Support for Authoring Service Front-Ends

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

ABSTRACT
The success of service-oriented computing has important implications on how people develop user interfaces. This paper discusses a method for supporting the development of interactive applications based on the access to services, which can be associated with user interface annotations. In particular, we show how model-based descriptions can be useful for this purpose and the design of an authoring environment for the development of interactive front-ends of applications based on Web services. A prototype of the authoring environment is presented.

Categories and Subject Descriptors
H5.m. Information interfaces and presentation (e.g., HCI).

General Terms
Design, Human Factors, Languages

Keywords
User Interface Composition, Model-based Design, Web services.

INTRODUCTION
Service-oriented solutions are becoming more and more adopted in the area of software engineering. There are mainstream approaches able to describe the workflow of applications exploiting compositions of such services (see for example BPMN, http://www.bpmn.org/), which can then be translated, to some extent, into executable descriptions, such as in WS-BPEL. However, they provide little support to describe the interactive part of an application. One specific characteristic of such interactive applications is that they have to be developed exploiting pre-existing functionalities implemented through Web services. The functional interface of such functionalities is described through WSDL (Web Services Description Language) files, which are XML-based descriptions indicating what operations are available and the associated input and output parameters and data types. Often the people who develop interactive applications are different from those who implemented the Web services. Thus, in order to facilitate the work of the UI designers, the Web services can be annotated with user interface hints whose level of detail can range from simple label suggestions to complex user interface specifications.

Another approach is to automatically generate the user interfaces corresponding to the Web services through rules mapping WSDL descriptions into user interface descriptions (see for example [6]). However, this approach produces reasonable results only when the application domain is well known.

In this paper, after discussing related work, we present the proposed methodological approach for addressing such issues. We introduce the main features of the new model-based language used and describe the tool supporting the method. We also provide a small example to better illustrate the features of the approach. Lastly, we draw some conclusions along with indications for future work.

RELATED WORK
A number of approaches have already been proposed for composing services in such a way that the output of one service is the input for another one, thus we will not address this issue. For example, WS-BPEL [7] (Business Process Execution Language) is an XML language for describing and executing business processes. It consists of the composition of various activities (building block of processes such as variable assignment, wait, raise exception etc.), using usual structured programming constructs. It can be used for the composition of Web services at the business level, but it does not include specific information on the user interface for the service access. An extension of WS-BPEL is BPEL4people [1] , which tries to add a specification for the user interaction into the business process. It introduces the people activity, which is performed by a human-being. However, the interaction is defined always at the business level: a logical description of the user interface is not in scope for BPEL4People.

Some work has been dedicated to the generation of user interfaces for Web services [11, 12] but without exploiting model-based approaches. In [13] there is a proposal to extend service descriptions with user interface information also exploiting model-based approaches. 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 modifications in order to generate the corresponding user interfaces, still exploiting logical interface descriptions.

Since often concrete logical descriptions are specified through XML-based user interface languages [5], there have been proposals to use XML tree algebra-based techniques [2], for composing concrete presentations or portions of it. For example, in [3] the general XML tree algebra is applied to user interface composition and decomposition of graphical user interfaces specified in a XML-based language (UsiXML) [4]. That work includes operations for combining interactors, i.e. the fusion (composition with repetition of the intersection) and the union (composition without repetition of the intersection). However, that method does not take into account the possible temporal constraints in the interactions.

Task models, such as those described by ConcurTaskTrees (CTT) [10], can overcome such limitations also thanks to the rich set of temporal relationships that they allow designers to express. In terms of granularity, tasks can be elementary tasks (namely: tasks that are considered as a logically atomic entity which cannot be

EICS'09, July 15–17, 2009, Pittsburgh, PA, USA.

further refined) and structured tasks (namely, tasks whose specification consists in a decomposition and refinement of a number of logically connected, smaller sub-tasks). It is also possible to identify task patterns: they are reusable structures in task models, which can be used in various applications. Thus, whenever designers realise that the problem they are considering is similar to one which has already been found and solved, then they can reuse the solution previously developed.

In the next section we describe how such task models can be exploited in the design of interactive applications based on Web services.

