MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments

FABIO PATERNO', CARMEN SANTORO, and LUCIO DAVIDE SPANO
ISTI-CNR

One important evolution in software applications is the spread of service-oriented architectures in ubiquitous environments. Such environments are characterized by a wide set of interactive devices, with interactive applications that exploit a number of functionalities developed beforehand and encapsulated in Web services. In this article, we discuss how a novel model-based UIDL can provide useful support both at design and runtime for these types of applications. Web service annotations can also be exploited for providing hints for user interface development at design time. At runtime the language is exploited to support dynamic generation of user interfaces adapted to the different devices at hand during the user interface migration process, which is particularly important in ubiquitous environments.

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*User-centered design; interaction styles; theory and methods*

General Terms: Design, Experimentation, Human Factors

Additional Key Words and Phrases: Model-based design, user interface description language, ubiquitous applications, multidevice user interfaces, Web services

ACM Reference Format:

Paterno', F., Santoro, C., and Spano, L. D. 2009. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. ACM Trans. Comput.-Hum. Interact. 16, 4, Article 19 (November 2009), 30 pages.
DOI = 10.1145/1614390.1614394 http://doi.acm.org/10.1145/1614390.1614394

1. INTRODUCTION

Model-based approaches are a well-known area in HCI. They rely on a number of models in which the relevant aspects of a user interface can be specified and

This work has been supported by the OPEN (http://www.ict-open.eu) and ServFace (http://www.servface.eu) ICT EU Projects.
Authors' addresses: F. Paterno' (contact author), C. Santoro, L. D. Spano, ISTI-CNR, HIIS Laboratory, Via Giuseppe Moruzzi, 1 56124 Pisa, Italy; email: fabio.paterno@isti.cnr.it.

DOI 10.1145/1614390.1614394 http://doi.acm.org/10.1145/1614390.1614394

manipulated, so as to ease the work of designers and developers. In recent years they have evolved in parallel with the aim of coping with the different challenges raised by the design and development of user interfaces in continuously evolving technological settings. We can identify a first generation of model-based approaches in which the focus was basically on deriving abstractions for graphical user interfaces (see, for example, UIDE [Foley et al. 1994]). At that time, user interface designers focused mainly on identifying relevant aspects for this kind of interaction modality. Then, the approaches evolved into a second generation focusing on expressing the high-level semantics of the interaction: This was mainly supported through the use of task models and associated tools, aimed at expressing the activities that the users intend to accomplish while interacting with the application (see, for example, Adept [Johnson et al. 1993], GTA [van der Veer et al. 1994], ConcurTaskTrees (CTT) [Paterno' 1999]). Then, thanks to the growing affordability of new interactive platforms, in particular mobile ones, the work of UI designers mainly focused on how to cope with the relentless appearance of new devices on the market and the need to cope with their different characteristics. As previously pointed out by Myers et al. [2000], the increasing availability of new interaction platforms raised a new interest in model-based approaches in order to allow developers to define the input and output needs of their applications, vendors to describe the input and output capabilities of their devices, and users to specify their preferences. However, such approaches should still allow designers to have good control on the final result in order to be effective. A number of approaches have addressed multidevice user interfaces by identifying relevant information to be contained in appropriate models (and languages) for addressing such issues (examples are UIML [Helms and Abrams 2008], TERESA [Mori et al. 2002], USIXML [Limbourg et al. 2004]). In this area one specific issue is how to facilitate the development of multiple versions. Example solutions of tools providing such support are Damask [Lin and Landay 2008] and Gummy [Meksen et al. 2008]. Damask allows designers to sketch a multidevice user interface exploiting patterns and layers to indicate the parts common to all devices and those specific to a given platform. Gummy is a multiplatform graphical user interface builder that can generate an initial design for a new platform by adapting and combining features of existing user interfaces created for the same application. These model-based approaches for multidevice user interfaces have stimulated a good deal of interest. An indication is a number of initiatives that have started to define international standards in the area[1] or to define their adoption in industrial settings.[2]

Nowadays, we can identify some trends for interactive applications, which represent a new challenge and pose requirements for a fourth generation of model-based approaches: the access to applications encapsulated into a number of pre-existing Web services that can be distributed everywhere, and the use of various types of devices (in particular mobile) able to exploit a variety

[1]For example, new W3C Group on Model-based User Interfaces: http://www.w3.org/2005/Incubator/model-based-ui/

[2]For example, Working Group in NESSI NEXOF-RA IP, http://www.nexof-ra.eu/

of sensors (such as accelerometers, tilt sensors, electronic compass), localization technology (such as RFIDS, GPS), and interaction modalities (multitouch, gestures, camera-based interaction).

Indeed, Web services are increasingly used to support remote access to application functionalities, which are often described using WSDL (Web Services Description Language) files. Therefore, with the availability of Web services, which are defined before the interactive applications, the challenge is shifted on the reuse of such functionalities, the design and development of the Service Front-End (SFE) for them, and how to compose such functionalities in integrated applications with suitable user interfaces that are able to adapt to various contexts. This combination of factors has further intensified the need for identifying a suitable universal declarative language.

The goal of this article is multifold:

—to present a novel model-based language for user interfaces (MARIA, Model-based lAnguage foR Interactive Applications), which draws on previous experiences in this area;

—to show how such language can be exploited in order to design and develop multidevice user interfaces, which support access to multiple pre-existing services; and

—to show how such an approach can be exploited to support migratory user interfaces, still for applications based on the use of Web services.

In the article, after discussing some related work and analyzing our previous experience with TERESA, we identify a number of requirements for the new language and the associated supporting tool. Next, we describe the language, how it can be exploited for UI annotations of Web services, a design space for composing user interfaces, and a supporting tool. Then, we discuss how we have introduced the use of the language into the software architecture able to support migratory user interfaces. We show an example application: the migratory Pac-Man game. Lastly, some conclusions along with indications for future work are provided.

2. RELATED WORK

Logical user interfaces are specifications of user interfaces, which are able to abstract out some details and focus on the semantic aspects of a user interface. In particular, using multiple levels of abstractions is useful to identify and describe aspects that are relevant at the specific abstract level considered, which is particularly useful when designing interactive applications in multiplatform environments. One of the main advantages of logical user interface description is that they allow developers to avoid dealing with a plethora of low-level details associated with the corresponding implementation languages.

XForms[3] represents a concrete example of how the research in model-based approaches has been incorporated into an industrial standard. XForms is an XML language for expressing the next generation of Web forms, through the use of abstractions to address new heterogeneous environments. However, the

[3]http://www.w3.org/MarkUp/Forms/

language includes both abstract and concrete descriptions (control vocabulary and constructs are described in abstract terms, while presentation attributes and data types in concrete terms). XForms supports the definition of a data layer inside the form. User interface controls encapsulate relevant metadata such as labels, thereby enhancing accessibility of the application when using different modalities. 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), etc. Through the use of the appearance attribute XForms refers to concrete examples like radio buttons, checkboxes, etc. XForms is mainly used for expressing form-based UIs and less for supporting other interaction modalities, such as voice interaction. UsiXML (USer Interface eXtensible Markup Language) [Limbourg et al. 2004][4] is an XML-compliant markup language developed at University of Louvain-la-Neuve, which aims to describe the UI for multiple contexts of use. UsiXML is decomposed into several metamodels describing different aspects of the UI. There is also a transformation model that is used to define model-to-model transformations between the different models. In UsiXML, a Concrete User Interface model consists of a hierarchical decomposition of CIOs. A Concrete Interaction Object (CIO) is defined as any UI entity that users can perceive. 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, etc. Each graphicalCIO is then subtyped into one of the two possible categories: graphicalContainer, for all widgets containing other widgets, or graphicalIndividualComponent, for all other traditional widgets. In this approach the modeling of containers as subtypes of concrete interaction objects can be misleading, since their purpose should be more to indicate how to compose elements rather than to model single interactions. The authors use graph transformations for supporting model transformations, which is an interesting academic approach with some performance issues. TERESA [Mori et al. 2004] has a modular approach to support the description of abstract and concrete user interfaces. One level (concrete interface description) is represented through a number of platform-dependent languages, which are refinements of the abstract language. In TERESA there are different types of elements: interactors (describing single interaction objects), composition operators (indicating how to compose interactors), and presentations (indicating the elements that can be perceived at a given time, such as the elements of a Web page). Paterno' et al. [2008] describe how various modalities have been supported through this approach. UIML [Abrams et al. 1999; Helms and Abrams 2008] was one of the first model-based languages targeting multidevice interfaces. It structures the user interface in various parts: structure, style content, behavior, even if it has not been applied to obtain rich multimodal-user interfaces. The model-based approach has also been applied to toolkit for developing context-dependent applications [Salber et al. 1999]. Context management can be integrated with our model-based approach for user interface design and generation, as we will show when we discuss our solution for migratory user interfaces.

[4]http://www.usixml.org

Jacob et al. [1999] have identified some requirements for non-WIMP user interfaces, such as the need for support of continuous interaction and parallel input flow. MARIA provides support on this aspect because it uses temporal operators taken from the ConcurTaskTrees notation in order to describe even parallel interactions. TAC [Shaer et al. 2004] is another notation able to overcome such limitations but it is focused on specifying tangible user interfaces, while with MARIA we aim to address multidevice ubiquitous interactive applications.

Interesting issues for the definition of the language are raised from the works about the runtime execution of models.

DynaMo-AID [Clerkx et al. 2004] is a design process and runtime architecture for the creation and execution of context-aware user interfaces. The dynamic model presented can change at runtime: According to the context information some parts of the model can be added while others can be removed. At design time the model is represented using a forest of ConcurTaskTrees models. They modified the traditional notation introducing a dynamic decision node, linked with a context element where different subtrees can be added or removed at runtime according to the context element value.

