Volume 84, issue 10, October 2011
ISSN 0164-1212
ELSEVIER
The Journal of Systems and Software

The Journal of Systems and Software 84 (2011) 1806–1822

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Engineering the authoring of usable service front ends

Fabio Paternò*, Carmen Santoro, Lucio Davide Spano

ISTI-CNR, Via Moruzzi 1, 56124 Pisa, Italy

ARTICLE INFO

Article history:
Received 1 August 2010
Received in revised form 26 April 2011
Accepted 16 May 2011
Available online 12 June 2011

Keywords:
HCI model-based design
Service front ends
Web Services

ABSTRACT

This paper presents a method and the associated authoring tool for supporting the development of interactive applications able to access multiple Web Services, even from different types of interactive devices. We show how model-based descriptions are useful for this purpose and describe the associated automatic support along with the underlying rules. The proposed environment is able to aid in the design of new interactive applications that access pre-existing Web Services, which may contain annotations supporting the user interface development. This is achieved through the use of task models as a starting point for the design and development of the corresponding implementations. We also provide an example to better illustrate the features of the approach, and report on two evaluations conducted to assess the support tool.

1. Introduction

Service-oriented solutions are becoming more and more widespread, representing an ever more relevant topic in the area of software engineering. They are based on a clear distinction between the front-end and the functionality offered by the service itself, which enables the possibility to reuse services across many interactive applications. Such clear distinction also implies a development model where the application developers and the service developers are different people. However, it can make the creation of the interactive parts more difficult and time consuming since its developers need to understand the Web Service functionalities and how to interact with them. Although the issues associated with this approach are not really new in software engineering, the increasing availability of services on the Web has exacerbated them, especially in user interface software development where it is extremely important to provide usable solutions for access from various contexts of use.

In the business process domain, there are mainstream approaches able to provide support in describing the workflow that drives applications exploiting compositions of such services (see for example BPMN (OMG, 2009)). However, the languages used in such approaches are unsuitable to describe the interactive part of an application. Task models (Diaper and Stanton, 2004; Paternò, 2000) provide descriptions of the activities to be carried out in order to reach users' goals. Such models can offer more useful support for the design of interactive applications, as well as more detailed descriptions of how the interactive activities should be performed compared to workflow notations, which have a more coarse-grained granularity.

Within the area of service-oriented approaches, we focus on Web Services (WS). The design of applications based on them has to be developed by taking into account and exploiting some pre-existing functionalities implemented within such Web Services. The functional interfaces are usually specified through WSDL (Web Services Description Language) files, which are XML-based descriptions indicating the service data types and the available operations, with their input and output parameters. However, 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 User Interface (UI) designers, Web Services can be annotated with user interface hints, whose level of detail can range from simple suggestions to entire UI specifications.

Another approach is to automatically generate user interfaces corresponding to the Web Services through rules that map WSDL descriptions into user interface descriptions (see for example Mori et al., 2008). However, automatic generation produces reasonable results only when the application domain is well-known and, in any case, it is often problematic to compose more or less complete user interface specifications associated with different Web Services to obtain new interactive applications.

A more general approach can be obtained by using Human–Computer Interaction (HCI) models to support the development of multi-platform service front-ends. One advantage of such approach is that it enables deriving user interfaces that adapt to devices with different interaction resources, e.g. mobile or multi-modal devices, without developing ad hoc UIs for each considered device. In particular, we refer to models (Calvary et al., 2002), which allow designers to take the users' viewpoint into

* Corresponding author.
E-mail addresses: Fabio.Paterno@isti.cnr.it (F. Paternò), Carmen.Santoro@isti.cnr.it (C. Santoro), Lucio.Davide.Spano@isti.cnr.it (L.D. Spano).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.05.025

account in the software development, thus obtaining more usable results. Indeed, task models represent the intersection between user interface design and more formal software engineering approaches by providing designers with means to represent and manipulate a formal abstraction of activities that should be performed to reach user goals (Paternò, 2000).

