Model-based Design of Multi-Device Interactive Applications based on Web Services

Fabio Paternò, Carmen Santoro and Lucio Davide Spano

ISTI-CNR, HIIS Lab, Via Moruzzi 1,
56124 Pisa, Italy
{Fabio.Paterno, Carmen.Santoro, Lucio.Davide.Spano}@isti.cnr.it

Abstract. Creating an interactive application based on pre-existing functionalities poses a number of novel issues in the design process. We discuss a method and an associated model-based language, which aim to address such issues in multi-device contexts. One specific aspect of this method is the ability to compose user interfaces specifically for different services. In addition, the possibility to specify interactive objects, Web services accesses and scripts allows designers to describe Rich Internet Applications as well.

Keywords: Model-Based Design, Multi-device Environments, User Interface Design, Web Services.

1 Introduction

Model-based approaches for UI design are characterised by the use of some representations (models) of the aspects that are supposed to be relevant in the UI software lifecycle. This involves the identification and representation of the characteristics that are meaningful at each design stage, and mainly highlights one of the most difficult parts of the work: identifying what characterizes a UI without having to deal with a plethora of low-level implementation details that can distract the designer from the most important issues. After having identified such characteristics, the next issue is specifying them through suitable languages that can enable simple integration within software environments, so as to facilitate the work of the designers.
The design of interactive multi-platform systems has further stimulated interest in model-based approaches in HCI. In the design and development of such systems the use of model-based approaches has revealed to be useful, especially through the capture and modelling of different levels of abstractions in which it is possible to gradually move from aspects that are technology-neutral to more concrete, platform-dependent detailed aspects. In such a way it is possible to start with a general abstract vocabulary and then obtain concrete languages for each type of platform by just refining the abstract language.

However, recently, the design of such systems has become even more challenging. Indeed, not only must the same interactive application be accessible from different devices, within different contexts of use, but in addition the way in which such interactive applications are built/created has changed, since there is the need to reuse existing code to reduce development time and effort. An example of this can be seen

in the role that Web services are playing in the development of interactive applications. Indeed, the increasing availability of functional units within Web services has driven the need to develop methods that are able to exploit such pre-existing functionalities by including them into more composite interactive applications. In particular, some heterogeneous issues have to be faced by the designers in this case. First, the need to exploit some (generally small) legacy functionalities that were developed without accounting for human interaction, since they were basically intended to support computer-to-computer (service-to-service) communication. Therefore, the first issue is how to obtain the UI for such functionalities, possibly in a semi-automatic way, so that it can ease the designer's work. Secondly, even when a UI for such portions of functionalities is available, there is the issue of including and integrating pre-existing user interfaces associated with functionalities into new, composite ones, and possibly support the designer during such composition.

In the paper, after discussing some related work we describe the main features of our approach for designing user interfaces for Web services. We also introduce the dimensions of a design space for composing user interfaces in such context. Afterwards, we express the requirements that have driven the development of MARIA, an XML-based language for describing user interfaces at various abstraction levels. Then, we detail an example to show more concretely how the proposed approach is able to support the design of user interfaces for applications exploiting Web Services in multi-device environments. Lastly, some conclusions and directions for future work are provided.

2 Related Work

Several model-based approaches have been put forward in the field of multi-device UIs. A sign of the maturity of this area can be seen by the recent interest in defining connected international standards (e.g.: new W3C Group on Model-based User Interfaces: http://www.w3.org/2005/Incubator/model-based-ui/) and their adoption in industrial settings (e.g.: dedicated Working Group in the NESSI NEXOF-RA IP, http://www.nexof-ra.eu/).

In particular, a number of approaches have been proposed to support descriptions of logical user interfaces. UIML [1] was one of the first model-based languages targeting multi-device interfaces. It structures the user interface in: structure, style content, behaviour, even if it has not been applied to obtain rich multimodal-user interfaces. XForms [http://www.w3.org/MarkUp/Forms/] is a W3C initiative, and 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 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 within the form, and is mainly used for expressing form-based UIs, though it does not seem particularly suitable for

supporting other interaction modalities, such as voice. UsiXML (USer Interface eXtensible Markup Language) [3] is an XML-compliant markup language, which aims to describe the UI for multiple contexts of use. UsiXML is decomposed into several meta-models describing different aspects of the UI. There is also a transformation model that is used to define model-to-model transformations. The authors use graph transformations to support model transformations, which is an interesting academic approach, albeit with some performance issues. TERESA XML [5] defines several abstraction levels for expressing the characteristics of a user interface. Among such levels, one (the concrete interface) is specified through a number of platform-dependent languages. These are refinements of the abstract level, which describes the user interface using a platform-neutral vocabulary: interactors (describing single interaction objects), composition operators (indicating how to compose interactors), presentations (indicating the elements that can be perceived at a given time). Various modalities are supported through this approach. However, it does not support data or event models.