The dialog model of the application can handle the transition among the presentation of a single (intradialog) or among multiple (interdialogs) trees. The traditional context model contains the services which can be dynamically discovered at runtime, together with the information about the environment and the user.

In Sottet et al. [2007] an engineering approach to produce transformation for plastic user interfaces is described. The user interface is formalized as a graph of models and mappings. The mappings have a set of usability properties that describe the criteria used for the model transformation, according to a reference framework. These criteria can be used at design time to check the support of properties of the user interface (i.e., error management, guidance, etc.), while at runtime they can be used for reasoning about the usability of the generated UI. The lack of a commonly accepted usability framework was stressed as a possible enhancement in future works.

An approach to creating executable models for human-computer interaction is presented in Blumendorf et al. [2008]. For describing a model able to change over time, they distinguish three types of elements: the definition elements that constitute the static (not changing over time) part of the model, the situation elements that constitute the dynamic part of the model, and the execution element, which describes the transition among the states of the model. A mapping connects definition elements of different models, using a situation element as trigger and an execution element for synchronizing the two definition elements. At runtime both mappings and models offer different perspectives of a system (task, domain, service, interaction), which can be used for deriving the user interface through a model-based runtime system. However, this article does not indicate precise solutions for supporting user interface adaptation at runtime exploiting the information contained in the models.

In general, on the one hand our analysis of the state-of-the-art in user interface description languages highlights that a good amount of work has

been dedicated to support multidevice user interfaces, but usually this work has been applied to support form-based interaction styles through different devices (desktop, mobile, vocal) and has not addressed the specific issues raised by emerging service-oriented architectures. On the other hand, the languages aiming to support more innovative interaction styles do not seem to have sufficient generality to support multidevice interfaces in ubiquitous environments.

Based on the lessons learned from the analysis of the state-of-the-art and our previous extensive experience with TERESA XML language and the associated tool, we have identified a number of requirements for a new language suitable to support user interfaces for applications based on Web services in ubiquitous environments. Indeed, TERESA XML language and the related tool have comprised a good case study for the model-based approach. They have been used in various applications in different projects,[5] with more than 2200 downloads, showing good potentiality for use in different approaches and different types of applications. The tool has been used in university classes, which has also provided suggestions for improving its usability and functionality; for example, students asked for support for patterns in order to be able to reuse pieces of specifications in different applications or across the same application. In addition, an evaluation of the TERESA tool has been performed [Chesta et al. 2003] in a Motorola software development center with the aim of assessing its usability for design and development of multiplatform applications. The evaluation of TERESA highlighted some flaws in the tool and the approach which provide useful input for the new MARIA language and tool, along with other inputs from its use and technological evolution.

To summarize, a number of requirements were identified for both the tool and the language. In particular, the following requirements have been identified for the tool.

—The evaluation of TERESA highlighted the need to provide designers with greater control of the user interface produced, which is an important feature, especially if we consider the ever-growing increase in flexibility (and complexity) required by the new technologies.

—The transformations for generating the corresponding implementations should not be hard-coded in the tool, but should be specified externally to allow for customization without changing the tool implementation, which requires considerable effort.

—The tool should provide support for creating front-ends for applications in which the functionalities are pre-existing in Web services.

Moreover, the following requirements were identified for the language.

—There is a need for a more flexible dialog and navigation model. For instance, with TERESA it was not possible to support complex dialogs and parallel inputs (this requirement was also supported in Jacob et al. [1999]).

[5]At http://giove.isti.cnr.it:8080/TERESA/Externaluse.jsp there is a list of organizations that have downloaded and used it

—There is a need for a flexible data model which allows the association of various types of data to the various interactors.
—The specifications of the abstract and concrete languages were too verbose in TERESA XML, with many redundancies: Cross-references between the different schemas would make the specification shorter, more readable, and consistent.
—There is a need to support more recent techniques able to change the content of user interfaces asynchronously with respect to the user interaction. Concrete examples of such techniques are Ajax scripts for Web interfaces.

3. MARIA XML

MARIA XML inherits the modular approach of TERESA XML, with one language for the abstract description and then a number of platform-dependent languages that refine the abstract one depending on the interaction resources considered. In its first version we have considered the following platforms: graphical form-based, graphical mobile form-based, vocal, digital TV, graphical direct manipulation, multimodal (graphical and vocal) for desktop and mobile, advanced mobile (with support for multitouch and accelerometers, e.g., iPhone).

3.1 Main Features

A number of features have been included in the language.

(a) *Introduction of Data Model.* We have introduced an abstract/concrete description of the underlying data model of the user interface, needed for representing the data (types, values, etc.) handled by the user interface. Thus, the interactors composing an abstract (concrete) user interface can be bound to elements of a type defined in the abstract (respectively, concrete) data model. The concrete data model is a refinement of the abstract one. The introduction of a data model also allows for more control over the admissible operations that can be performed on the various interactors. In MARIA XML, the data model is described using the XSD type definition language. Therefore, the introduction of the data model can be useful for: correlation between the values of interface elements, conditional presentation connections, conditional layout of interface parts, and specifying the format of the input values. The dependencies between the state of an interactor and the data model imply that at runtime, if a data element bound to an interactor changes its value, this change has to be notified to the interactor for updating its state and vice versa. The logic of this callback mechanism can be defined using the target technology constructs during the generation derived from the interface description.

(b) *Introduction of an Event Model.* In addition, an event model has been introduced at different abstract/concrete levels of abstractions. The introduction of an event model allows for specifying how the user interface responds to events triggered by the user.

In MARIA XML two types of events have been introduced:

(1) *Property Change Events*. These are events that change the status of some UI properties. The handlers for this type of event are only change_properties, which indicate in a declarative manner how and under which conditions property values change.
(2) *Activation Events*. These are events raised by activators, which are interactors with the purpose of activating some application functionality (e.g., access to a database or to a Web service). This type of event can have either change_properties or script handlers (which have an associated generic script).

The abstract definition of these events contains the information for the generation of the dynamic behavior of the final UI.

(c) *Supporting Ajax Scripts, which Allow the Continuous Updating of Fields.* Another aspect that has been included in MARIA is the possibility of supporting continuous fields updating at the abstract level. We have introduced an attribute to the interactors: continuosly updated= "true"["false"]. The concrete level provides more detail on this feature, depending on the technology used for the final UI (Ajax for Web interfaces, callback for stand-alone application, etc.). For instance, with Ajax asynchronous mechanisms, there is a behind-the-scene communication between the client and the server about what has to be modified in the presentation, without an explicit request from the user. When it is necessary the client redraws the relevant part, rather than redrawing the entire presentation from scratch. Thus it allows for quicker changes and real-time updates. It is worth noting that while at the abstract level a certain interactor has to support continuous dynamic updating of its values from a certain abstract data source, at the concrete level, the specific platform-dependent technology used to support such continuous updating of the interactor must be specified.

(d) *Dynamic Set of User Interface Elements.* Another feature that has been included in MARIA XML is the possibility to express the need to dynamically change only a part of the UI. This has been specified in such a way as to affect both how the UI elements are arranged in a single presentation, and how it is possible to navigate between the different presentations. The content of a presentation can dynamically change (this is also useful for supporting Ajax techniques). In addition, it is also possible to specify dynamic behavior that changes depending on specific conditions: This is obtained through the use of conditional connections between presentations.

In the next sections we will provide a more detailed description of concepts and models that have been included in MARIA, both for the Abstract UI and the Concrete UI. Regarding the definition of these abstraction layers, there is a general agreement in the model-based community (see, for example, the CAMELEON Reference Framework [Calvary et al. 2002]): The abstract description is independent of the interaction resources available in the target device while the concrete description depends on the type of interaction modality but is independent of the implementation language.

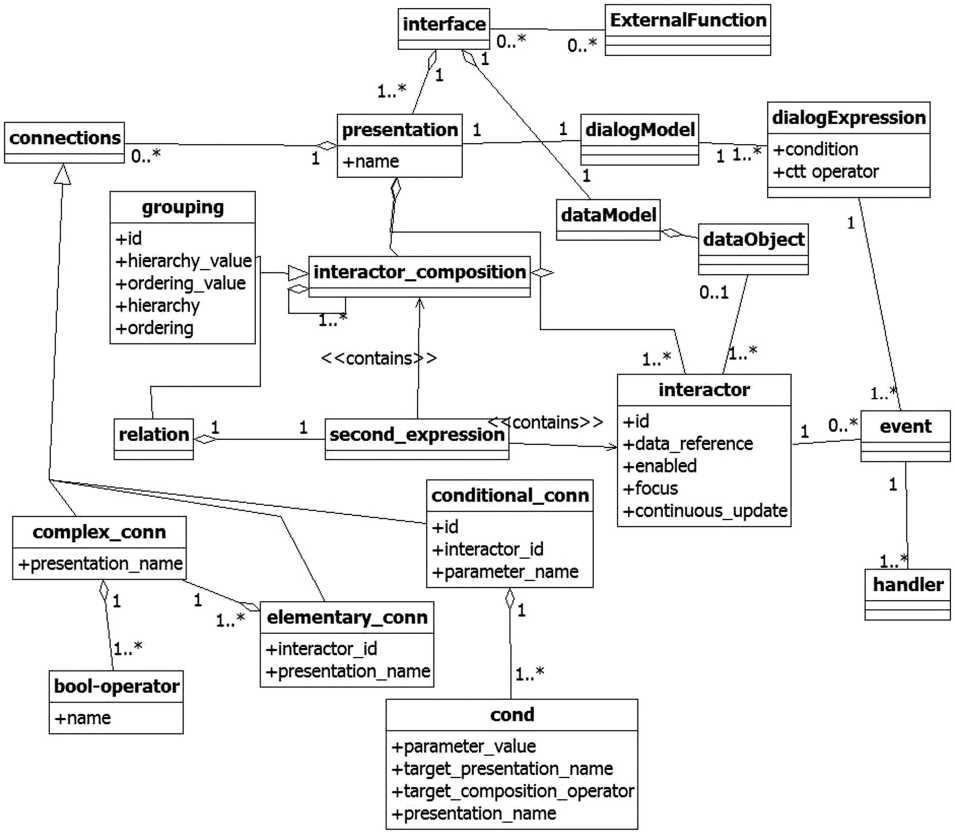

Fig. 1. The metamodel for the abstract user interface.

