice et al., 1999), which rely on a central server to manage connections between interfaces and appliances. A PUC could discover appliances by intermittently sending out broadcast requests. To connect the PUC to any real appliance, an appliance adaptor has to be built. An adaptor should be built for each proprietary appliance protocol that the PUC communicates with. To make the construction of new appliance adaptors easy, an adaptor development framework has been created. It is implemented entirely in Java, and manages all PUC communication for the adaptor, allowing the programmer to concentrate on communicating with the appliance.

A description of an appliance's functions is available, so that the PUC can automatically generate an interface. This description does not contain any information about look or feel: decisions about look and feel are left up to each interface generator. The PUC specification language is XML based. Some of the key information contained are:

- *State variables and commands*: Interface designers must know what can be manipulated on an appliance before they can build an interface for it. Therefore, the manipulable elements are represented as state variables. Each state variable has a given type that tells the interface generator how it can be manipulated. Commands are also useful when an appliance is unable to provide feedback about state changes back to the controller, either by manufacturer choice or a hardware limitation of the appliance.
- *Type information*: Each state variable must be specified with a type so that the interface generator can understand how it may be manipulated. In PUC, seven generic types may be associated with a state variable: Boolean, integer, fixed point, floating point, enumerated, string, custom.
- *Label information*: The interface generator must also have information about how to label the interface components that represent state variables and commands. Providing this information might be difficult, because different interface modalities require different kinds of label information. In PUC this information is provided with a generic structure called the label dictionary, in which every label contained, whether it is phonetic information or plain text, will have approximately the same meaning. Thus the interface generator can use any label within a label dictionary interchangeably.
- *Dependency information*: The two-way communication feature of the PUC allows it to know when a particular state variable or command is unavailable. This can make interfaces easier to use, because the components representing those elements can be disabled. The specification contains formulas that specify when a state or command will be disabled depending on the values of other state variables, currently specified with three types of dependencies: equal-to, greater-than, and less-than. Each state or command may have multiple dependencies associated with it, combined with the logical operations AND and OR. These formulas can be processed by the PUC to determine whether a component should be enabled when the appliance state changes.

Following is an excerpt of specification with PUC for a to-do list application (see PUC Documentation):

```
<spec>
  <groupings>
    <group name="Commands" is-a="list-
    commands" priority="10">
    <command name="Add" is-a="list-add"
    priority="8">
      <labels>
        <label>Add To-Do Item</label>
        <label>Add To-Do</label>
        <label>Add</label>
      </labels>
    </command>

<command name="Delete" is-a="list-remove"
priority="7">
  <labels>
    <label>Delete To-Do Item</label>
    <label>Delete To-Do</label>
    <label>Delete</label>
  </labels>
  <active-if>
    <greaterthan state="ToDo.List.
    Length"><constantvalue="0"/>
    </greaterthan>
    <defined state="ToDo.List.Selection"/>
    </active-if>
  </command>
<state name="SortBy" priority="5">
  <type>
  <enumerated>
    <item-count>3</item-count>
  </enumerated>
  <value-labels>
    <map index="1">
      <labels>
        <label>Category</label>
      </labels>
    </map>
    <map index="2">
      <labels>
        <label>Completion Date</label>
        <label>Date</label>
      </labels>
    </map>
    <map index="3">
      <labels>
        <label>Completed</label>
      </labels>
    </map>
  </value-labels>
</type>
<labels>
  <label>Sort By</label>
          <label>Sort</label>
        </labels>
        <active-if>
          <greaterthan state="ToDo.List.
          Length">
            <constant value="0"/>
          </greaterthan>
        </active-if>
      </state>
    </group>
    <group>
    …
    </group>
  </groupings>
</spec>
```

In this XML excerpt there is a grouped expression with two commands (Add and Delete) that can be used on the manipulable elements (actions in the to-do list) and also a state for sorting them, depending on different criteria: category, completion date, completed. The specification contains also formulas that specify when a state or command will be disabled depending on the values of other state variables: for instance, as it appears in the previous specification, the Delete command is available only if the length of the to-do list is greater than 0.