In order to better locate our contribution, we have analysed the type of compositions that can occur in WS-based applications, and identified three main levels at which such composition occurs (all levels are able to invoke services in specific orders through mechanisms that depend on the level considered):

- At the *service* level the services are directly composed one with the other. For instance, this is the situation when the output of a service becomes the input of another service, as it happens when using BPEL (OASIS, 2007).
- At the *application* level the composition is carried out by the application. This means that, for instance, the application can take the output of a service, process it and then provide input to another service.
- At the *user interface* level the input or the output of the services are obtained directly through the user interface without additional processing, as happens in some mash-up applications.

The main contribution of this paper is the description of an authoring environment and the underlying method, which support designers in developing multi-device interactive applications able to address composition of Web Services both at the user interface and at the application level. Therefore, we do not consider the service level (for which there are well-known approaches such as BPEL), rather, we are interested in considering the user interface part of an interactive application, and how the exploitation/composition of Web Services can affect its design. While some preliminary ideas of this work were discussed in Paternò et al. (2009a), a first attempt to support user interface annotations of Web Services in model-based development was introduced in Paternò et al. (2010a), and a discussion on the role of HCI models in Service Front End development is in Paternò et al. (2011), the original content of this paper focuses on presenting the engineering rules, and the associated tool support, for obtaining usable service front-ends generation starting with model-based descriptions of an interactive, service-based application.

After the discussion of related work and the provision of some background information, we introduce the proposed methodological approach for addressing such issues. We illustrate the rules composing the various possible transformations, and describe the tool supporting the method. We also consider an example to better illustrate the features of the approach. Then, we report on two evaluations conducted, indicating the modifications that have been derived from them. Lastly, we draw some conclusions along with indications for future work.

2. Background

In this section we first discuss a number of proposals at research level in areas relevant for our work, then we introduce the model-based languages that are exploited in our work, and we describe the user interface annotations for Web Services that have been considered.

2.1. Related work

In this paper we present a solution to support the development of interactive applications able to compose multiple Web Services and access them, through a variety of different interactive device types. Due to the structured scope of our work, it was difficult to find contributions covering all the aspects that we address, though we have drawn interesting insights from the analysis of related areas.

We can start this analysis by considering some work that focused on the issue of how to automatically generate UIs for accessing WSs (Kassoff et al., 2003; Spillner et al., 2008). However, such approaches do not address the problem of service composition or the adaptation of the resulting interactive application to multiple platforms. More specifically, in Kassoff et al. (2003) the authors aim to obtain a method for easily building browser-based GUIs (Graphical User Interfaces) for Web Services (WSGUI). In their architecture, a WSGUI engine mediates the interaction between the user and the Web Service. The mechanism used is based on Web Services' GUI Deployment Descriptors (GUIDDs), which, together with the WSDL document, will be used by the WSGUI engine to generate an abstract GUI description that will be converted into an HTML page. When the user submits some data (e.g. by filling in a form), the WSGUI engine translates such data into a call to the Web Service, which delivers the result and then the WSGUI engine sends the response to the user as an HTML page. Differently from our solution, the approach proposed offers just one abstraction level that makes handling multi-device issues difficult. Dynvoker (Spillner et al., 2008) is an environment that dynamically generates user interfaces for accessing Web Services. The user interface is created by exploiting both the service definition and a GUIDD file, which contains information about how the internal operation can be made visible to the user. The result of the generation process is an HTML form, which allows the user to invoke the service. As the generation is made at runtime, according to a single service selected by the user, Dynvoker cannot support composed services. Moreover, a designer cannot edit the resulting forms. However, this work is relevant because it shows how it is possible to provide basic user interfaces from the operation and data type description included in a service definition. Some work (e.g. Song and Lee, 2008) has been dedicated to the generation of user interfaces for Web Services but without exploiting model-based approaches.

As we mentioned in the previous section, another relevant aspect in our approach is the problem of *composition,* which can be treated at various levels (service, UI, application). BPMN models are an example in which the composition of services is done only at the service level. Such models can be translated into WS-BPEL (OASIS, 2007) (WS Business Process Execution language), an XML language for describing and executing business processes, which can be used for the composition of Web Services at the business level. However, it does not include information on the user interface for service access. An extension of WS-BPEL is BPEL4People (Agrawal et al., 2007), whose goal is to add into the business process a specification of the user interaction. To this aim, it introduces the concept of people activity. Although this represents a first attempt to take into consideration the issues connected with UI, in this case the interaction is defined always at the business level and therefore a logical description of the user interface is not included in BPEL4People.