3.2 MARIA XML—Abstract Interface Description

It is generally recognized that one of the main benefits of using a user interface abstract description is for designers of multidevice interfaces, because they do not have to learn all the details of the many possible implementation languages supported by the various devices. Therefore, designers can reason in abstract terms without being tied to a particular platform/modality/implementation language. In this way, they can focus on the semantics of the interaction (what the intended goal of the interaction is), regardless of the details and specificities of the particular environment considered.

Figure 1 shows the main elements of the abstract user interface metamodel (some details have been omitted for clarity). As can be seen, an interface is composed of one data model and one or more presentations. Each presentation is composed of name, a number of possible connections, elementary interactors, and interactor compositions. The presentation is also associated with a dialog model which provides information about the events that can be triggered at a given time. The dynamic behavior of the events, and the associated handlers, is specified using the CTT temporal operators (for example, concurrency,

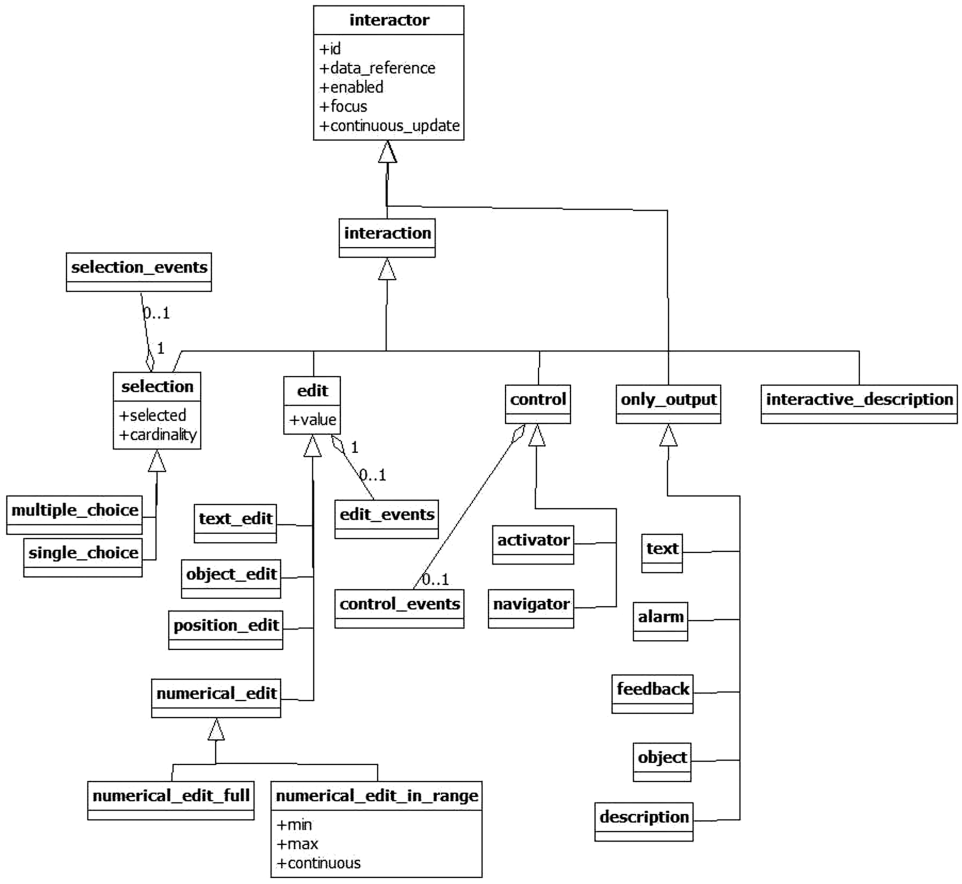

Fig. 2. The metamodel for the part of the abstract user interface dedicated to interactors.

or mutually exclusive choices, or sequentiality, etc.). Indeed, CTT provides a richer set of temporal operators with respect to traditional techniques for task modeling such as GOMS [John and Kieras 1996]. When an event occurs, it produces a set of effects (such as performing operations, calling services, etc.) and can change the set of currently enabled events (for example, an event occurring on an interactor can affect the behavior of another interactor, by disabling the availability of an event associated to another interactor). The dialog model can also be used to describe parallel interaction between user and interface.

A connection indicates what the next active presentation will be when a given interaction takes place. It can be either an elementary connection, a complex connection (when Boolean operators compose several connections), or a conditional connection (when specific conditions are associated with it).

There are two types of interactor compositions: grouping or relation. The latter has at least two elements (interactor or interactor compositions) that are related to each other. An interactor (see Figure 2) can be either an interaction object or an only_output object. The first one can be one of the following

types: selection, edit, control, interactive description, depending on the type of activity the user is supposed to carry out through such objects. An only_output interactor can be object, description, feedback, alarm, text, depending on the supposed information that the application provides to the user through this interactor.

The selection object is refined into single_choice and multiple_choice depending on the number of selections the user can perform. It is worth pointing out that further refinement of each of these objects can be done only by specifying some platform-dependent characteristics, therefore it is specified at the concrete level (see the next section for some examples). The edit object can be further refined at the abstract level into text_edit, object_edit, numerical_edit, and position_edit, depending on the type of effect desired. A more refined indication of the elements that can be edited is obtained through the use of the data model. The control object is refined into two different interactors depending on the type of activity supported (navigator: navigate between different presentations; activator: trigger the activation of a functionality). It is worth pointing out that all the interaction objects have associated events in order to manage the possibility for the user interface to model how to react after the occurrence of some events in their UI. The events differ depending on the type of object they are associated with. Also this feature was not supported in TERESA.

3.3 MARIA XML—Concrete Description

The purpose of the concrete description is to provide a platform-dependent but implementation language-independent description of the user interface. Thus, it assumes that there are certain available interaction resources that characterize the set of devices belonging to the considered platform. It moreover provides an intermediate description between the abstract description and that supported by the available implementation languages for that platform.

In order to enhance the readability of the language, we decided to specify in the concrete user interface only the details of the concrete elements, leaving the specification of the higher hierarchy in the abstract metamodel. In this way, we avoided the problem of redundancy among the two models (abstract/concrete), since the higher-level hierarchy is fully specified at the abstract level and just referred at the concrete level.

As one example of the various concrete language descriptions in MARIA, we discuss the one developed for the *multitouch mobile user interface* platform. Indeed, the iPhone has modified the typical interaction with a mobile phone and other vendors (HTC, Samsung, etc.) followed its example and introduced similar interaction techniques in their devices. Thus, it is reasonable to introduce a new concrete platform for modeling this new type of device, the *multitouch mobile platform*, which enhances the typical mobile platform by modeling the following capabilities:

—a multitouch screen, able to handle different touches at the same time. A set of gestures, which can be handled as events, is then defined based on this capability (such as two-finger zooming and rotating); and

Table I. XML Schema Definition of the Single Choice Interactor

```
<xs:complexType name="single_choice_type">
  <xs:complexContent>
    <xs:extension base="single_choice_type">
      <xs:choice minOccurs="1" maxOccurs="1">
        <xs:element name="radio_button" type="radio_button_type" />
        <xs:element name="list_box" type="list_box_type" />
        <xs:element name="drop_down_list" type="drop_down_list_type" />
        <xs:element name="image_map" type="image_map_navigator_type" />
      </xs:choice>
    </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

—a three axis accelerometer, able to get the orientation of the screen: landscape, landscape inverse, portrait, portrait inverse.

For supporting the multitouch interaction the device generates three types of events.

(1) *Mouse Emulation Events*. Most of the one-finger touches and some two-finger touches (for instance, the scrolling) are mapped onto the traditional mouse events (i.e., mouse over, mouse down, mouse move, mouse up, etc.).
(2) *Gesture Events*. The device provides the support for recognizing predefined interactions with the multitouch screen, allowing the developer to handle high-level gesture events, such as zooming and rotating, without explicitly tracking the screen touches.
(3) *Touch Events*. This type of event allows the programmers to create ad hoc multitouch interactions, receiving an array of touches, each one with the position on the screen.

Orientation changes are notified by a single event called *orientation changed*, which indicates the new screen orientation.

Also the new concrete platform is a refinement of the abstract user interface metamodel: Each interactor type is refined by adding the specification of the concrete events. For instance, the *single choice* interactor can be refined with four concrete interactors: *radio_button, list_box, drop_down_list, image_map* (see Table I).

A typical interactor for the *mobile multitouch platform* contains events corresponding to mouse events emulation, touch events, the list of reasonable gesture events (for example, we do not expect to have a rotation event on a radiobutton), and the orientation change event (see Table II).

The previous example (see Table II) shows the specification of the radio button interactor: It contains a list of choice elements (key-value pairs) and a list of events, which consist of four group references and an element. The groups define, respectively, the typical mouse and key events, the touch events, and the orientation events.

The *touch property events* group contains three events: *touch start* (sent when a finger touches the screen surface), *touch move* (sent when a finger moves on

Table II. XML Schema Definition of the Radio Button Concrete Interactor