The communication protocol enables appliances and PUC adaptors to exchange information bidirectionally and asynchronously. The protocol is XML-based and defines six messages, two sent by the appliance and four sent by the PUC. The PUC can send messages requesting the specification, the value of every state, a change to a particular state, or the invocation of a command. The appliance can send the specification or the current value of a particular state. When responding to a PUC request for the value of every state, the appliance will send a current value message for each of its states.

The PUC architecture has been designed to be independent of the type of interface presented to the user. Generators have been developed for two different types of interfaces: graphical and speech. The graphical interface generator takes an arbitrary description written in the PUC specification language and makes use of the group tree (specifying the states variables and commands to be included in the UI) and dependency information (containing formulas specifying when a state or a command will be disabled depending on the values of other state variables) to create the related UI. The actual UI components that represent each state variable and command are chosen using a decision tree; the components are then placed into panels according to the inferred structure, and laid out using the group tree; the final step of the generation process instantiates the components and places them on the screen.

The speech interface generator creates an interface from a PUC specification using USI (universal speech interface) interaction techniques. The speech-interface generator differs from the graphical interface in that it connects to multiple PUC adaptors (if multiple adaptors can be found on the network), requests the device specifications, and then uses those specifications collectively to configure itself so that it can control all devices with

the same speech grammar. This includes building a grammar for the parser, and a language model and pronunciation dictionary for the recognizer.

One of the main disadvantages of this approach is that its application area is limited to home appliances that require similar interfaces.

26.3.6 XForms

XForms is an XML application that represents the next generation of forms for the web, and has introduced the use of abstractions to address new heterogeneous environments. The primary difference when comparing XForms with HTML Forms, apart from XForms being in XML, is the separation of the data being collected from the markup of the controls collecting the individual values. This not only makes XForms more tractable, by making it clear what is being submitted and where, but it also eases reuse of forms, since the underlying essential part of a form is no longer bound to the page it is used in. A second major difference is that XForms, while designed to be integrated into XHTML, is no longer restricted only to be a part of that language, but may be integrated into any suitable markup language. In the XForms approach, forms are comprised of a section that describes what the form does, called the XForms model, and another section that describes how the form is to be presented.

By splitting traditional XHTML forms into three parts—XForms model, instance data, and UI—it separates presentation from content, allows reuse, gives strong typing, reduces the number of roundtrips to the server, as well as offers device independence and a reduced need for scripting.

- *Model*: The data layer that describes the form's data and logic. In other words, it is the nonvisible definition of an XML form as specified by XForms.
- *Instance data*: An internal tree representation of the values and state of all the instance data nodes associated with a particular form.
- *UI items*: The presentation layer that lets the user input data that will be stored in the data layer. XForms UI items are used to display data from the XForms model.

XForms creates a separate data layer inside the form, allowing for collecting data from the form and copying the data into a separate block of XML that can be formatted as preferred. Separating the data layer from the presentation layer makes XForms device independent. The data model can be used for all devices. The presentation can be customized for different UIs, like mobile phones, handheld devices, and Braille readers for the blind. Since XForms is device independent and based on XML, it is also possible to add XForms elements directly into other XML applications, such as VoiceXML (speaking web data) and SVG. UI controls encapsulate all relevant metadata such as labels, thereby enhancing accessibility of the application when using different modalities. XForms UI controls are generic and suited for device independence. Therefore, the high-level nature of the UI controls, and the consequent intent-based authoring of the UI, make it possible to retarget the user interaction to different devices.

In particular, XForms aims to separate presentation from content through the definition of a set of platform-independent, general-purpose controls, and focus on the goal (or intent) behind each form control. Indeed, the list of XForms controls includes objects such as select (choice of one or more items from a list), trigger (activating a defined process), output (display-only of form data), secret (entry of sensitive information), and so on, rather than refer to concrete examples like radio buttons, checkboxes, and so forth, which are hardwired to specific representations of such controls. This kind of logical description locates the types of abstractions supported by XForms at the abstract UI level. However, the task level is not explicitly addressed. In addition, XForms is basically aimed at expressing form-based UIs and less for supporting other modalities (e.g., vocal).