One issue with such model-based approaches is that they have not sufficiently addressed the recent increasing trend in software design towards building atomic software components, called Web services, which are available in distributed settings. Thus, applications have to be assembled starting from such pre-existing building blocks. Especially for enterprises this offers several advantages in terms of code reuse, increase productivity and leveraging integration processes. Some work has been dedicated to the generation of user interfaces for Web services [7, 8] but without exploiting model-based approaches. Previously, there have been approaches investigating the possibility of automatic generation with model-based support for applications based on Web services [4]. but such approaches work well only with not too complex cases and when the application domain is well-known. In [9] there is a proposal to extend service descriptions with user interface information. For this purpose the WSDL description is converted to OWL-S format, which is combined with a hierarchical task model and a layout model. We follow a different approach, which aims to support the access to the WSDL without requiring their substantial modification in order to generate the corresponding user interfaces, still exploiting logical interface descriptions. Therefore, model-based approaches have to cope with further requirements. There is less need to design an application from scratch, but they have to support interactive application development starting with small functionalities (services) that are already available, even if these were not built with that particular application in mind. In addition, there is a need to access the same service through an increasing number of device types (in particular mobile) available in the mass market, sometimes able to exploit a variety of sensors (e.g. accelerometers, tilt sensors, electronic compass), localization technology (such as RFIDs, GPS) and interaction modalities (multi-touch, gestures, camera-based interaction). This has further urged the identification of suitable universal declarative languages able to address such composite number of aspects in a comprehensive specification.

3 The Approach

A top-down approach essentially consists in breaking down and progressively refining an overall system into its sub-systems, thus it is particularly effective when the design starts from scratch. In such cases the designer has an overall picture of the system to be designed and can refine it gradually, without any particular constraints. However, when the designer wants to include already existing pieces of software as services, this necessarily requires that a bottom-up approach is considered in the design process in order to include and exploit not only such legacy, fine-grained functionalities, but also composite and higher level functionalities obtained by assembling the elementary ones. Therefore, the best option seems to be a *hybrid* solution in which a mix of bottom-up and top-down approaches is used.

Automatic or semi-automatic *composition* of user interfaces associated with various services is one important issue in this context. Indeed, the design and development of an interactive application based on pre-existing Web services is by definition driven by a composition-oriented approach. Not only must functionalities be composed (for this purpose various approaches already exist, e.g. BPEL, WS-BPEL) in order to provide arbitrarily complex functionalities, but also the corresponding user interface specifications associated with the elementary services (which can be provided through specific annotations) can be composed as well. In order to better understand how this composition activity can be carried out, we have identified a design space for this specific activity (see Figure 1).

Three main aspects have been identified as important in order to compose user interfaces: the abstraction level of the user interface description, the granularity of the user interface considered, and the types of aspects that are affected by the UI composition. Regarding the *abstraction level*, since a user interface can be described at various abstraction levels (task and objects, abstract, concrete, and implementation), it is straightforward that the user interface composition can occur at each of them. 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), groups of objects (for instance a navigation bar with a list of news), we can also compose various types of interface elements and groups to obtain an entire presentation, and we can compose presentations in order to obtain the user interface for an entire application. It is worth pointing out that by the term ‘presentation’ we refer to the set of user interface elements that can be perceived at a given time, a common example being a graphical Web page.

Lastly, we have to distinguish the compositions depending on the main *UI aspects* that they affect, which are: i) the dynamic behaviour of the user interface, which means the possible dynamic sequencing of user actions and system feedback (e.g.: when some elements of the UI appear or disappear depending on some conditions); ii) the perceivable UI objects (for example in graphical user interface we have to indicate the spatial relationships among the composed elements); iii) the data that are manipulated by the user interface.