```
<xs:complexType name="radio_button_type">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded"
                  name="choice_element" type="choice_element_type" />
      <xs:element name="events" minOccurs="0" maxOccurs="1">
        <xs:complexType>
          <xs:sequence>
            <xs:group ref="mouse _events" />
            <xs:group ref="key _events" />
            <xs:group ref="touch _events" />
            <xs:group ref="orientation _events" />
            <xs:element name="zoom_gesture" minOccurs="0"
                        maxOccurs="1" type="gesture _event" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="label" type="xs:string" use="required" />
    <xs:attribute name="alignment" type="alignment_type"
                  use="required" />
</xs:complexType>
```

the surface), and *touch end* (sent when a finger leaves the screen surface). Each event contains an array of touches with the current position on the screen. The *orientation property events* group contains the event *orientation changed* which contains the current screen orientation. The *zoom gesture event* notifies that a zoom command has been recognized by the system and contains the scale factor. The data types defined into the events definition are supposed to be mapped with the real data type engine of the target technology when the model will be transformed into executable code. So they are an abstraction that allows developer to reason about the runtime behavior abstracting from technologies issues.

An example of implementation language that can be derived from this *multitouch concrete description* is XHTML + Safari DOM Extension.

4. SUPPORT FOR APPLICATIONS BASED ON WEB SERVICES

Web services are increasingly used to support remote access to application functionalities, particularly in ubiquitous environments. They are described using WSDL (Web Services Description Language) files, which are XML-based descriptions as well.

In this section we discuss how MARIA can be exploited to support their development. In particular, we show how it can be used to provide user interface-related annotations of Web services and then to compose their corresponding user interface specifications.

4.1 UI Annotations for Web Services

UI annotations are hints associated with Web services to obtain better user interfaces. Indeed, with this approach, the functionalities/services are created without aiming at any particular application in which to include them, since

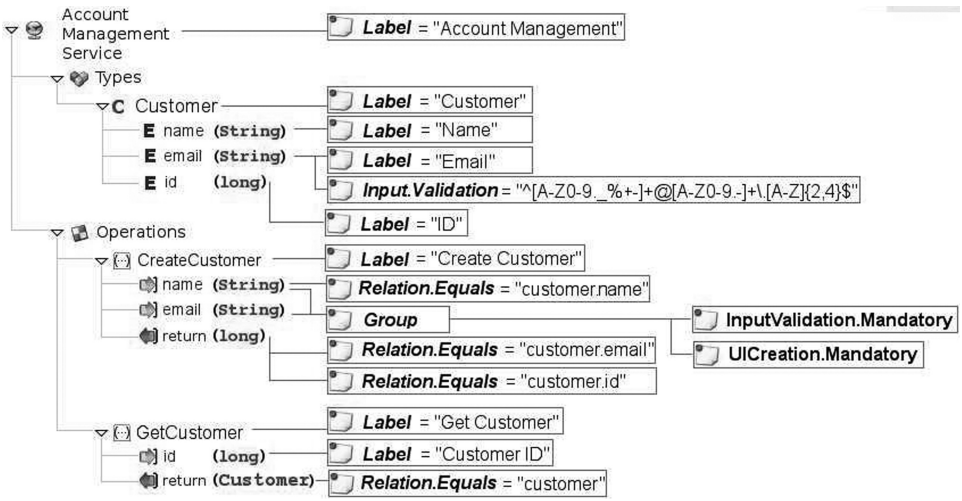

Fig. 3. An example of annotation for a service.

their main characteristic is reusability across different applications, even using different types of interactive technologies to make them available to the final user. In order to obtain good quality user interfaces in the various contexts in which such services will finally be used, annotations can be included: Their goal is to provide hints for creating the user interface to access a Web service.

As happens for UI languages, also annotations can be specified using such models and different abstraction levels. Here we show an example regarding a simple service definition for managing customer accounts, which has two operations: one for creating a new customer and another one for getting information about a customer. The service annotator would like to specify some hints about how the service should be finally rendered. Figure 3 (right part) shows the information s/he wants to add to the service definition (left part).

As can be seen from Figure 3, some information regards the data types manipulated by the service (the regular expression for validation, default labels for fields), while other information is about the presentation of a UI for accessing an operation (group, input validation, relation between operation parameters, and data types). The resulting annotations can be expressed at the abstract level using MARIA XML: The relevant part of the WSDL file is annotated using the "ref" attribute of the annotation element (see XML excerpt in Table III).

As shown in the preceding XML code excerpt, the annotations on the data types are represented as a redefinition of the Customer type, adding the validation restriction (using the regular expressions of XSD). This new data type is manipulated in the UI, and can be simply mapped on the service data type.

For the operation Create Customer, the label of the operation is the name of the corresponding grouping of elements. The type of interactor to be used, if not specified directly in the annotation, can be obtained as follows.

Table III. XML Excerpt from a WSDL Annotation