26.4 Transforming Markup-Language-Based Specifications

There are various solutions for transforming different XML-based specifications one into another. One solution is represented by XSLT (http://www.w3.org/TR/xslt), which is an XML-based specification allowing the transformation of a XML document into another format. This format can be XHTML, XML, or anything else. Consequently, XSLT could be used to transform models. Nevertheless, it appears difficult to directly use XSLT on a real system for some reasons:

- Writing an XSLT program is long and tedious.
- One important problem with XSLT is its poor readability and the high cost of maintenance for associated programs.
- Executing an XSLT program is not user-friendly for models transformation. There are no error messages that depend on the application domain. For example, the XSLT processor does not inform the user about nonexisting concepts.

An example of an approach using XSLT transformations is TADEUS (Müller, Forbrig, and Cap, 2001). The creation of the XSL-based model description is based on the knowledge of available UIOs for specific representations. It is necessary to know which values of properties of a UIO are available in the given context. The XML-based device-dependent abstract interaction model (skeleton) and available values of properties are used to create an XSL-based model description specifying a complete UI. In addition, by XSL transformation a file describing a specific UI will be generated. The XSL transformation process consists of two subprocesses: (1) creation of a specific file representing the UI (WML, VoiceXML, HTML, Java, etc.) and (2) integration of content (database, application) into the UI.

Another approach is based on graph transformations (GT) techniques that has been developed in UsiXML (Limbourg et al., 2004) to formalize explicit transformations between any pair of models (except for the implementation level). The reasons

for this choice were that it is (1) visual: every element within a GT-based language has a graphical syntax; (2) formal: GT is based on a sound mathematical formalism (algebraic definition of graphs and category theory); and (3) it enables verifying formal properties on represented artifacts. Furthermore, the formalism applies equally to all levels of abstraction of UIs. The implementation level is the only model that cannot be supported by graph transformations, because it would have supposed that any markup or programming language to be supported should have been expressed in a meta-model to support transformations between meta-models: the one of the initial language to the one of the desired specification language. It was observed that to address this problem, the powerfulness of GT techniques was not needed and surpassed by far other experienced techniques, such as derivation rules.

To support the manipulation of models in UsiXML, two software tools were developed: TransformiXML (Limbourg et al., 2004) and IdealXML (Montero et al., 2005). TransformiXML consists of a Java application that triggers transformations of models expressed by graph grammars. However, TransformiXML takes a long time for its processing and the performance is slow. IdealXML is a Java-based application containing the graphical editor for the task model, the domain model, and the abstract model. It can also establish any mapping between these models either manually (by direct manipulation) or semiautomatically (by calling TransformiXML).

Another approach is followed in the TERESA tool (Mori, Paternò, and Santoro, 2004), which supports various transformations: from task-model-related information to abstract UI, from abstract UI to concrete interface for the specific platform, and, finally, an automatic UI generation.

In the first transformation, the goal is transforming the task-based specification of the system into an interactor-based description of the corresponding abstract UI. It is worth pointing out that within TERESA it is also possible to access the inverse transformation, since for each interactor the tool is able to automatically identify and highlight the related task, so that designers can immediately spot the relation. Both the static arrangement of interactions in the same presentation and the dynamic behavior of the abstract UI are derived by analyzing the semantic of the temporal operators included in the task model specification. For instance, a sequential operator between two tasks implies that the related presentations will be sequentially triggered: this will be rendered, at the abstract level, by associating a connection between two different abstract presentations (with each presentation supporting the performance of just one task), so that the performance of the first task will trigger the activation of the second presentation and render the sequential ordering. On the contrary, a concurrency operator between two tasks implies that the associated interactors is presented at the same time, so as to support concurrency between the connected activities; therefore, the abstract objects supporting their performance will be included in the same presentation.

The transformation from abstract UI to concrete UI represents an example of a model-to-model mapping, as it allows for mapping one abstract UI model to a number of related concrete UI models. Each concrete UI specification is associated with a specific interaction platform (e.g., the desktop, the mobile platform, the vocal platform, the digital TV, etc.). The generated concrete UIs can be further customized according to a number of parameters made available to the designer. For instance, one can consider an abstract interactor supporting the task of selecting an item for a set of possible elements: this abstract interactor can be mapped onto a radio button, or a pull-down menu, or a vocal selection (on a vocal platform).