METHOD

Since our goal is to support applications based on Web services, a traditional top-down approach going through the various abstraction layers does not seem particularly effective. There are many Web services already available and their exploitation is supposed to be able to include such existing third party functionalities, rather than being able to design a dedicated application from scratch, as a top-down approach generally does. Indeed, differently from traditional approaches in which design and development of dedicated pieces of functionalities are assumed, in the case of Web services the idea is to create interactive applications by accessing application functionalities developed by others. The basic building blocks of the system already exist and this imposes that a bottom-up stage should also be included. Therefore, the solution that has been envisaged is to have first a bottom-up step in order to analyse and include in the design the Web services providing functionalities useful for the new application to develop.

In order to do this we envisage a number of steps. First, as we plan to describe the composition of the user accesses to the various services using a task model language (e.g. CTT), we have to perform an association between the elementary tasks that we want to include in the task model and the operations specified in the Web services. The task model is supposed to express how the interactive application assumes that the tasks are carried out. Then, if some elementary (system) tasks are associated with the relevant Web Services, we can provide useful indications about how the Web services, and the associated user interface annotations, if any, should be exploited. It is worth pointing out that the development of the task model is generally carried out by a multidisciplinary team in which various roles/stakeholders are involved and is largely driven by user requirements.

Some rules can be followed in order to perform such associations in a consistent way. As said, since Web services are application functionalities, they will be associated with system tasks. In addition, it will be important to use within the task model a level of granularity that is suitable to expressing the details of the functionalities described in the Web services. Then, beyond associating system tasks to Web services, it is important to further decompose such system tasks into system sub-tasks in which each such subtask will be associated with an operation defined in the web service. Thus, if a Web service supports three operations, then there would be three basic system tasks.

Once such associations are performed, the resulting task model provides a description of how the various activities are supposed to be carried out within the interactive application (exploiting Web services associated to some application tasks). This is the result of the intersection of a bottom-up step with the initial, top-down phase consisting of the development of the hierarchical task model. Subsequently, the task model thus obtained will be used to derive, through a further top-down stage, a first draft of the user interface at a logical level. The information contained in the task model will be used to derive a first draft of the user interface at an abstract level (which means in a platform-independent way), which will then be refined into more concrete terms (namely, platform-dependent), until a final user interface description is defined in a platform-dependent implementation language. However, it is worth pointing out that in the current approach customized for Web services, an additional piece of information will also be used to derive the user interface implementation. This is represented by the so-called service annotations, which are pieces of information associated to Web services and aimed at providing some indications that can be useful for rendering the user interface. Examples of annotations are labels that are suggested for presenting the associate data, which may be platform dependent (e.g. short version for mobile devices). Another example is an annotation that provides more concrete information regarding the data types considered. For example, the Web service operator can have a string data type, but the annotation indicates that it is an enumerated value, which at the interface level does not require an editing string interaction object (e.g. a text box), but a selection interactor (e.g. a pull-down menu). There should be an analysis of the operations and the data types associated with the input and output parameters of the Web services considered. This has to be done in order to associate them with suitable abstract interaction objects.

At the abstract level it is possible to compose user interface elements by identifying either single groups of logically connected elements or relations among groups of elements. Groupings and relations are considered *abstract composition operators*. Groups, relations, and elements can be composed into presentations. Each (abstract) presentation identifies the elements that will be rendered at the same time. In addition, a way to compose entire presentations is using *abstract connections*, which specify the temporal order according to which they have to be made available to the user. When moving to concrete descriptions (which assume the existence of a given platforms, but are still implementation language independent), the abstract concepts are refined in a platform-dependent manner. For example, in a graphical user interface the techniques for grouping can be the use of the same colour, the alignment of the elements, the use of graphical containers and so on.

In order to support our method we have designed a new authoring tool, which provides a rich set of functionalities. The software modules composing the architecture of our authoring environment are shown in Figure 1. The FUI indicated in the figure is the Final User Interface (which is the implementation of the user interface).