```
<annotations ref="http://someDomain.org/accountManagementService.wsdl">
  <annotation ref="wsdl:service/AccountManagementService">
      <aui>
          <grouping name="Account Management Service" />
      </aui>
  </annotation>
  <!-- DATA ANNOTATIONS -->
  <annotation ref="wsdl:types/customer">
     <data>
        <xs:simpleType name="nameType">
           <xs:restriction base="xsd:string">
              <xsd:pattern value="[A-Za-z0-9]"/>
           </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="emailType">[...]</xs:simpleType>
        <xs:complexType name="Customer">
           <xs:sequence>
              <xs:element name="Name" type="nameType" />
                  [...]
           </xs:sequence>
        </xs:complexType>
     <data>
  </annotation>
  <!-- CREATE CUSTOMER -->
  <annotation ref="wsdl:operations/createCustomer">
     <aui>
       <aui:grouping name="Create Customer">
          <aui:grouping>
            <aui:text_edit parameter="Customer/name" required="true"/>
               [...]
            </aui:only_output parameter="Customer/id">
          <aui:grouping>
       </aui:grouping>
    </aui>
  </annotation>
  <!-- GET CUSTOMER -->
  <annotation ref="wsdl:operations/getCustomer">
     <aui>
        <aui:grouping name="Get Customer">
           <aui:numerical_edit parameter="Customer/Id"/>
           <aui:only_output parameter="Customer"/>
        </aui:grouping>
    </aui>
  </annotation>
</annotations>
```

—As the result value *only output* interactors will be used.

—As the input parameters *editing* or *selection* interactors will be used.

—The parameter type will help choose the correct interactor, for example, numerical_edit for editing numbers, text_edit for editing texts, etc.

—The input and output interactors are linked with the UI data types using the parameter attribute.

For the Create Customer operation, we have two text_edit interactors for customer name and email (linked to the data type) and a single output for the resulting id. Name and email are grouped together as specified by the Service

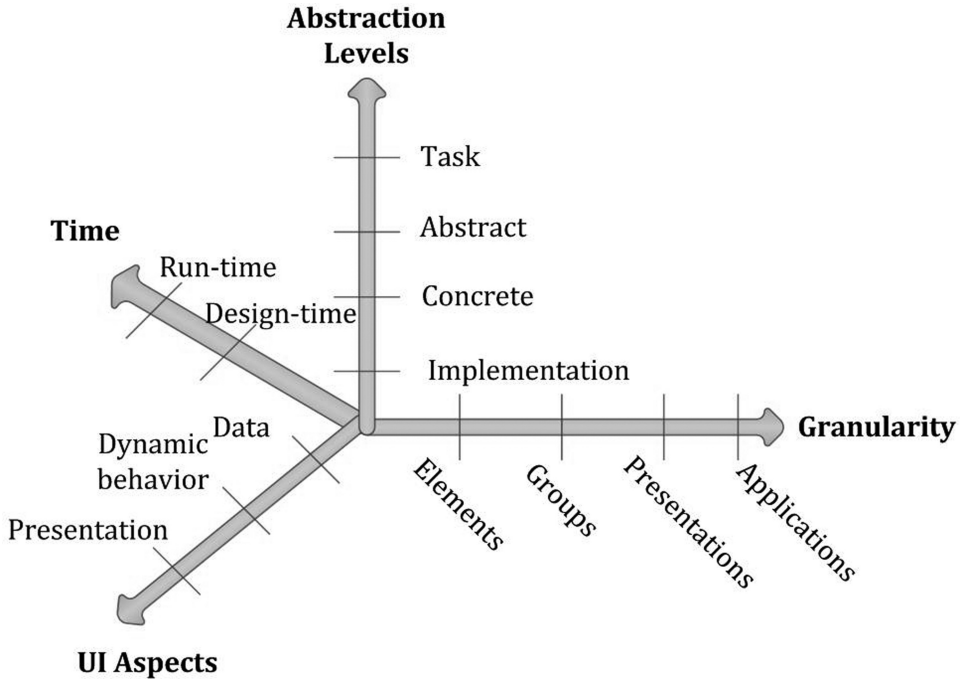

Fig. 4. The design space.

Annotator (see Figure 3, right part). The Get Customer operation takes a long integer as input (the customer id), which is supported by a numerical edit interactor, while the output is a Customer whose rendering is not specified in the annotation, though in a further step a default rendering can be suggested using the Customer data type definition (two outputs only for name and email).

4.2 UI Composition

In the case of applications based on Web services, designers and developers often have to compose existing functionalities and corresponding user interface specifications (which can be identified through the annotations discussed in the previous section). In order to better understand this composition activity we have identified a design space for composing user interfaces (see Figure 4).

Four main aspects have been identified: the abstraction level of the user interface description, the granularity of the user interface considered, the types of aspects that are affected by the UI composition, and the time when the composition occurs (design time/runtime).

Regarding the abstraction level, since a user interface can be described at various abstraction levels (task and objects, abstract, concrete, and implementation), the user interface composition can occur at each of such abstraction levels. The granularity refers to the size of elements to be composed: Indeed, we can compose single user interface elements (for example, a selection object

with an object for editing a value), we can compose groups of objects (for example, a navigation *bar* with a *list* of news), we can compose various types of interface elements and groups to obtain an entire presentation, we can join presentations in order to obtain the user interface for an entire application. It is also possible to compose user interfaces of applications to obtain so-called mash-ups. It is worth pointing out that the term "presentation" refers to the set of user interface elements that can be perceived at a given time; a common example is a graphical Web page.

Also, we have to distinguish the different types of compositions depending on the main aspects that they affect:

—the dynamic behavior of the user interface, which means the possible sequencing of user actions and system feedback, but also the additional dynamic behavior of some UI objects (e.g., when some elements of the UI appear or disappear depending on some conditions);
—the perceivable UI objects (for example, in graphical user interfaces we have to indicate the spatial relations among the composed elements); and
—the data that are manipulated by the user interface.

Lastly, we have to identify the phase when the composition occurs: It can be either a static composition (occurring at design time), or a dynamic composition (occurring at runtime, namely during the execution of the application). This latter composition is especially significant in ubiquitous applications, since in such environments services can dynamically appear and disappear.

In the following section we provide some examples of the composition in order to clarify and describe in more detail some of the elements appearing in the framework. However, we have to point out that in some situations, multiple values on the same dimension can be involved in the same composition process.

Figure 5 shows an example composition: There is a first service (the related UI is in the top part of the figure) which delivers a list of restaurants in a specific area (certain information is available for the list elements: name, address, opening time, photo, etc.). The geographical area given as input to this first service can vary, for instance, depending on the current position of a mobile user (provided by a GPS). Another service is available, and this is a mapping service which simply visualizes objects on a map. Therefore, this second service receives the position of a set of objects, and shows them on a graphical map. These two services can be combined together and this composition is carried out based on temporal sequencing aspects. For example, as soon as the user selects a particular restaurant among the currently available ones, the information regarding the restaurant is rendered on the composed user interface, and the UI adapts to the characteristics of the current (mobile) device.

In particular, as can be seen in the bottom left of Figure 5, a picture of the selected restaurant is displayed on a separate part of the window. The position of the restaurant is displayed on the map using a bulb-shaped blue icon to distinguish it from the (red) color used for displaying the other restaurants. In addition, since the device considered is mobile, an adaptation step is

Restaurants

Name	Address	Opening Time	Photo	Info	Contact
Antica Osteria del Teatro	Via Ghibellina, 15	Mon-Sun 12.30-15.00; 20.00-24 Closed on Wednesday	Osteria.jpg	Typical Tuscan cuisine. Situated in the historical center of the...	Tel: +39 055 244277 info@osteriadelteatro.com
Da Francesco	Via di San Giuseppe, 26	Closed on Monday	Francesco.gif	Vegetarian food	Tel/fax: +39 055 212677 contact@dafrancesco.net
Osteria da Giovanni	Via S. Egidio, 53	Closed on Tuesday (lunch time)	ost-giovanni.gif	Informal place.	Tel: +39 055 145770 info@osteriadagiovanni.com
Ristorante Ambassador	Via Ghibellina, 25	Mon-Thu: 19.45-24 Fri-Sun: 12.30-15.00; 19.45-24	ambassador.tiff	International food. Elegant restaurant located in an amaz...	Tel: +39 055 1257870
Cantina Barbagianni	Strada Borgo Pinti, 124	Tuesday Closed (entire day)	Cantina-barbagianni.jpg		Tel/Fax: +39 055 6351870

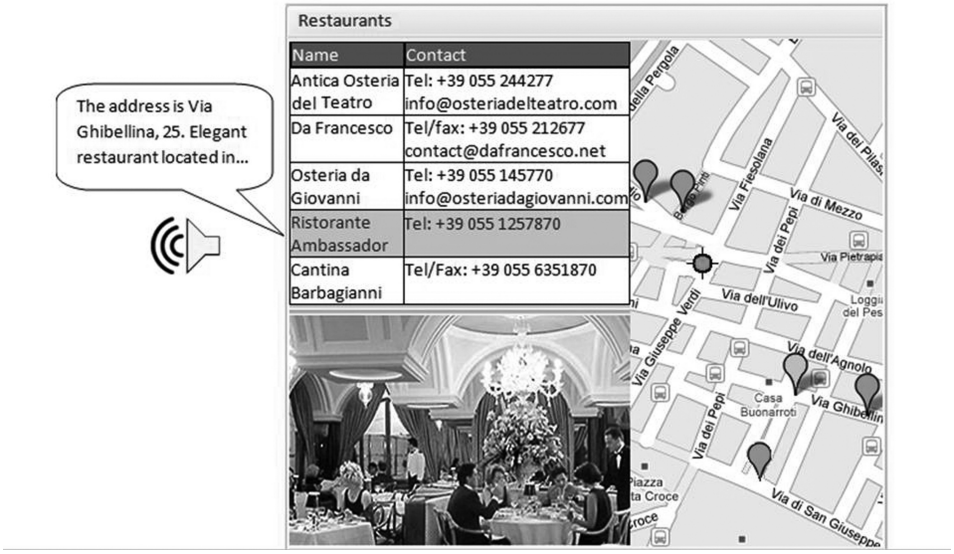

Fig. 5. (top) An example of UI service; (bottom) an example of composition of UI services.

performed when the UI is rendered on the user's mobile device. Indeed, only a part of the information available for the restaurants is immediately visible on the screen, and this is done in order to save screen space. The complete information for a restaurant is only rendered when the user selects a particular restaurant: In this case a picture is displayed, a part of (textual) information is visualized, while the remaining information is rendered vocally on the mobile device.

As far as our design space is concerned, in this case the composition is carried out at the implementation level, involving groups of UI objects. Moreover, it affects temporal aspects, since as soon as the user selects a particular item on the list, different events occur concurrently in the UI: The map changes its appearance (the selected item is highlighted with a different color in the map), the picture shown at the bottom left of the UI changes, and a vocal rendering of the remaining information about the selected restaurant starts. However, another level of composition is carried out in this example: Indeed, also the UI objects are affected. For instance, a part of the information about the restaurants is rendered vocally in the composed UI (see Figure 5, bottom part), while in the first service it was simply visualized graphically, as can be seen from Figure 5, top part.

5. MARIA TOOL

A new authoring tool has been developed in order to support the editing of user interface specifications with the new language, in particular for interactive applications based on Web services.

5.1 MARIA Tool Requirements

Several requirements have driven the building of the new authoring environment. First of all, there is the requirement for an environment to support the design and generation of user interfaces for applications based on Web services. For this purpose, the tool is able to support mappings between the WSDL files (which contain the interface descriptions of the functionalities supported by Web services) and the logical user interfaces. Then, it is able to perform a number of transformations that support the generation of the resulting user interface through a semi-automatic refinement process.