The last transformation from concrete UI to implementation is a model-to-text mapping connecting a concrete UI for a specific platform to one or multiple types of final UIs, depending on a particular selected implementation language. Currently, transformations for implementations in a number of languages (XHTML, XHTML MP, VoiceXML, X+V, Java for Digital TV, JSP) are available, and others are under development.

However, the problem of such transformations is that they are hardcoded within the TERESA tool. A first solution to overcome this problem is to introduce a declarative mechanism to specify mappings between models that can be dynamically used. Indeed, in this approach, XSLT processing can be used to support model-to-model transformation. This solution implies writing a single XSLT transformation that will accept (1) a generic transformation and (2) configuration parameters (both in XML) as source trees and produce a specialized transformation as the result tree. With this approach it is possible to define mappings between abstract and concrete UI specifications (model-to-model transformation) and between concrete and implementation UI models (model-to-text transformations). The objective of this approach is to be the most general possible.

Another type of transformation supports the inverse process—abstraction. It uses reverse engineering techniques able to take the UI of existing applications for any platform and then build the corresponding logical descriptions. Early work in reverse engineering for UIs was motivated by the need to support maintenance activities aiming to reengineer legacy systems for new versions using different UI toolkits (Moore, 1996), in some cases even supporting migration from character-oriented UIs to graphical UIs. More recently, interest in UI reverse engineering has received strong impetus from the advent of mobile technologies and the need to support multidevice applications. To this end, a good deal of work has been dedicated to UIs reverse engineering to identify corresponding meaningful abstractions. Other studies have investigated how to derive the task model of an interactive application starting with the logs generated during user sessions (Hudson et al., 1999). However, this approach is limited to building descriptions of how the UI was actually used, which is described by the logs, but is not able to provide a general description of the tasks supported, which includes even those not considered in the logs. Previous work in reverse engineering has addressed only one level or platform at a time. For example, ReversiXML (Bouillon, Vanderdonckt, and Chow, 2004) has focused on creating a concrete/abstract UI from HTML pages for desktop systems. WebRevenge (Paganelli and Paternò, 2002)

has addressed the same types of applications to build only the corresponding task models.

In general, there is a lack of approaches able to address different platforms, especially involving different interaction modalities, and to build the corresponding logical descriptions at different abstraction levels. ReverseAllUIs (Bandelloni, Paternò, and Santoro, 2007) aims to overcome this limitation.

26.5 Conclusions

This chapter has discussed the many aspects that can be formalized through markup languages (based on XML) in HCI. The current trend is to further exploit such tools to obtain more intelligent environments and better support interoperability. One important aspect in the design of a markup language is to identify the relevant information, avoiding a plethora of low-level details that complicate its processing. Another important issue is the availability of tools able to support the editing, application, and transformation of markup languages. Tools need to be implemented in such a way that if modifications to the language are made, then the tools that exploit their information do not require making profound changes in the implementation.

Lastly, the challenges raised by ambient intelligence call for solutions able to model not only the interdependencies and configuration options of ambient technologies, but also their behavior, privacy/visibility effects, and reliability in order to provide a rich description of such environments.

References

Abascal, J., Arrue, M., Fajardo, I., Garay, N., and Tomás, J. (2004). Use of guidelines to automatically verify Web accessibility. *Universal Access in the Information Society* 3: 71–79.

Abrams, M., Phanouriou, C., Batongbacal, A., Williams, S., and Shuster, J. (1999). UIML: An appliance-independent XML user interface language, in the *Proceedings of the 8th International World Wide Web Conference,* 11–14 May 1999, Toronto, Canada. http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html.

Alí, F., Perez-Qinones, M., and Abrams, M. (2003). Building MultiPlatform user interfaces with UIML, in *Multiple User Interfaces: Cross-Platform Applications and Context-Aware Interfaces* (A. Seffah and H. Javahery, eds.), pp. 95–118. New York: John Wiley & Sons.