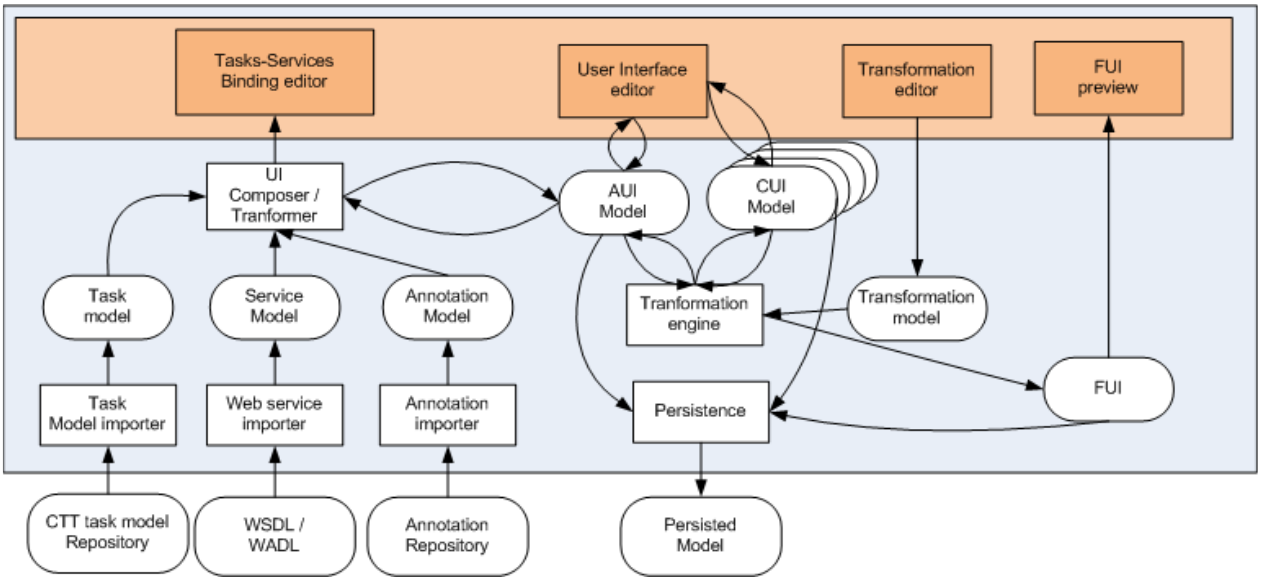

Figure 1: The software architectural components of the tool

As it can be seen, some basic modules are provided to the UI developers (see top part of the figure). The first one allows them to perform the associations between tasks and Web services. The second one supports editing of the user interface at various abstractions levels. The third one allows for setting up a number of mappings for model transformation that can vary depending on the different designer requirements and needs. The opportunity of having transformations that are not hardwired in the code enables the designer to modify such transformations easily and then obtain an environment that includes such mappings in the supported transformations. Lastly, the possibility of previewing the user interface generated is supported.

MARIA

MARIA is a new model-based language, which inherits the multilayer approach of TERESA [9] with one language for the abstract description and multiple platform-dependent concrete languages refining the abstract one depending on the interaction resources at hand. At the concrete level it is necessary to identify platform-dependent techniques for the interface elements and for making groups and relations perceivable to the user. For example, in tangible interfaces physical proximity can identify a group of elements and trigger a functionality when the group is dynamically created, while in a graphical interface attributes such as colour, alignment, and containers are used to indicate a group of elements logically related to each other.

With respect to TERESA, a number of new features have been included in the new language. In MARIA we have introduced an abstract description of the data model associated with the user interface, which is needed for representing the data (types, values, etc.) handled by the UI. Indeed, by means of defining an Abstract Data Type model, the interactors (the elements of the abstract or concrete user interface) composing an abstract [concrete] user interface, are connected either with a specific type or with an element of a type defined in the abstract [resp.:concrete] data model.

In addition, the introduction of a data model also enables for more control over the admissible operations that will be carried out on the various interactors. The introduction of a data model allows for better supporting the format of the various input values. Further advantages of having a data model are also the possibility of correlating the values of interface elements, supporting conditional presentation connections, and specifying conditional layout of interface parts. For example, we can express the case when depending on the value of a selection object a different presentation is accessed. In MARIA XML the data model is specified using the XSD type definition language.

Another aspect that has been included in the new language is represented by the support for features that are typical, for instance, of complex javascript codes/Ajax scripts, which allow continuously updating of fields. Indeed, we have introduced the continuously-updated Boolean attribute to the interactors. 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.).