More specifically, in the proposed approach first a bottom-up step is envisaged, in order to analyse the Web services providing functionalities useful for the new application. We then specify the application task model in ConcurTaskTrees (CTT) [6], a standard de facto for task model specification (http://giove.isti.cnr.it/ctte.html).

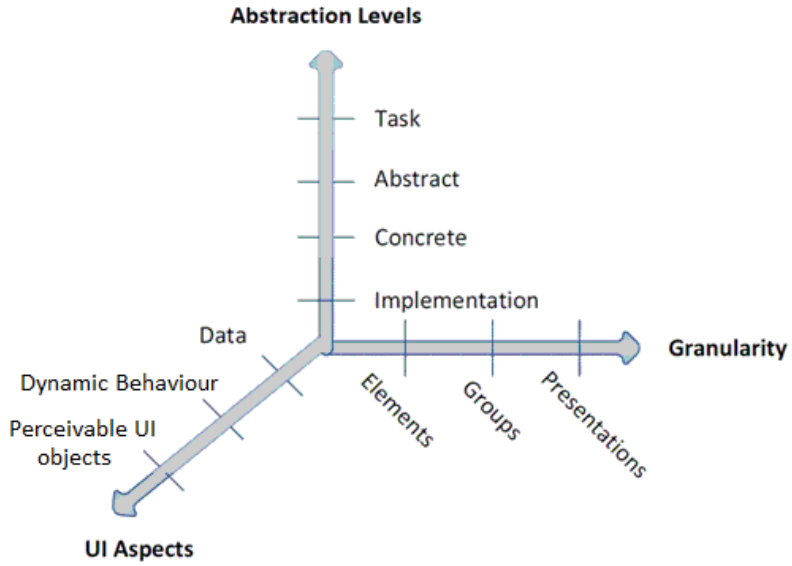

Fig. 1. The Design Space for UI Composition.

The Web services can be seen as a particular type of task (system tasks, namely tasks whose performance is entirely allocated to the application), and the temporal relationships that are specified in a task model indicate how to compose such functionalities. The specification of the task model should be driven by the user requirements, and it also implies some constraints on how to express such model. Indeed, in order to address the right level of granularity, not only will a Web service be associated with an application task, but it is useful that each operation of the Web service be associated to a specific system 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). Such system tasks are related to the user and interaction tasks in the overall task model.

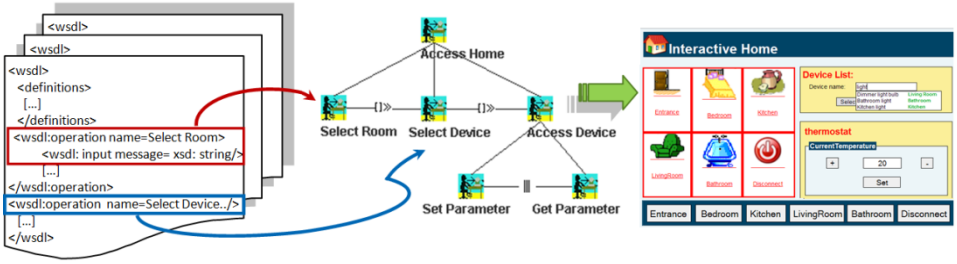

Fig. 2. The Approach

After having performed this step, we have obtained a level of composition, which also involves the functionalities associated with the Web services. The result is a structured model in which such functionalities have been progressively organised in a hierarchical task model, which includes system tasks associated with Web services and their operations. At this point, once we have obtained the task model it is possible (through a top-down step) to generate the various UI logical descriptions, and then refine them up to the implementation, by using the MARIA language (the final phase in Figure 2).

4 MARIA

Based on the lessons learned from the analysis of the state of the art and previous experiences conducted by various groups with TERESA (see [2] for a test in an industrial setting), we have identified a number of requirements for a new language suitable to support user interfaces in ubiquitous environments.
In particular, the following requirements have been identified for the new language:

- providing the designer with higher control of the user interface produced, also through an event model;
- a more flexible dialogue and navigation model, also supporting parallel interactions;
- a flexible data model, which allows the association of various types of data to the various interactors;
- support for recent dynamic techniques, such as ajax scripts;
- streamlining the specifications of the abstract and concrete languages, in order to make the specifications shorter and more readable.