AQ2 Bandelloni, R., Paternò, F., and Santoro, C. (2007). Reverse engineering cross-modal user interfaces for ubiquitous environments, in the *Proceedings of the Engineering Interactive Systems 2007 (EIS 2007)*, 22–24 March 2007, Salamanca, Spain.

Beirekdar,A.,Keita,M.,Noirhomme,M.,Randolet,F.,Vanderdonckt, J., and Mariage, C. (2005). Flexible reporting for automated usability and accessibility evaluation of web sites, in the *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction 2005 (INTERACT 2005)*, pp. 281–294. Berlin/Heidelberg: Springer-Verlag.

Berti, S., Correani, F., Paternò, F., and Santoro, C. (2004). The TERESA XML language for the description of interactive systems at multiple abstraction levels, in the *Proceedings of the Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages 2004*, May 2004, Gallipoli, Italy, pp. 103–110. http://giove.isti.cnr.it/cameleon/cp25.html.

Bouillon, L., Vanderdonckt, J., and Chow, K. C. (2004). Flexible re-engineering of Web sites, in the *Proceedings of the 8th ACM International Conference on Intelligent User Interfaces (IUI'2004)*, 13–16 January 2004, Madeira Island, Portugal, pp. 132–139. New York: ACM Press. AQ3

Burbeck, S. (1987). *Applications Programming in Smalltalk- 80: How to Use Model-View-Controller (MVC).* Softsmarts.

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. (2003). A unifying reference framework for multi-target user interfaces. *Interacting with Computers* 15: 289–308.

Eustice, K. F., Lehman, T. J., Morales, A., Munson, M. C., Edlund, S., and Guillen, M. (1999). A universal information appliance. *IBM Systems Journal* 38: 575–601.

Heckmann, D. and Krueger, A. (2003). A user modeling markup language (UserML) for ubiquitous computing, in the *Proceedings of the 9th International Conference on User Modeling 2003 (UM 2003)*, 22–26 June 2003, Johnstown, PA, pp. 393–397. Berlin/Heidelberg: Springer-Verlag.

Hudson, S., John, B., Knudsen, K., and Byrne, M. (1999). A tool for creating predictive performance models from user interface demonstrations, in the *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology (UIST'99)*, 7–10 November 1999, Asheville, NC, pp. 93–102. New York: ACM Press.

Leporini, B., Paternò, F., and Scorcia, A. (2006). Flexible tool support for accessibility evaluation. *Interacting with Computers* 18: 869–890.

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and Lopez, V. (2004). UsiXML: A language supporting multi-path development of user interfaces, in the *Proceedings of the 9th IFIP Working Conference on Engineering for Human-Computer Interaction Jointly with 11th International Workshop on Design, Specification, and Verification of Interactive Systems (EHCI-DSVIS'2004)*, 11–13 July 2004, Hamburg, Germany, pp. 200–220. Berlin/Heidelberg: Springer-Verlag.

Mohamad, Y., Stegemann, D., Koch, J., and Velasco, C. A. (2004). Imergo: Supporting accessibility and Web standards to meet the needs of the industry via process-oriented software tools, in the *Proceedings of the 9th International Conference on Computers Helping People with Special Needs (ICCHP 2004)* (K. Miesenberger, J. Klaus, W. Zagler, and D. Burger, eds.), 7–9 July 2004, Paris, France, pp. 310–316. Berlin/Heidelberg: Springer-Verlag.

Montero, F., Jaquero, V. L., Vanderdonckt, J., Gonzalez, P., Lozano, M. D., and Limbourg, Q. (2005). Solving the mapping problem in user interface design by seamless integration

in IdealXML, in the *Proceedings of the 12th International Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'2005)* (M. Harrison, ed.), 13–15 July 2005, Newcastle, U.K., pp. 161–172. Berlin/Heidelberg: Springer-Verlag.

AQ4 Moore, M. M. (1996). Representation issues for reengineering interactive systems. *ACM Computing Surveys* 28.

Mori, G., Paternò, F., and Santoro, C. (2004). Design and development of multi-device user interfaces through multiple logical descriptions. *IEEE Transactions on Software Engineering* 30: 507–520.