Furthermore, an event model at abstract/concrete levels has been included in the language. 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 particular, in MARIA XML two types of events have been introduced: i) property change events: events that change the status of some UI properties. The handlers for this type of event indicate in a declarative manner how and under what conditions property values are changed; ii) activation events are events with the purpose to activate some application functionality (e.g. access to a database or to a Web service).

Another feature that has been included in MARIA XML is the possibility to express the fact that only some parts of a UI presentation can dynamically change (this is also useful for supporting Ajax techniques). In addition, it is also possible to specify dynamic behaviour that changes depending on specific conditions: this has been implemented thanks to the use of conditional connections between presentations. More detailed information on MARIA is available in [8].

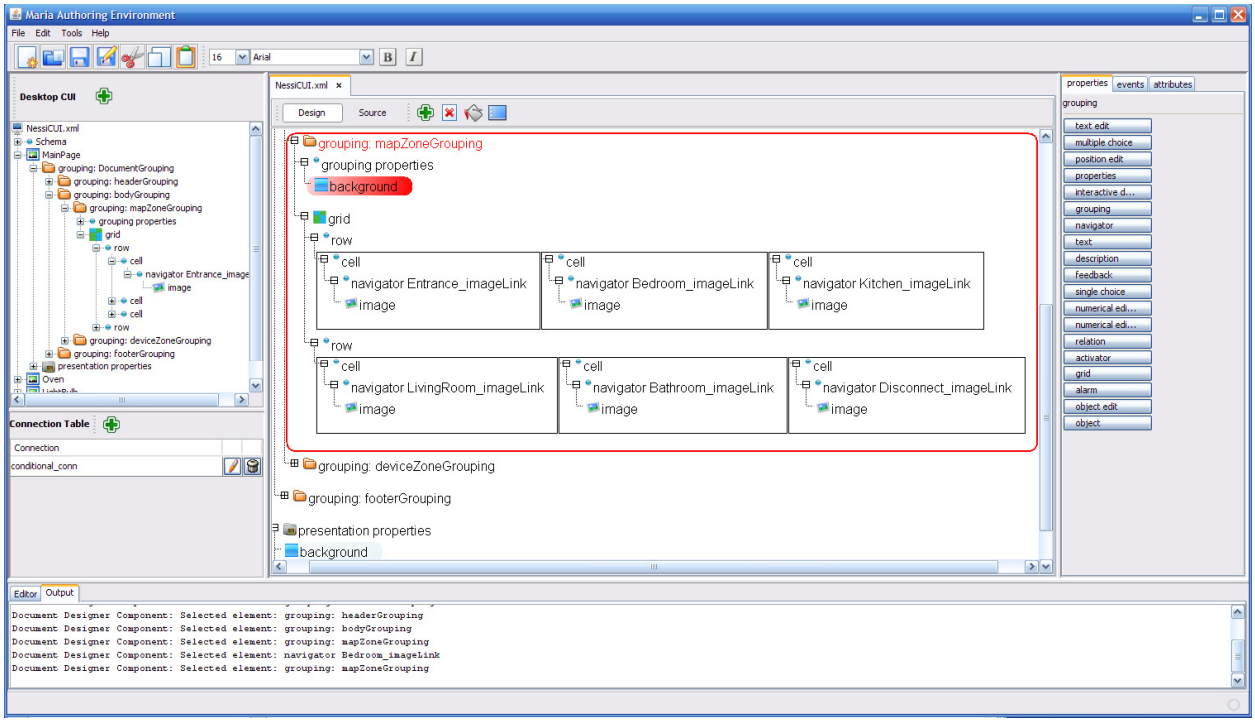

Figure 2: The Abstract/Concrete User Interface Specification Editor

TOOL SUPPORT

The authoring environment supporting the method proposed is a tool composed of three main sub-environments, with in addition the possibility of previewing the implemented user interface.

The first one, "Tasks-Services Binding Editor", is aimed at supporting the associations between the tasks included in the model corresponding to the application to be developed and the Web services that the designer wants to include. In order to do this, the designer has to access the repository of task models and the URI where the Web services are made available. In addition, within this module, it is also possible to import some annotations associated to the Web service considered. Such annotations provide further information about the part of the user interface associated with the Web service. Once such task-Web service associations have been carried out, a dedicated module ("UI Composer/Transformer") is then able to produce a first draft of the corresponding Abstract/Concrete User Interface (AUI/CUI) description by exploiting such various pieces of information (tasks, web services, annotations).