Still with regard to obtaining a more "end-user oriented" service *composition*, we found two other attempts to include the user perspective in the design of composite service-based applications. In one case (Obrenovic and Gasevic, 2008), the authors show how spreadsheets can be used for complex service compositions. In particular, they present their end user-oriented framework enabling spreadsheets to send requests and retrieve results from different heterogeneous software services. Their tool supports different composition patterns (e.g. choice, sequence, split), and the authors show how it is possible to facilitate service composition by specifying in a declarative manner the service dependencies. It is worth pointing out that, on the one hand, spreadsheets are a highly efficient and productive tool, and thus this method is very useful for rapid prototyping, especially for not particularly skilled users. On the other hand, it basically uses a text-based interface (for instance, temporal relationships are modelled through text spreadsheet formulas applied to the cells), which could make the specification of complex temporal behaviours rather complicated and unintuitive. In our visual environment a graphical notation is used to express the temporal relationships between the various activities in an intuitive manner. Another user-centred approach for the service composition has been developed in the EzWeb/Fast project (Lizcano et al., 2008). The authors define a platform where developers can create gadgets (also known as widget), which offer a single functionality and a set of events (that contain data inserted by the user through the UI) and slots (inputs needed by a certain widget, i.e. the video url for a video player), used to allow the user to interconnect them. They are stored in a repository and can be composed directly by the users, who can create their own composite application. The limitation of this approach is that all widgets have to be created manually. The composition mechanism is very simple in order to make it understandable for end users, but it cannot express more complex composition cases (for instance, it does not have flow control constructs like if-then-else).

Another interesting work (Daniel et al., 2007) discussed the problem of GUI integration when building composite applications from reusable components. By "GUI integration" the authors mean how to combine components by integrating their front-end/presentation. To this aim, they identify a number of existing UI technologies that might be exploited for future UI compositions. All such solutions basically exploit two kinds of composition languages: *General-purpose programming languages* (very flexible languages, which lack abstraction mechanisms) and *Specialized composition languages* (higher level languages, generally XML-based, which allow composing UI at a more abstract level). Within this framework, our model-based solution supports composition at different abstraction levels, using appropriate XML-based specialized composition languages: therefore it falls more appropriately in the second category.

Still in there search direction covering the issue of building interactive applications by using/composing reusable components, other contributions also address aspects related to *multi-device support*. Indeed, a multi-layer architecture (Pinna-Dery et al., 2003) (one abstract UI and multiple concrete UIs) was proposed for promoting the reuse of UI components (e.g. merging different UI elements), and also adapting them to multiple devices. In order to support composition, a merging language is used, which is based on rules such as union, intersection, etc. The resulting composite UI component becomes a new one, which can be in turn reused for another composition. In their approach, a UI component is specified through a mark-up language (SUNML). While an implementation of the environment for Swing and VoiceXML renderers is already available, the authors admit that the usability of the generated UIs has still to be improved.

Another work dealing with the creation of adaptable user interface using a model composition approach is presented in Raphael et al. (2002). With this methodology user interfaces are built by exploiting the composition of model fragments, stored in a library and appropriately selected in order to fulfil their needs. Each one represents a specific aspect of the GUI. The users specify the view and the control part of each fragment, and they are also able to adapt the UI. For instance, new fragments can be included and the visual rendering of fragment attributes can be modified. However, the proposed approach only deals with graphical user interfaces. Damask (Lin and Landay, 2008) provides support for multi-device user interfaces using patterns and layers, which indicate what should be available in all platforms and what is specific to a given platform, thus facilitating reuse in multi-device applications, but it does not provide any specific support for applications based on Web Services or related user interface annotations.

Finally, since our approach is a model-based one, we deemed it better to analyse the work that has been done so far on the use of HCI models for exploiting WS in interactive applications. What we noticed is that model-based approaches for HCI have not adequately addressed the trend towards exploiting Web Services (which, especially for enterprises offers several advantages in terms of code reusing, productivity and leveraging integration processes). Thus, we have considered existing model-based approaches in HCI able to exploit Web Services from two viewpoints: design and evaluation.