Secondly, there is the ability to provide flexible support, able to address different approaches. Thus, not only top-down approaches (generally used with model-based approaches at design time), but also bottom-up approaches, or even the support for mixed approaches can be used. Moreover, there is the possibility of transformations between the different abstraction levels, which are not hard-coded in the environment, but externally defined. In this way, it is possible to modify them without modifying the implementation of the tool. As the tool can handle XML-based descriptions, we have used XSLT to express and carry out such transformations.

5.2 Transformations in MARIA

The new MARIA authoring environment provides the possibility to customize the model-to-model transformations (forward and reverse) and also the rules allowing passing from a model to the final implementation. For instance, it is possible to specify general rules for associating concrete elements with abstract ones, or specifying how to render concrete elements using a specific implementation technology. These types of transformations involve the document structure, and are suitable for the transformation of all documents of a specified type to another type. It is also possible to define document instance rules, which can override the general template, for the fine-tuning of the result. In the following paragraphs we will further detail the two types of transformations.

General transformation. The first type of transformation maps the elements of a source document to the elements of a target document. The transformation process consists of three high-level steps.

(1) *Analysis of the Document Structure*. From the XSD definition of the structure of the source and target documents, the transformation engine creates the list of elements and attributes.
(2) *Definition of Mappings*. The designer defines a set of mappings between the elements and attributes of the source and target documents. These mappings can be one-to-one, or many-to-one. The mapping can also contain the definition of some data transformations between the source and the

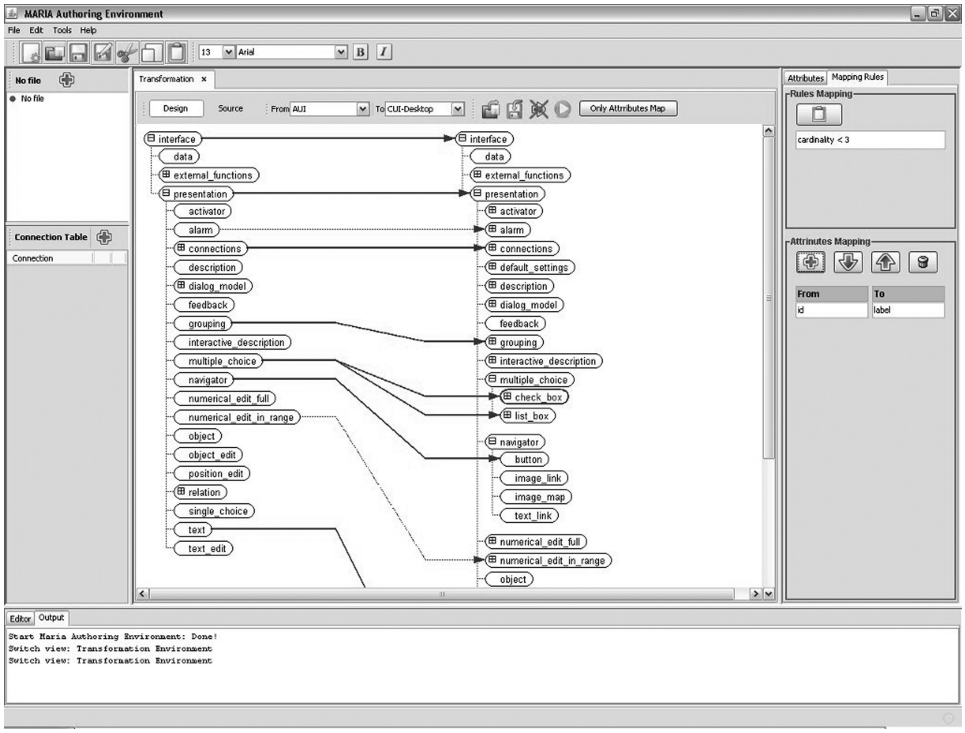

Fig. 6. The UI of the transformation tool.

target (for example, we can extract a single piece of information from an attribute, or merge two attributes values into one, etc.)

(3) *Performing the Transformation*. The source document and the mappings are the input to the transformation engine that creates the target document as output.

The user edits the rules through a graphical interface that presents the source and the target document with a tree layout. The elements and the attributes can be linked by drawing arrows: When the transformation engine encounters the specified source element (attribute), a target element (attribute) is added in the output and filled in with the source element (attribute) content. However, it is also possible to define a transformation of the content able to create a different representation of the source information (for example, an address can be the source attribute and only the postal code is extracted for the target attribute). Figure 6 shows the UI of the part of tool supporting the transformation.

Document instance transformation. Sometimes the user needs to change the transformation rule for a small part of a specific document, applying a special transformation to fine-tune the final result. It is possible to override the template rules defining a mapping at the document instance level, by selecting a specified element and transforming it into an element of the target document.

The transformation is performed only for the specified instance of the element in the document.

The transformation engine has been implemented using an XSLT generator that creates a stylesheet from the mappings defined by the user. In turn, the stylesheet and the source document are input to an XSLT transformation engine, which creates the target document. This implementation has the advantage that the transformation defined can also be used by different tools.

5.3 The Design Process for Applications Based on Web services

In this section we briefly discuss how to exploit the described universal declarative language to support the development of user interfaces for applications based on Web services in ubiquitous environments.

If we want to address the issues of creating interactive applications accessing functionalities developed by others (such as in UI for services), it soon becomes clear that a traditional top-down approach going through the various abstraction layers is not particularly effective. A top-down approach essentially consists in breaking down an overall system by refining it into its subsystems. Indeed, since the top-down approach aims to refine the entire system, it is particularly effective when the design starts from scratch, so that the designer has an overall picture of the system to be designed and refines it gradually.

Instead, if the designer wants to include already existing pieces of software, like services in SOA, this necessarily requires that a bottom-up approach be included in the design process in order to exploit such legacy, fine-grained functionalities and identify their relationships to compose them in larger/higher-level functionalities.

However, the best option seems to be a hybrid solution in which a mix of bottom-up and top-down approaches is used. More specifically, first a bottom-up step is envisaged, in order to analyze the Web services providing functionalities useful for the new application to develop. This implies analyzing the operations and the data types associated with the input and output parameters in order to associate them with suitable abstract interaction objects. Then, there is a step aiming to define the relationships among such elements. In order to do this, we envisage the use of task models expressed in CTT for describing the interactive application and how it assumes that tasks are performed. In this case, the Web services can be viewed as a particular type of task (system task, namely a task whose performance is entirely allocated to the application), and the temporal relationships that are specified in a task model can indicate also how to compose such functionalities. This process (specifying the task model) should be driven by the user requirements and also implies some constraints on how to express such functionalities. Indeed, in order to be able to address the right level of granularity, not only will a Web service be associated to an application task, but it is required that each operation specified within the Web service be associated to a different task. Thus, if a Web Service supports three operations, then there would be three basic system tasks, with the parent

task being another application task (corresponding to the Web service itself). Once we have obtained the task model, it is possible to generate the various UI descriptions in a top-down manner, and then refine them up to the implementation, by using the MARIA tool. In this refinement process the information contained in the Web service annotations can be exploited as well.

6. EXPLOITING MARIA MODELS AT RUNTIME IN UBIQUITOUS ENVIRONMENTS: APPLICATION TO MIGRATORY USER INTERFACES

Model-based UIDLs are utilized at design time to help the user interface designer cope with the increasing complexity of today's applications and contexts. The underlying user interface models are mostly used to generate a final user interface code, which is then executed at runtime. Nevertheless, approaches utilizing the models at runtime are receiving increasing attention. We agree with Sottet et al. [2007], who call for keeping the models alive at runtime to make the design rationale available. We are convinced that the utilization of models at runtime can provide useful results, such as support of migratory user interfaces. Migratory user interfaces are interactive applications that can transfer among different devices while preserving the state and therefore giving the sense of a noninterrupted activity. The basic idea is that devices that can be involved in the migration process should be able to run a migration client, which is used to allow the migration infrastructure to find such devices and know their features. Such a client is also able to send the trigger event (to activate the migration) to the migration server, when it is activated by the user. At that point the state of the source interface will be transmitted to the server in order to be adapted and associated to the new user interface automatically generated for the target device (see Figure 7).

We have designed and developed a software architecture able to support the main phases in the migration, based on previous experiences in the field [Bandelloni et al. 2005]. The first phase is device discovery (step 1 in Figure 8). It concerns the identification of the devices that can be involved in the migration process and the attributes that can be relevant for migration (private or public device, their connectivity, their interaction resources, etc.). The device discovery has to be activated in every device that is involved in the migration (including the Migration Server). Its purpose is allowing the user to trigger the migration by selecting the migration target device. In order to do this, the device discovery module has to notify the presence of the associated device to a known multicast group. The list of the devices currently subscribed to such a group defines the list of devices that are available and could be involved in a migration process. In order to carry out this notification, the device discovery/migration client modules use multicast datagrams communications using the UDP/IP protocol.