Müller, A., Forbrig, P., and Cap, C. (2001). Model-based user interface design using markup concepts, in the *Proceedings of the 8th International Workshop on Interactive Systems: Design, Specification, and Verification (DSV-IS 2001)*, 13–15 June, Glasgow, Scotland, pp. 16–27. Berlin/Heidelberg: Springer-Verlag.

Mullet, K. and Sano, D. (1995). *Designing Visual Interfaces*. Upper Saddle River, NJ: Prentice Hall.

AQ5 Nichols, J., Myers, B. A., Higgins, M., et al. (2002). Generating remote control interfaces for complex appliances, in the *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2002)*, 27–30 October 2002, Paris, France, pp. 161–170. New York: ACM Press.

Paganelli, L. and Paternò, F. (2002). Automatic reconstruction of the underlying interaction design of Web applications, in the *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)*, 15–19 July 2002, Ischia, Italy, pp. 439–445. New York: ACM Press.

Passani, L. (2005). *Welcome to the WURFL the Wireless Universal Resource File*. http://wurfl.sourceforge.net.

Paternò, F. (1999). *Model-Based Design and Evaluation of Interactive Application*. Berlin/ Heidelberg: Springer Verlag.

Ponnekanti, S. R., Lee, B., Fox, A., Hanrahan, P., and Winograd, T. (2001). ICrafter: A service framework for ubiquitous computing environments, in the *Proceedings of the International Conference on Ubiquitous Computing (UBICOMP 2001),* 30 September–2 October 2001, Atlanta, pp. 56–75. Berlin/ Heidelberg: Springer-Verlag.

PUC Documentation (n.d.). http://www.pebbles.hcii.cmu.edu/puc/specification.html#examplespec.

Puerta, A. (1997). A model-based interface development environment. *IEEE Software* 14: 40–47.

Puerta, A. and Eisenstein, V. (2001). XIML: A common representation for interaction data, in the *Proceedings of the 7th International Conference on Intelligent User Interfaces (IUI 2001)*, 13–16 January 2001, Madeira Island, Portugal, pp. 214–215. New York: ACM Press. http://people.csail.mit.edu/jacobe/papers/mui_chapter.pdf.

Souchon, N. and Vanderdonckt, J. (2003). A review of XML-compliant user interface description languages, in the *Proceedings of 10th International Conference on Design, Specification, and Verification of Interactive Systems (DSV-IS 2003)*, 11–13 June 2003, Madeira Island, Portugal, pp. 377–391. http://www.isys.ucl.ac.be/bchi/publications/2003/Souchon-DSVIS2003.pdf.

Szekely, P. (1996). Retrospective and challenges for model-based interface development, in the *Proceedings of the 3rd International Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96),* pp. 1–27. Vienna: Springer-Verlag.

Szekely, P., Sukaviriya, P., Castells, P., Muthukumarasamy, J., and Salcher, E. (1995). Declarative interface models for user interface construction tools: The MASTERMIND approach, in *Engineering for Human-Computer Interaction* (L. J. Bass and C. Unger, eds.), pp. 120–150. London: Chapman & Hall.

UAProf (2001). *Wireless Application Protocol.* http://www.openmobilealliance.org/tech/affiliates/wap/wap-248-uaprof-20011020-a.pdf.

Vanderdonckt, J. (2005). A MDA-compliant environment for developing user interfaces of information systems, in the *Proceedings of the 16th Conference on Advanced Information Systems Engineering (CAiSE 2005),* pp. 16–31. Berlin/ Heidelberg: Springer-Verlag.

VT Course (n.d.). *Virginia Tech Master Course on Internet Software: UIML Lesson.* http://mit.iddl.vt.edu/courses/cs5244/coursecontent/mod5/lesson2/uiml5.html.

Author Queries

AQ1: Please reframe the sentence, since the box text is placed on the next page.
AQ2: Please provide publisher and publisher location for Bandelloni et al. (2007).
AQ3: Please provide complete publication information (i.e., missing publisher location) for Burbecy (1987).
AQ4: Please provide missing page range for Moore (1996).
AQ5: Please provide all authors up to six before using "et al." for Nichols et al. (2002).