Regarding design, some authors propose (Vermeulen et al., 2007) 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 access to WSDL descriptions without requiring any substantial modifications to generate the corresponding user interfaces, and exploiting logical interface descriptions. Model-driven design and deployment of service-enabled Web applications using WebML have been proposed as well (see for example Manolescu et al., 2005). Our work has a different focus since we propose an environment based on HCI models for generating usable service front ends, which can be implemented in a variety of implementation environments and not only for the Web. In addition, since concrete logical descriptions are often specified through XML user interface languages (Luyten et al., 2004), there have been proposals to use XML tree algebra-based techniques (Lepreux et al., 2006), in order to compose entire or partial concrete presentations. The UI composition techniques are relevant in order to explore the possibility of composing existing UIs for accessing Web Services. For example, the introduction of operations for combining interactors, such as fusion (composition with repetition of the intersection) and union (composition without repetition of the intersection) has been proposed (Lepreux et al., 2006). In that proposal the general XML tree algebra (El Bekai and Rossiter, 2005) is applied to user interface composition and decomposition of graphical user interfaces specified in an XML-based language (UsiXML, Limbourg et al., 2004). However, the method relies only on limited information and does not take into account the possible temporal constraints in the interactions.

Regarding evaluation, the use of models and more formal techniques can also be relevant for enabling early evaluations, which should aim to reduce the need for multiple iterations in the late stages of the software development process (when the modifications to the code are generally more expensive). Indeed, in such cases the use of specifications/models rather than real prototypes (as it happens in user testing) helps in reasoning about some properties of the UI directly in the model. As a consequence, this should allow the designer to discover possible issues early in the development process. The importance of early usability evaluation in model-driven architecture environments is also highlighted in Abrahao and Insfran (2006), which presents a framework incorporating usability as part of a MDA development process. In particular, a usability model for early evaluation is proposed. Using this, the usability of a software system is evaluated and improved at a platform independent level, and also the correspondences between the abstract user interface elements and the final user interface elements in a specific platform are considered. This idea can be supported by our approach, since the various UI specifications at the different abstraction levels are connected each other. In this way, the changes that are made at one level (as a result of a usability evaluation) can be mirrored to the other levels.

2.2. ConcurTaskTrees and MARIA

In this section we introduce the HCI model-based languages exploited in order to design and develop service front-ends.

Task models, such as those described by ConcurTaskTrees (CTT) (Paternò, 2000) or GTA (Van der Veer et al., 1996), can provide useful support. In terms of granularity, tasks can be elementary tasks (namely, tasks that are considered as a logically atomic entity which cannot be further refined) and structured tasks (namely, tasks whose specification consists in a decomposition and a 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 that has already been found and solved, then they can reuse the solution previously developed. The hierarchical structure of this specification has two advantages: it provides a large range of granularity allowing large and small task structures to be reused and it enables reusable task structures to be defined at both a low and a high semantic level.

In our approach the various activities that are supported by an interactive application, together with their temporal relationships are described using the ConcurTaskTrees (CTT) notation (Paternò, 2000), which is widely used in the field of notations for task models, and is even currently considered for standardization within a new W3C group on Model-based User Interfaces (http://www.w3.org/2005/Incubator/model-based-ui/charter/). This notation has a formally defined semantics, which allows the designer to control the flow of the activities supporting the user's goals before designing the user interface, so avoiding the technical details about the system implementation and deployment. The concepts managed by such models are essentially the *task*s, which are activities that can be performed by the *user*, the *system* or by an *interaction* between the two. Each task can be connected with other tasks using a set of temporal operators (*enabling, choice, concurrency, synchronization, disabling, suspend-resume, order independence*), which formally defines their execution order. A task can be refined into sub-tasks, which specify more elementary activities to be performed in order to complete it. Each task is associated with the user interface and the application domain objects it manipulates. The resulting model representation is a hierarchical decomposition of all the tasks and their refinements.