After the device discovery phase, the user requests a page access from the current device (2), the request is transmitted to the application server (3), which provides the page (4). Such page is downloaded by the Migration Server (which works also as a proxy server) and then annotated by the Migration Server before delivering it to the requesting device (5). The annotation is the automatical insertion of a script that is then used to transmit the state of the user interface.

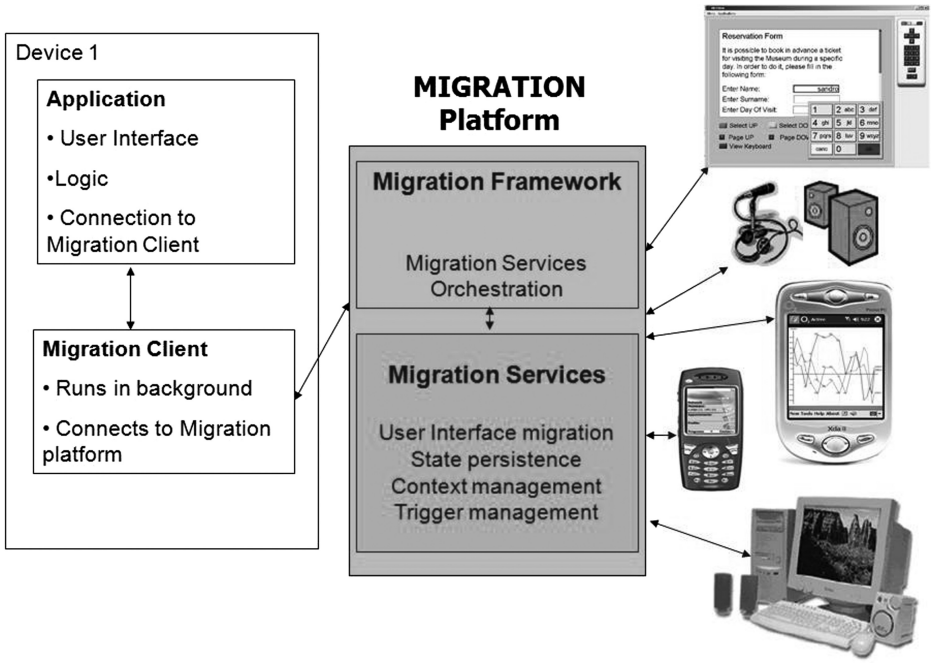

Fig. 7. The migration approach.

Thus, when the migration is triggered and the target device is identified (6), the current page along with its state is transmitted through an Ajax Script (7) to the migration server. At that point a version of the page adapted for the target device is created, with associated the state of the source version, and uploaded so that the user can immediately continue with all the data entered beforehand.

Figure 9 shows how the abstraction layers are exploited to support migratory user interfaces, by showing the various activities that are done by the Migration Server. First of all the migration approach supposes that various UI models at different abstraction levels are associated to each of the various devices involved in a migration: Such UI models are stored and manipulated centrally, in the Migration Server.

The current architecture assumes that a desktop Web version of the application front-end exists and it is available in the corresponding Application Server: This seems a reasonable assumption given the wide availability of this type of applications. Then, from such a final UI version for the desktop platform, the Migration Server automatically generates a logical, concrete UI description for the desktop platform through a reverse-engineering process. After having obtained such a concrete UI description for the desktop platform, the Migration Server performs a semantic redesign of such CUI [Paterno' et al. 2008] for creating a new, concrete, logical description of the user interface, adapted to the target device. The purpose of the semantic redesign is to preserve the semantics of the user interactions that should be provided to the user but to adapt the structure of the user interface to the resources available in the target device.

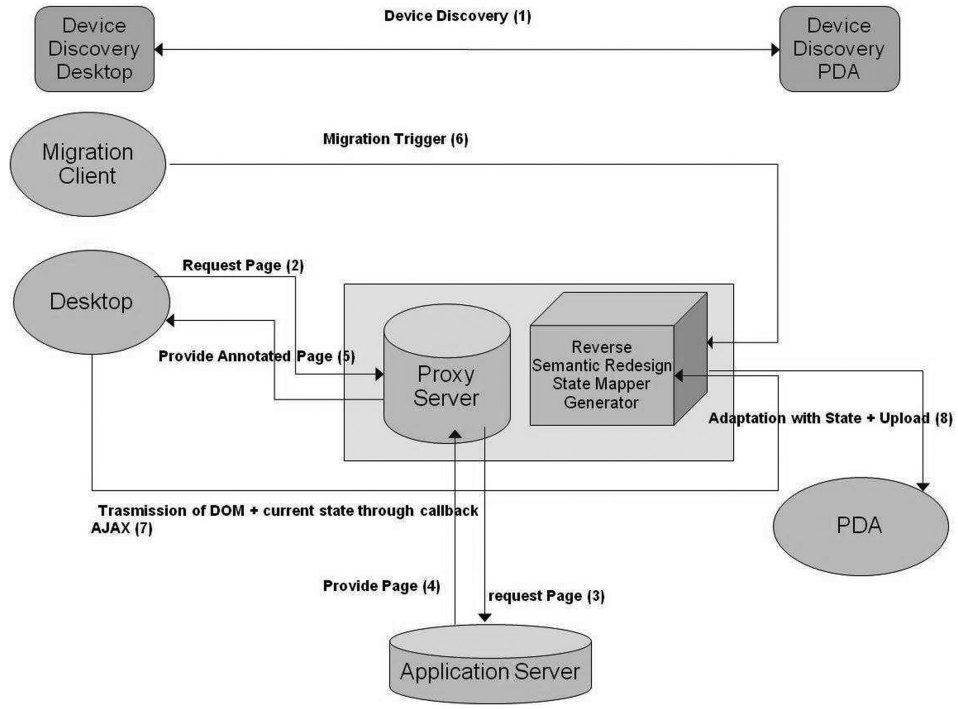

Fig. 8. The steps of the migration.

For all the tasks that can be supported, the semantic redesign module identifies concrete techniques that preserve the semantics of the interaction but supports it and are most suitable for the new device (for example, in mobile devices it will replace interactors with others that provide the same type of input but occupying less screen space). In a similar way also page splitting is supported: When there are pages too heavy for the target device, they are split taking into account their logical structure so that elements logically connected remain in the same page. Thus, the groupings and relations are identified and some of them are allocated to newly created presentations so that the corresponding page can be sustainable by the target devices.

The state is extracted through specific JavaScripts, which are automatically included in the Web pages when they are accessed through the intermediate migration server. When the migration is triggered, the state is transmitted to the server where there is a module (State Mapper) whose purpose is to associate the state with the concrete description for the target device, which is used for the generation of the final user interface.

In testing our prototype we realized that the previous language (TERESA) was inadequate to support the automatic creation of the logical descriptions of various existing Web pages (for example, an event model was not supported). Thus, we have introduced the use of MARIA, which is able to overcome such limitations.

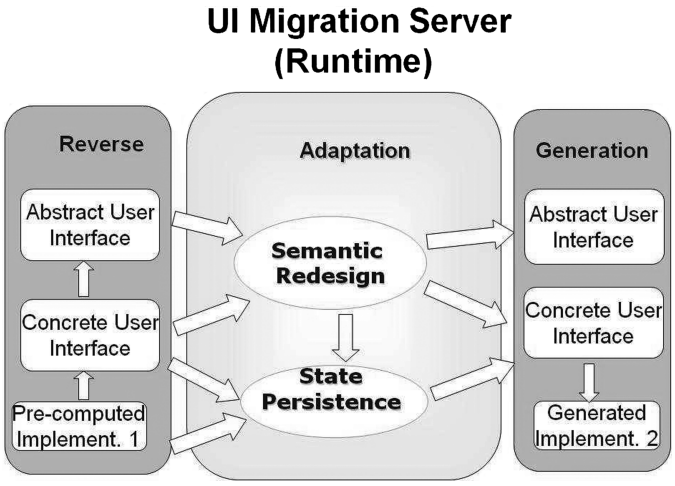

Fig. 9. The relationships among abstraction layers supporting migration.

7. MIGRATORY PAC-MAN CASE STUDY

In this section we show an example application of our approach, which supports a migratory Pac-Man game exploiting MARIA specifications. We considered Web services as the implementation technology for the application logic, since with this solution the distinction between the service frontend and the application logic is clear and this technology is a mainstream solution in the area of software and services. We consider a scenario in which one user plays on a desktop device and at some point migrates to an iPhone, through which s/he can interact using multitouch and accelerometer as well. In this case the service frontend supports positioning of the game elements (user inputs, collision detection, and user interface presentation), while the Web services support ghost game strategy, postcatch ghost policy (namely, what happens to the ghost after being eaten); scoring rules, and rules for game levels. The information flows between the Service Front End (SFE) and the Web Services (WS) during the game. For instance, when the SFE detects that all the dots have been eaten by the Pac-Man, the SFE requests a new maze from the WS, which then sends it. Another example is the situation when, during the game, one of the special pills is eaten by the Pac-Man: In this case, the SFE has to communicate this to the WS, which delivers the new algorithm for controlling the new ghosts' strategy (for escaping from Pac-Man) and also sets the time interval (e.g., number of seconds) during which this new situation should last.

The considered desktop game version (written in XHTML and JavaScript) includes different parts (see Figure 10): the maze of the game with the different characters (the ghosts and the Pac-Man), a part devoted to visualizing the current state of the game (the number of Pac-Man lives still available, the current level of the game, the score), and interaction elements for controlling the game. This last part includes the controls for moving the Pac-Man (the 4 arrow - imagemap, used for directing the Pac-Man) together with the two game controls (starting a new game, implemented with the central "New Game" button in the

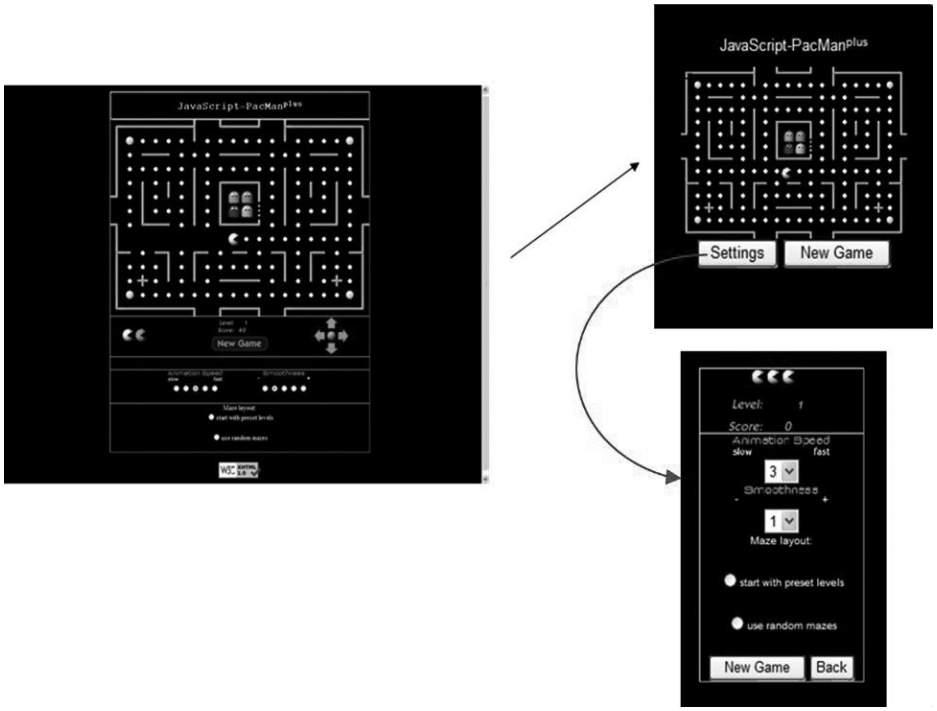

Fig. 10. Pac-Man migration example: (left) desktop version; (right) mobile version.

left part of Figure 10 and pausing the game, which is implemented with the central round-shape button in the imagemap), as well as the settings for specifying the general game configuration (e.g., the animation speed of the Pac-Man, and the possibility of using random maze layouts for the various levels, etc.). In the desktop Web page, for controlling the Pac-Man, it is also possible to use the keyboard (e.g., by using the arrow keys or some other predefined keys). In addition, in order to pause and start a new game two keys are also available ("n" and "p", respectively).

The Web page of the Pac-Man game for the desktop platform is transformed by the migratory platform in order to obtain a new version adapted for the mobile device. To do this, a reverse engineering of Pac-Man for the Web desktop platform is needed in order to obtain its logical concrete (desktop) UI description. Such a description will then be redesigned to finally generate the adapted user interface for the mobile device.

The application of an *enhanced* reverse-engineering process to the Pac-Man example allowed supporting some UI aspects that were not supported in previous versions of the associated algorithm (i.e., that producing TERESA XML -based specifications). Now, by using a MARIA XML-compliant specification as the planned output, we were able to specify aspects that were not considered before by the reverse-engineering transformation (e.g., some UI elements were not supported and event handling was also not available). For instance, imagemap (XHTML <map> tag) was an example of an element not supported.

Also, the possibility of handling *events* able to activate some functions in the application was not supported either. Indeed, in the desktop version, the Pac-Man can be controlled either using the keyboard (i.e., arrow keys can be used to specify directions) or by using an imagemap. In the latter case, the user can click a specific region of the image or even pass the mouse over such a region (event "onMouseOver") in order to activate a function for controlling the new direction that the Pac-Man should take.

The imagemap allowing the user to select a particular direction for the Pac-Man is reversed into a refinement of an activator element: the image_map concrete element. This element, at the concrete level, has a number of attributes to specify not only the attributes of the various image regions, but also the different mouse (mouse_activation_events) or keyboard events (key_activation_events) that can be triggered, which comprise another feature that can be specified with MARIA XML language.

As you can see, for the selection of the maze layout (as well as the animation speed of the Pac-Man) a radiobutton element has been used in the desktop version, which is reversed into an interactor of single selection type.

When migrating to the mobile device, the original page is split into two pages (see Figure 10). Indeed, the more limited capabilities of the target device do not allow all the elements to be included in one page, also because the maze is a 14×20 table and then, even if it is shrunk in order to fit the screen of the mobile device, it will occupy almost all the first mobile page.

The splitting strategy considers some aspects. For instance, in the desktop version, the imagemap used for controlling the Pac-Man is displayed in the same presentation in which the maze of the game is also rendered: This is done in order to allow the player to have instant feedback of the selected direction, to control the Pac-Man position, and to be able to react promptly when a ghost is approaching. This has to be preserved in the mobile version, since splitting the Pac-Man controls onto a different page from the maze would result in an unusable game (the user would have to navigate between different pages to control the game and check the updated situation in the maze).