4.1 Main Features

A number of features have been included in the language:
a) *introduction of data model*
We have introduced an abstract description of the underlying data model of the user interface, needed for representing the data (types, values, etc.) handled by the user interface. Indeed, by means of defining an abstract data model, the interactors (the elements of the abstract or concrete user interface) composing an abstract [concrete] user interface, can be bound to a specific type or an element of a type defined in the abstract [concrete] data model. The introduction of a data model also allows for more control over the admissible input that can be provided to 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: doing some correlations between the values of interface elements (for instance, the value of one element can vary depending on the value of the another element), conditional presentation connections (triggering the activation of a presentation depending on a certain value associated to an interactor), conditional layout of interface parts (showing or not a portion of a presentation depending on the value associated to a UI

element), specifying the format of the input values (depending on the data type it is possible to specify a certain acceptable template for input values associated with that data type), application generation from the interface description (having information on the values associated with a UI description enables the actual generation of a working application).

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 at different abstraction levels how the user interface is able to respond to events triggered by the user. In MARIA XML two types of events have been introduced:

i) *Property change events*: events that change the status of some UI properties (e.g. when a user selects an element in a drop-down menu then the text label of a text field changes accordingly).
ii) *Activation events:* some interactors can raise events with the purpose of activating some application functionality (e.g. access to a database or to a web service).

c)*Support for Ajax scripts, which allow continuously updating of fields*
Another aspect that has been included in MARIA is the possibility of supporting continuously updating of fields at the abstract level. To this aim we have added an attribute to the interactors: continuosly-updated= "true"["false"]. The concrete level has the duty to provide more detail on this feature, depending on the technology used for the final UI (Ajax for web interfaces, callback for standalone 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.

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 to be able to affect both how the UI elements are arranged in a single presentation, and how it is possible to navigate between the different presentations. Therefore, the content of a presentation can dynamically change (this is also useful for supporting Ajax techniques). In addition, it is also possible to specify a dynamic behaviour that changes depending on specific conditions: this has been implemented thanks to the use of conditional connections between presentations.

In the next sections we provide a more detailed description of concepts/models that have been included in MARIA, both for the Abstract UI and the Concrete UI.

4.2 MARIA – Abstract Level

The advantage of using an abstract description of a user interface is that designers can reason in abstract terms without being tied to a particular platform/modality/implementation language. In this way, they have the possibility to

focus on the *semantic* of the interaction (namely: what the intended goal of the interaction is), regardless of the details and specificities of the particular environment considered. Figure 3 shows the main elements of the abstract user interface meta-model (some details have not been shown for readability reasons). An interface is composed of one data model and one or more presentations. The presentation includes a data model and a dialog model, which contains information about the events that can be triggered by the presentation in a given time. The dynamic behaviour of the events is specified using the CTT temporal operators. 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 (e.g. an event occurring on an interactor can affect the behavior of another interactor, by e.g. disabling the availability of an event associated to another interactor). The dialog model can also be used to describe parallel interaction between the user and the interface. A *connection* indicates what the next active presentation will be when a given interaction is performed and it can be either an elementary connection, or a complex connection (when a Boolean operator composes several connections) or a conditional connection (when various conditions on connections are specified).