The logical descriptions thus obtained are the output of the first module and, in turn, the main input to another module (the "User Interface Editor") which is specifically aimed at supporting designers in refining the logical descriptions depending on the specific needs and requirements of the application considered. Such User Interface Editor module exploits the "Transformation engine" module to obtain a concrete description from an abstract one, and then a user interface implementation from a concrete description.

The tool is designed to contain a set of generators, each of them implements a transformation that delivers a UI written in a specific platform-dependent description language. The rules included in the Transformation engine are defined in a specific model, the "Transformation model", which allows for specifying the transformations that enable passing from a UI description to a more concrete one. The usefulness of having a Transformation Editor as a separate module lies in the enhanced flexibility for designers to easily specify the transformations to be supported from time to time and avoid having them hardwired in the code.

Figure 2 shows the environment for editing a concrete specification: the left part contains an interactive tree diagram of the available presentations, each one with its interactors and interactor compositions defined in the model. The central part is a direct manipulation interface for editing the user interface model, where each interactor composition is a container for different interface elements. The interface elements can freely be added by drag-and-drop in the logical description: the right part of the interface is a toolbox for adding new instances of interactors to the model. It shows only the allowed elements for the currently selected element, for example in Figure 2 one grouping has been selected and the toolbox lists the interactors that can be added to it. The user can also edit the interactor attributes through the attribute list on the second tab, or set the event handlers through the corresponding tab. In the left-bottom part it is possible to specify the possible navigations across various presentations through the various connections.

Figure 3 shows the interface of the Tasks-Services Association Editor. The main part contains the CTT model using a hierarchical tree representation: the children of a node are the decomposition of the parent. The nodes at the same level are connected using different temporal operators, which indicate the dynamic behaviour of the various tasks. Each task is categorized as Abstraction, User, Interaction or System. The System tasks can be bound to Web service operations from the repository on the right, where different services with their operations and data types are listed. To import these descriptions, the designers must simply specify the service Internet address. The CTT model enhanced with the Web service associations and annotations, which are represented on the left, is the starting point to generate the corresponding abstract and concrete descriptions, which can be modified by the designer using the associated editor.