However, in the desktop version of the game there are two possibilities for controlling the game (through the keyboard and also by using the imagemap). Thus, since a touch-screen mobile device is used as migration target, the keyboard controls and the imagemap are considered by the adaptation engine not suitable on this type of platform and they do not appear in the newly generated version. In this case, the support for controlling the Pac-Man is obtained through touch-based interactions, which are available on this kind of platform.

In addition, when migrating to the mobile device, some controls are redesigned in order to adapt them to the mobile device: For instance, radio buttons with several choices appearing in the desktop version (which are a refinement of single_selection objects) are replaced in the mobile device by pull-down menus. This adaptation has been performed thanks to the fact that such two elements (radiobuttons and pull-down menus) are both a refinement of the single_selection object. In this case the suitability of one element with respect to the other one is decided according to the platform considered: On the mobile platform, a pull-down menu is preferred in order to save space.

Apart from displaying the maze, the first mobile page will also contain the button allowing the user to start a new game and another button allowing the user to navigate to another page on the mobile device, which will include all the remaining elements (current Pac-Man lives, current level and score, settings for animation speed and smoothness).

Regarding the state of the game, it is preserved and then reactivated in the mobile device at the point where the migration was activated. Therefore, the current Pac-Man and ghost positions are saved, together with the current level and score and all the other settings that the player already selected in the desktop version. For instance, if a radiobutton (on the source device) has to be mapped (together with its associated state) onto a pull-down menu (on the target device), the process is basically the following one. The state is mainly an XML-based string in which there are couples (id, value) where id is the identifier of every (modifiable) XHTML element, and value represents the information about the current value(s) associated with such element. Therefore, the *state information* associated with an XHTML radiobutton having *id_name* as its identifier will provide the list of options that can be selected, and also the element (among the various options) that was currently selected at the time when the migration was activated. Since the concrete user interface is derived from such an XHTML version (through a reverse-engineering process), an XHTML radiobutton (included in the final UI) and having id_name as its identifier will be translated onto a CUI interactor of the type radio_button and having the same identifier (id_name). When the CUI radiobutton is semantically redesigned, the same identifier (id_name) will be maintained in a new CUI element, which is the result of the tranformation (e.g., a pulldown menu). Therefore, by accessing the state information via such an identifier, it will be possible to access the related state information and, by opportunely mapping it in the new element, provide the full specification of the new CUI object (a pull-down menu in our case) that also includes the state information.

8. CONCLUSIONS AND FUTURE WORK

In this article we have presented a language for describing user interfaces at different abstraction levels (and the associated tool that supports such language). The language lays its foundation from previous experiences. In the article we mainly highlight the characteristics of the new UIDL, which allows for supporting the new evolution towards service-oriented architectures in ubiquitous environments. More specifically, we have shown how this novel language can provide useful support at both design and runtime. We have presented specific support for applications based on Web services associated with annotations based on MARIA XML and a design space for composing user interface descriptions associated to different services. Moreover, we have discussed the use of the language in a software architecture able to support migratory user interfaces, also showing an application example (the migratory Pac-Man). Migratory user interfaces are able to exploit ubiquitous environments and thereby follow mobile users as they change devices while maintaining the interactive application state.

Future work will be dedicated to further empirically testing the usability of the corresponding tool and investigate further additions in order to make it suitable for end-user development, when the applications are developed by nonprofessional software developers.

REFERENCES

ABRAMS, M., PHANOURIOU, C., BATONGBACAL, A., WILLIAMS, S., AND SHUSTER, J. 1999. UIML: An appliance-independent XML user interface language. In *Proceedings of the 8th World Wide Web Conference (WWW).* Elseiver, 617–630.

BANDELLONI R., MORI, G., AND PATERNO', F. 2005. Dynamic generation of migratory interfaces. In *Proceedings Mobile Human-Computer Interaction Conference*. 83–90.

BLUMENDORF, M., LEHMANN, G., FEUDERSTACK, S., AND ALBRAYARK, S. 2008. Executable models for human-computer interaction. In *Proceedings of the XVth International Workshop on the Design, Verification and Specification of Interactive Systems (DSVIS'08)*. 238–251.

CALVARY, G., COUTAZ, J., BOUILLON, L., FLORINS, M., LIMBOURG, Q., MARUCCI, L., PATERNO', F., SANTORO, C., SOUCHON, N., THEVENIN, D., AND VANDERDONCKT, J. 2002. The CAMELEON reference framework. Deliverable 1.1, CAMELEON Project. http://www.w3.org/2005/Incubator/model-based-ui/wiki/Cameleon_reference_framework.

CHESTA, C., PATERNO, F., AND SANTORO, C. 2004. Methods and tools for designing and developing usable multi-platform interactive applications. *PsychNology J. 2*, 1, 123–139.

CLERCKS, T., LUYTEN, K., AND CONIX, K. 2004. DynaMo-AID: A design process and a runtime architecture for dynamic model-based user interface development. In *Proceedings of the Working Conference on Engineering for Human-Computer Interaction and International Workshop on Design Specification and Verification of Interactive Systems* (*EHCI/DS-VIS*). 77–95.

FOLEY, D. AND NOI SUKAVIRIYA, P. 1994. History, results, and bibliography of the user interface design environment (UIDE), an early model-based system for user interface design and implementation. In *Proceedings of Design, Verification and Specification of Interactive Systems (DSVIS'94).* 3–14.

JACOB, R., DELIGIANNIDIS, L., AND MORRISON, S. 1999. A software model and specification language for non-WIMP user interface. *ACM Trans. Comput.-Hum. Interact. 6*, 1, 1–46.

JOHN, B. AND KIERAS, D. 1996. The GOMS family of analysis techniques: Comparison and contrast. *ACM Trans. Comput.-Hum. Interact. 3*, 4, 320–351.

JOHNSON, P., WILSON, S., MARKOPOULOS, P., AND PYCOCK, J. 1993. ADEPT: Advanced design environment for prototyping with task models. In *Proceedings of the International Conference on Human Computer Interaction and ACM Conference on Human Aspects on Computer Systems (INTERCHI).* 56.

HELMS, J. AND ABRAMS, M. 2008. Retrospective on UI description languages based on eight years' experience with the user interface markup language (UIML). *Int. J. Web Engin. Technol. 4*, 2, 138–162.

LIN, J. AND LANDAY, J. 2008. Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces. In *Proceedings of the 26th Annual SIGCHI Conference on Human Factors in Computing Systems (CHI)*. 1313–1322.

MESKENS, J., VERMEULEN, J., LUYTEN, K., AND CONINX, K. 2008. Gummy for multi-device user interface designs: Shape me, multiply me, fix me, use me. In *Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI'08)*. 233–240.

LIMBOURG, Q., VANDERDONCKT, J., MICHOTTE, B., BOUILLON, L., AND LÓPEZ-JAQUERO, V. 2004. USIXML: A language supporting multi-path development of user interfaces. In *Proceedings of the Working Conference on Engineering for Human-Computer Interaction and International Workshop on Design Specification and Verification of Interactive Systems* (*EHCI/DS-VIS*). 200–220.

MORI, G., PATERNO', F., AND SANTORO, C. 2004. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Trans. Softw. Engin. 30*, 8, 507–520.

MYERS, B., HUDSON, S., AND PAUSCH, R. 2000. Past, present, future of user interface tools. *ACM Trans. Comput.-Hum. Interact. 7*, 1, 3–28.

PATERNO', F. 1999. *Model-Based Design and Evaluation of Interactive Applications*. Springer.
PATERNO', F., SANTORO, C., MÄNTYJÄRVI, J., MORI, G., AND SANSONE, S. 2008. Authoring pervasive multimodal user interfaces. *Int. J. Web Engin. Technol. 4*, 2, 235–261.
PATERNO', F., SANTORO, C., AND SCORCIA, A. 2008. Automatically adapting Web sites for mobile access through logical descriptions and dynamic analysis of interaction resources. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI)*. 260–267.
SALBER, D., ANIND, D., AND ABOWD, G. 1999. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 434–441.
SHAER, O., LELAND, N., CALVILLO-GAMEZ, E. H., AND JACOB, R. J. K. 2004. The TAC paradigm: Specifying tangible user interfaces, *Personal Ubiq. Comput. 8*, 5, 359–369.
SOTTET, J., CALVARY, G., COUTAZ, J., AND FAVRE, J. 2007. A model-driven engineering approach for the usability of plastic user interfaces. In *Proceedings of the Working Conference on Engineering Interactive Systems*. 140–157.
VAN DER VEER, G., LENTING, B., AND BERGEVOET, B. 1996. GTA: Groupware task analysis — Modelling complexity. *Acta Psychologica 91*, 297–322.

Received January 2009; revised June 2009; accepted July 2009