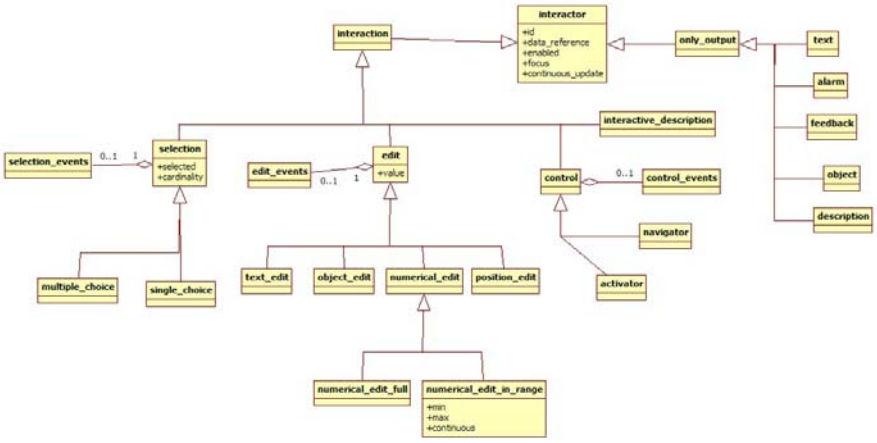

Fig. 3. An overview of the AUI metamodel

There are two types of *interactor composition*: *grouping* or *relation*, the latter has at least two elements (interactor or interactor compositions) that are in relation to each other. An interactor (see Figure 3) can be either an interactor object or an only_output object. The first one can assume 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. The 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 next section for some examples). 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.

4.3 MARIA – Concrete Level

The concrete description is aimed at providing a platform-dependent but implementation language-independent description of the user interface. It assumes that there are certain available interaction resources that characterise 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. Thus, for example, if at the abstract level there is a single selection object at the concrete level, this can be refined into a radio-button or a drop-down menu or a list (in case of a graphical platform) but it can also be refined into a vocal selection or gesture-based selection if different platforms are addressed.

In order to enhance the readability of the language and also for consistency reasons (cross-references between different models enabling more consistency because they avoid to replicate the same data in two different places), we decided to furnish the concrete user interface only with the details of the concrete elements, leaving the specification of the higher hierarchy in the abstract meta-model. At this level differences associated with the specific characteristics of the platform will be modelled. For instance, when focusing on a iPhone platform the concrete user interface language has to express the fact that interaction is carried out through the use of not only a simple touch-based interface (which is also to some extent available on PDA), but it also has to handle *multi-touch* events. Therefore, on this platform, there is the need of introducing and modelling a different group of events, the so-called *touch property events*, which includes *touch start* (activated when a finger tap the screen surface), *touch move* (triggered when a finger moves on the surface), *touch end* (activated when a finger leaves the screen surface). In addition, the *zoom gesture event* (which is done through a multi-touch interaction) notifies that a zoom command has been recognized by the system and contains the scale factor that should be considered for zooming. Another peculiar characteristics of the iPhone is the existence of an accelerometer. In this case, the concrete user interface language has to support the specification of the current screen orientation and also to support the associated events.

More generally, the flexibility introduced at the abstract level is reflected also at the concrete level. Thus, for example, there is no more a rigid separation between interface elements for activating functions and elements supporting a selection (as it happened in traditional model-based approaches) but it is possible to model a radio-button, which is associated with different functionalities depending on the selected element.

5 Example Application

As an example application of the features of MARIA XML we consider a home application in which users can control some interactive devices and appliances. In this home application we focus on a specific subset of functionality for demonstrating some of the MARIA XML features. In particular, we focus on i) the possibility to provide suggestions for searching a device through a text editing interactor (for example, the user enters a part of the device name and some suggestions for the completion appear) and ii) displaying information on a set of appliances in a part of the presentation while the user can dynamically add or remove elements from the appliance set.

Regarding the first aspect, let us consider in the home scenario a web service which, given a string, returns a list of suggestions for selecting an appliance that matches the input string. For modelling such a situation we need at the abstract level: an edit interactor for receiving the input string from the user; when the user enters the text, we need to express that the web service has to be invoked and a selection interactor must be populated with the web service output.

To explain how it is possible to model such interaction at the abstract level, we use the following MARIA XML features:

- Abstract events on interactors (to detect the change in the input string);
- Syntax for expressing external functions calls;
- Binding between the UI model and the data model within the UI definition.

First of all we need to “import” the Web service into the UI definition. This is possible using the external functions introduced before. An external function is an abstract representation of services and functionalities that are not defined in the UI (such as Web services or database access). When an abstract function is declared, it can be called by the abstract scripts to express how the interface should use the output of these functions. The following XML excerpt shows a possible abstract representation of the suggestion service:

```
<aui:external_functions>
    . . .
  <aui:function name="getSuggestions" type="web_service">
    <aui:output type="UserSession/suggestions" />
    <aui:input type="xs:string" name="inputString"/>
  </aui:function>
    . . .
<aui:external_functions>
```

The external_functions tag contains all the external function declarations. A single function is declared specifying: a name (e.g. getSuggestions); its type (such as Web service, database, code etc); its output type (in this case we presume a data type UserSession in the data model that contains an element *suggestions*, which is the suggestion list and corresponds to the external function output type); its parameters (in this case the input string).