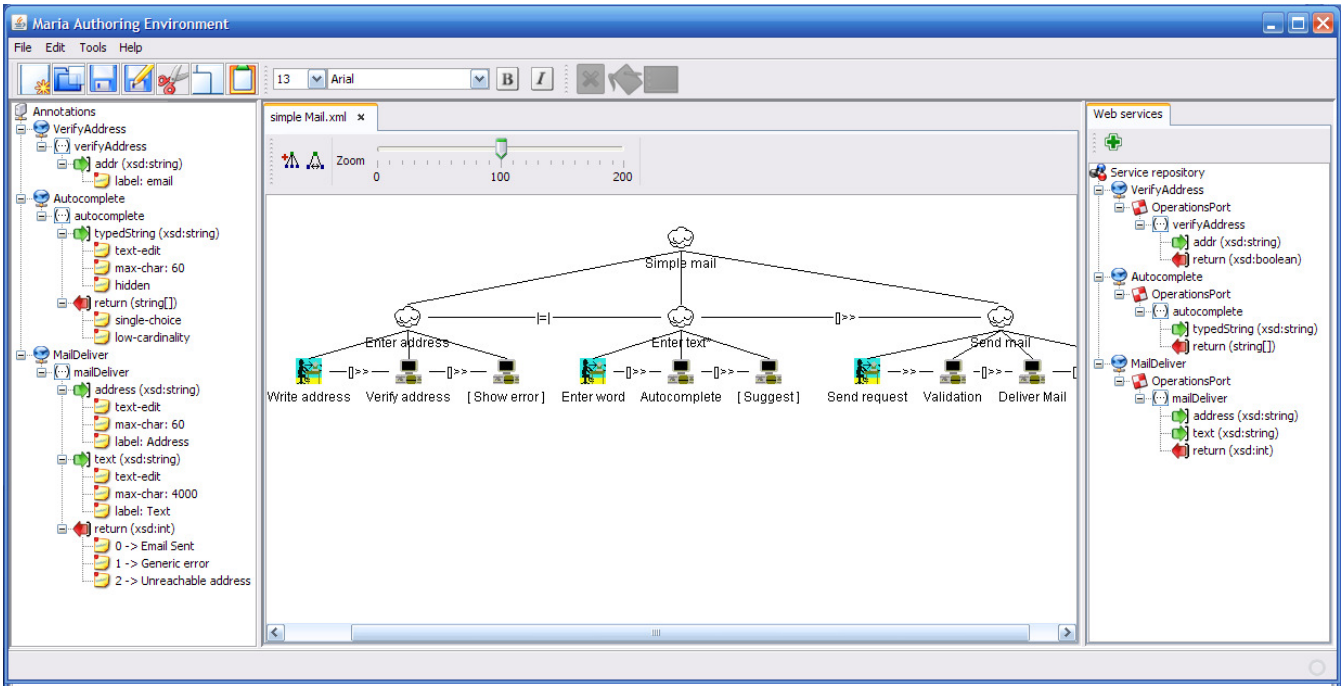

Figure 3: The Task-Service Association Editor

EXAMPLE

In this section we present an example of use of annotations and compositions in the design of UIs for Web Services. For the sake of clarity, we consider a simple example in which the user is supposed to provide information for sending out an email, which is actually an excerpt extracted from a larger application. One excerpt of the task model is visualised in Figure 4. Here we analyze the methodological steps for a set of tasks for sending an email, with address verification and auto text completion.

The system has to provide three functionalities: the address validation, the word suggestions and mail delivery. These functionalities can be implemented by three operations in three different Web services, which are completely independent. Regarding the following described services it is worth pointing out that i) the services input and output parameters are simplified versions of existing services; ii) the operations are part of three different services that are probably from different providers and are not designed to work together. The services are:

- *Verify address*
  http://providerone/verifyService/service.wsdl

  operation: Boolean verifyAddress(string addr). The operation will return true if the address is valid, false otherwise

- *Autocomplete*
  http://providertwo/AutocompleteService/service.wsdl;
  operation: string[] suggestCompletition(string typedString). The operation will return an array of suggested words, with a maximum length.

- *Deliver Mail*
  http://providerthree/MailDeliverService/service.wsdl

  operation: int sendMail(string address, string text). The operation sends an email to the specified address, and returns the result of the operation (for example 0 for success, 1 for unreachable address etc.)

The task-service association binds the system tasks to the Web services operations using the Web service pane of the editor and specifies the connections between the tasks and the Web service parameters. In particular, the authoring environment has a Web service browser where the developer can specify the URL of the WSDL file for inspecting operations (with input and output parameters) and data types defined for invoking the service. Then, s/he can load annotations for the selected Web service (if any) for supporting the logical user interface generation process.

In the example considered, we have the following annotations for the three operations:

- Verify address [input] addr: string (label: email)
- Verify address [output] Boolean
- SuggestCompletion [input] typedString: string (text-edit, max-characters = 60, hidden)
- SuggestCompletion [output]: string[] (single choice, low cardinality)
- Deliver mail [input] address: string (text-edit, max-characters = 60 , label = Address)
- Deliver mail [input] text: string (text-edit, max-characters = 4000, label = Text)
- Deliver mail [output] code: int (table error code-> output message)

The generator creates a first draft of the abstract user interface description from the task model. In our example the result is a partition of the tasks in two sets shown in Figure 4 (blue and red rectangles).

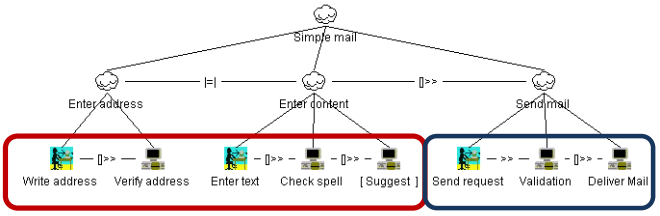

Figure 4: The task model for sending out an email

In this approach, on the one hand the service operation bindings is used to: i)generate the list of the external functions (reference pairs <service URL, operation name>); ii) generate the abstract script for calling the web service (to be "translated" to real code when generating the FUI); iii) generate the handlers for the abstract events (i.e. modify the text of the mail text area with the first suggestion for the typed word). On the other hand, the annotations are used for i) generating user friendly attributes (such as labels) for presentations, groupings and interactors; ii) using the correct interactor types according to the service developer suggestions.