The abstract description of the UI for accessing the service is specified using MARIA (Paternò et al., 2009b), an XML-based language supporting the abstraction layers indicated by the CAMELEON Reference Framework (Calvary et al., 2002). This reference framework is based on the experiences in the HCI (Human–Computer Interaction) model-based community and slightly differs from the OMG framework since the concrete level is dependent on the interaction resources available but independent of the implementation language, while in OMG the platform-dependent level mainly means dependent on the implementation language considered. MARIA supports the CAMELEON framework, with one language for the abstract description (the so-called "Abstract User Interface" level, in which the UI is defined in a platform-independent manner) and multiple platform-dependent languages (which are at the level of the so-called "Concrete User Interface"), which refine the abstract one depending on the interaction resources at hand. Examples of platforms are the graphical desktop, the graphical mobile, the vocal, etc. At the abstract level, a user interface is composed of a number of presentations, has an associated data model, and can access a number of external functions. In turn, each presentation is composed of a number of interactors and there are four types of interactor composition operators: grouping, relation, composite description and repeater. These composition operators support the structuring of the elements inside a presentation. Each presentation is also associated with a dialogue model, which describes how the events generated by the interactors can be handled. With respect to previous languages in this area a number of substantial features have been added (Paternò et al., 2009b), such as a data model, a dialogue model, and the possibility to support typical Web 2.0 interactions.

2.3. User interface annotations for Web Services

Web Services annotations are typically pieces of information that are attached to service definitions, in order to enhance or complement its description. In general, annotations have been provided for many purposes, e.g. to support specification of service operations and parameters semantics based on ontologies. In some cases (e.g. Vermeulen et al., 2007), providing user interface models as annotations has been proposed. In our approach, annotations are considered as suggestions that Web Service developers provide to facilitate the design of service-based interactive applications without prescribing any specific model or implementation of the user interface. In user interface annotations it is important that:

- They can be just hints, so they do not aim at defining the whole user interface.
- They can address various aspects (presentation, content, dynamic behaviour).
- They should be independent of the UI implementation language.
- They can abstract from the platform (if necessary).

They can be attached to each component of the Web Service (the service itself, its data type definitions, its operations with their input and output parameters).

In our work we consider the *ServFace* annotation meta-model (Paternò et al., 2010b) for the definition of UI-related information to be attached to Web Services. This model defines a number of annotations categorised in four groups: *rendering* (how data should be rendered in the UI), *behavioural* (e.g. validation, suggestions, completions), *relational* (properties and associations) and custom (with custom semantics). These annotations can be attached to different service elements such as the whole *service*, *operations*, *parameter groups* (e.g. input and output), *parameter, parameter elements* (for complex type parameters), *data types, data type elements* (for complex data types).

Each annotation can be associated to one or more platforms defining its validity for a given set of devices. The annotation model has also multi-language support.

The *rendering* annotations are:

- *Feedback:* UI elements or element parts that contain information to be communicated to the user, in a text or multimedia format. A feedback element can be a *Label* (human understandable name for a service element), *Activator* (description of an action supported by the service element, i.e. the label of a button), *Help* (an help message), *Error* (an error message), *Warning* (a warning message), *Status* (feedback about the execution of an application functionality), *Multiprompt* (for vocal UIs, a text to be repeated if the user has not understood the first message).
- *Format*: it describes formatting rules for strings fields (e.g. *dd-mm-yyyy* for representing a date).
- *Group*: a set of service elements that should be grouped together (e.g. the surname and the name of a customer).
- *Unit*: it associates a unit to a model element and defines rules for changing the measurement unit (e.g. a service parameter is expressed in meters, with conversion rules for miles).
- *Enum*: static set of possible values for a service element.
- *MIME type*: it describes the multimedia format for the annotated service element.

- *Special Data Type*: additional semantics for a service element (e.g. a given string parameter is a colour, and it is possible to use a different widget for the user input).
- *Rendering property*: presentation-related property of service element such as hidden or perceivable, enabled or disabled, editable or unchangeable, obscured etc.

The *behavioural* annotations are the following:

- *Validation*: rules for checking the validity of input elements.
- *Suggestion*: possible completions for user input, provided as a static list or through a Web Service operation.
- *Form Completion*: automatic completion of an entire form according to one field value.
- *Synchronous update*: update of a presentation part without user intervention.
- *Conditional Rendering Rule*: event–condition–action rules for changing the UI state.
- *Default value*: default value for a service element.
- *Example data*: usage example of a service element.
- *Mandatory field*: to indicate that the specification of a service element is mandatory.