Now we can describe that when the input string changes, the external function must be called and the suggestions must be displayed. To this end, we use the value changed event of the text_edit interactor. When this event occurs, the function is called using an abstract script, and the *hidden* property of the choice interactor (a single choice in this case) is changed to false. The following excerpt is the definition of the text edit interactor:

```
<aui:text_edit id="device_search">
  <aui:events>
    <aui:value_change>
      <aui:handler>
        <aui:script>
          <![CDATA[
            data:UserSession/suggestions
=external:getSuggestions(ui:device_search.value); ]]>
[…]
      <aui:handler>
        <aui:change_property interactor_id="device_suggestions"
                    property_name="hidden" property_value="false" />
      </aui:handler>
[…] </aui:text_edit>
```

The above definition states that when the input text, which is specified by using an abstract object of type text_edit with id=”device_search”, changes (this is specified by the fact that the event type is “value_change”), the field suggestions of the UserSession data type (see “UserSession/suggestions” field above) is populated with the output of the external function getSuggestions, invoked by passing the input text value, see “external:getSuggestions(ui:device_search.value);” in the excerpt above. After the function call, the device_suggestion interactor (a single choice interactor) has to be shown. This interactor is bound to the same data field populated by the external function invocation so, when this field changes, the interactor is also updated with the new options. The following excerpt contains the single_choice interactor definition:

```
<aui:single_choice  id="device_suggestions"
data_reference="UserSession/suggestions" >
  <aui:events>
   <aui:selection_change>
    <aui:handler>
     <aui:change_property interactor_id="device_select_activator"
                    property_name="enabled"
                    data_value="true" />
     <aui:change_property interactor_id="device_monitor_activator"
                    property_name="enabled"
                    data_value="true" />
     <aui:change_property interactor_id="device_search"
                    property_name="value"
```

```
                data_value="ui:device_suggestions.selected" />
    <aui:change_property interactor_id="device_suggestions"
                property_name="hidden"  property_value="true" />
[…]
 </aui:single_choice>
```

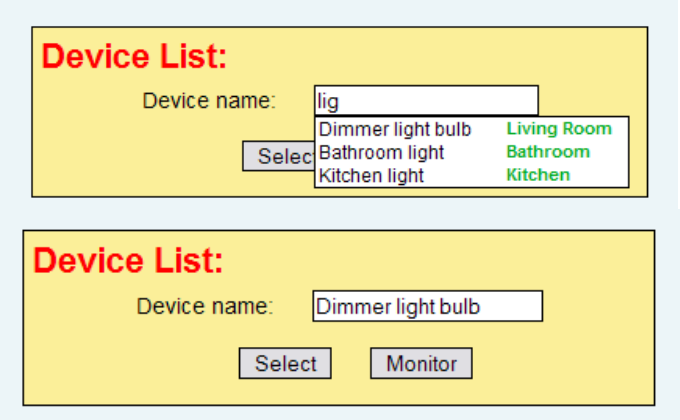

Fig. 4. The interaction modelled in the example

The interactor is bound to the data using the data_reference attribute. When the selected element changes, it enables two activators (activator is the interactor type that models interface elements dedicated to activate functionalities): one for getting the control panel for the device and the other for monitoring it (see *Select* and *Monitor* buttons in Figure 4). Then it completes the input text of the text_edit presented before (setting the value attribute with its selected value) and hides itself. Note that the specification is completely abstract, it is not specified how the service is called, how the interactors are hidden or shown and what the UI platform is.

We can refine the interface definition to various concrete platforms and final implementations. The interface can be adapted to the target platform capabilities (screen size, processor speed etc) and interaction techniques (mouse, multitouch, vocal commands etc). Figures 5 and 6 show two possible final implementations (obtained passing through a concrete description generation step) of the same abstract user interface for two devices (desktop and iPhone).