The resulting abstract interface description is shown in Figure 5. It is composed of two presentations (*Write_Mail* and *Result_Presentation*). This is obtained partially automatically. The generated output has been edited in order to move one interactor to the second presentation and have more meaningful interactor names.

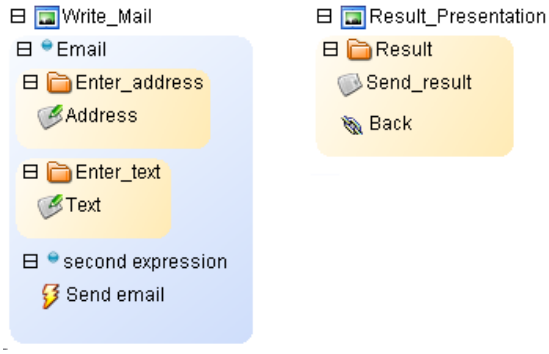

Figure 5: The Abstract Description of the Example.

The next step is the selection of a target concrete platform (for example the graphical desktop platform) and a transformation for creating the corresponding concrete interface. The transformations define (automatically or manually) the concrete interactor that will implement the abstract one (i.e. text field for the address text-edit, a text area for the email text and a button for the send mail activator). The developer can also fine-tune the attributes of the presentations, such as background colour, fonts etc.

The last step is the generation of the final UI selecting an implementation language suitable for the current concrete description. For example, a graphical concrete description can be associated with XHTML or Java.

CONCLUSIONS and FUTURE WORK

We have presented the design of an authoring environment for the development of user interfaces for applications based on Web services and the associated tool support. We have shown how task models can be used to describe how activities should be performed, including those implemented through the Web services. Then, the resulting task model can be used to start the generation of corresponding user interfaces, which can be improved exploiting specific Web services annotations, whose purpose is to provide developers with hints about related aspects. The environment also provides designers with support to easily edit the user interface logical descriptions at various abstraction levels.

A prototype of the design environment has been shown. Future work will be dedicated to improving the rules for transforming task models into user interface logical descriptions, and testing the usability of the authoring environment.

ACKNOWLEDGMENTS

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

REFERENCES

1. Agrawal, A., Amend, M. Das, et al.: Web Services Extension for People (BPEL4People), Version 1.0, June 2007.

2. El bekai A., Rossiter N., "A Tree Based Algebra Framework for XML Data Systems", Proceedings ICEIS 2005, Miami, USA, May 25-28, 2005.

3. Lepreux, S., Vanderdonckt, V., Michotte, B.: Visual Design of User Interfaces by (De)composition. Proceedings DSV-IS 2006, LNCS, Springer 2006, pp.157-170.

4. 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.

5. Luyten, K., Abrams, M., Vanderdonckt, J., & Limbourg, Q. (2004). Developing User Interfaces with XML: Advances on User Interface Description Languages, Advanced Visual Interfaces 2004. Gallipoli.

6. Mori G., Paternò F., Spano L. D.: Exploiting Web Services and Model-Based User Interfaces for Multi-device Access to Home Applications. Kingston, Canada, DSV-IS 2008, Springer Verlag, LNCS 5136, pp.181-193.

7. Oasis standard: Web Service Business Process Execution Language, April 2007.

8. Paternò, . F. , Santoro, C. , Spano, L. D., Model-based Design of Multi-Device Interactive Applications based on Web Services, Proceedings INTERACT'09, Springer Verlag.

9. Paternò F., Santoro C., Mantyjarvi J., Mori G., Sansone S., Authoring Pervasive MultiModal User Interfaces, International Journal of Web Engineering and Technology, Inderscience, 4(2) pp.235-261, 2008.

10. Paternò F., Model-Based Design and Evaluation of Interactive Applications, Springer Verlag, 1999.

11. 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

12. Spillner, J., Braun, I., Schill, A., 2007. Flexible Human Service Interfaces, Proceedings of the 9th ICEIS conference, pp.79-85.

13. Vermeulen J., Vandriessche Y., Clerckx T., Luyten K. and Coninx K., Service-interaction Descriptions: Augmenting Services with User Interface Models, Proceedings EIS 2007, Salamanca, Springer Verlag.