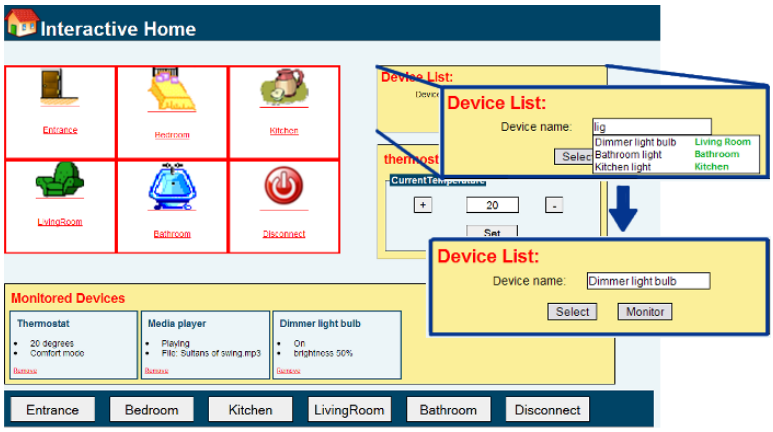

Fig. 5. Example implementation for desktop platform.

However, the differences between a desktop computer and the iPhone can require a different number of presentations for the same content and also different locations of the groups in the screen (in the figure the controls for the selected device and the list of monitored status in the iPhone is in a different page and the groups have a flow layout). However, the suggestion mechanism is the same in both devices (although it can be implemented in different ways) and this aspect is reflected in the abstract description.

Fig. 6. Example implementation for the iPhone platform.

6 Conclusions, Future Work and Acknowledgments

In this paper we present our method for developing interactive applications based on the access to Web services. The described approach exploits a multi-layer framework of languages for describing UIs through a mix of bottom-up and top-down phases. This allows designers to develop service front-ends for Web services, which were originally developed without exactly knowing the interactive applications that will access them. We have also discussed how the MARIA language is able to support specification of flexible interactions exploiting such Web services and scripts, for then generate implementations for different types of devices. This type of interactions are becoming widely used in Web 2.0 and Rich Internet Applications.

We are developing an authoring environment to support the various phases of the method presented, including the association of system tasks with Web services and their operations, and ease the use of MARIA and the associated transformations. We also plan to integrate in MARIA some concepts of the WAI-ARIA (Accessible Rich Internet Applications, http://www.w3.org/WAI/intro/aria) in order to support generation of user interfaces accessible to disabled people, such as blind people interacting through screen readers.

We gratefully acknowledge support from the EU ServFace Project (http://www.servface.eu).

References

1. Abrams, M., Phanouriou, C., Batongbacal, A., Williams, S., Shuster, J. UIML: An Appliance-Independent XML User Interface Language, Proceedings of the 8th WWW conference, 1999.
2. Chesta, C., Paterno, F., Santoro, C. (2004): Methods and Tools for Designing and Developing Usable Multi-Platform Interactive Applications. In Psychnology, 2 (1) pp. 123-139
3. Limbourg Q., Vanderdonckt J., Michotte B., Bouillon L., Lopez-Jaquero V. USIXML: A Language Supporting Multi-path Development of User Interfaces. EHCI/DS-VIS 2004: 200-220
4. Mori G., Paternò F., Spano L. D.: Exploiting Web Services and Model-Based User Interfaces for Multi-device Access to Home Applications. Kingston, Canada, July 2008, DSV-IS 2008, Springer Verlag, LNCS, pp.181-193.
5. Paternò F., Santoro C., Mantyjarvi J., Mori G., Sansone S., Authoring Pervasive MultiModal User Interfaces, International Journal of Web Engineering and Technology, Inderscience Publishers, 4(2) pp.235-261, 2008.
6. Paternò F., Model-Based Design and Evaluation of Interactive Applications, Springer Verlag, 1999.
7. Song, K., Lee, K.-H., 2008. Generating multimodal user interfaces for Web services, Interacting with Computers, Volume 20, Issues 4-5, September 2008, Pages 480-490
8. Spillner, J., Braun, I., Schill, A., 2007. Flexible Human Service Interfaces, Proceedings of the 9th International Conference on Enterprise Information Systems, 79-85.
9. Vermeulen J., Vandriessche Y., Clerckx T., Luyten K. and Coninx K., Service-interaction Descriptions: Augmenting Services with User Interface Models, Proceedings Engineering Interactive Systems 2007, Salamanca, Springer Verlag.