The *relational* annotations are the following:

- *Authentication*: the service element needs authentication in order to be accessed.
- *Semantic data type relation*: equivalence between service elements.
- *Bundle*: specifies a set of operations that should profitably be used together.
- *Operation properties*: semantic properties of operations, such as idempotence.

3. The proposed approach

Since our goal is to support applications based on Web Services (in multi-platform environments), a completely top-down approach going through the various abstraction layers does not seem to be particularly effective. There are many Web Services already available, and their exploitation involves integrating existing third party functionalities, rather than designing a dedicated application completely from scratch, as a top-down approach generally does.

Fig. 1 provides an overall description of the proposed approach, in which five main steps are envisaged. First of all a task model of the interactive application should be designed. It provides an initial draft of the application definition, describing how the various activities are supposed to be carried out within the interactive application (exploiting Web Services operations associated to some application tasks). The task model is described by using a task model language, the ConcurTaskTrees (CTT) notation. The development of a task model is generally carried out by a multidisciplinary team in which various roles/stakeholders are involved, and it is largely driven by user requirements. Therefore, a task model referring to existing services can be useful in order to design how users should carry out the activities in order to achieve effective interaction. Also, the development of a task model (including the precise specification of the various temporal relationships among the different tasks) prompts the multidisciplinary teams to analyse the different application components in depth, in order to identify how they should be effectively combined in a structured design.

The next step in the methodology is to perform an association between the operations specified in the Web Services with some system tasks existing in the task model. The Web Services are described through WSDL specifications, which include description of types, operations, etc. The annotations (associated to the service definition) are described using the *ServFace* annotation model (Paternò et al., 2010b). In addition, as a consequence of this connection, the designer could also refine a bit the task model in order to properly integrate and handle the information associated with the Web Services (and annotations). The output of this phase is a resulting “enriched” task model, which is the input for the next phases and will be used to generate a first draft of abstract descriptions (which can be further edited). When considering different target platforms the abstract descriptions can vary because the tasks supported can vary accordingly. Thus, we can have abstracts specifications that use the same abstract vocabulary but vary in terms of what interactions are actually performed.

Then, the abstract specifications are refined into concrete, platform-dependent descriptions, and in such descriptions the possible vocabularies vary depending on the target platforms.

Lastly, the implementations for the various target devices, which will support access to the Web Services, are generated.

We have also designed a software environment that aims to support the method proposed (it is called MARIAE, the MARIA authoring Environment). It is composed of a number of editors and transformations.

In the next sections we provide further details on our method and the associated tool support, by also considering an example, which is first illustrated.

4. Example application

For the sake of clarity, we consider a simple example, which is an excerpt of a larger application, in which the user is supposed to provide information for performing a DVD search and then watch the selected movie trailer. The associated task model is illustrated in Fig. 2. In this figure the system tasks that are connected with Web Services (*search_film*, *get trailer data*) are identified by the icon used for system tasks modified by the addition of a small circular shape. Here we analyse a set of tasks for searching, selecting and watching the trailer of a DVD.

The system has to provide two functionalities: the DVD search and the trailer delivery. These functionalities can be implemented by two operations in two 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 different services that could be offered by different providers and are not designed to work together. The services are:

- The Amazon *AWSECommerce* service, which exposes operation and data types for performing a search into the Amazon.com DVD catalogue. This search can be performed specifying a so-called *ItemSearchRequest* (containing the information about the requested keywords, title, author and director) and sending it to the Amazon server through the *ItemSearch* operation. It returns a list of matching items representing the search results.
- The You tube search service, which exposes the operation for performing a search in its video repository. The keyword for performing the search can be specified, which will be the title of the selected DVD concatenated with the “trailer” keyword. The first item matching the search will be presented to the user. This service is exploited through an ad hoc SOAP wrapper. It contains a *SearchVideo* operation, which receives as input a *Query* object (containing the keywords, the maximum number of result expected, etc.) and returns the list of matching videos. The query is filled in order to return only one video.

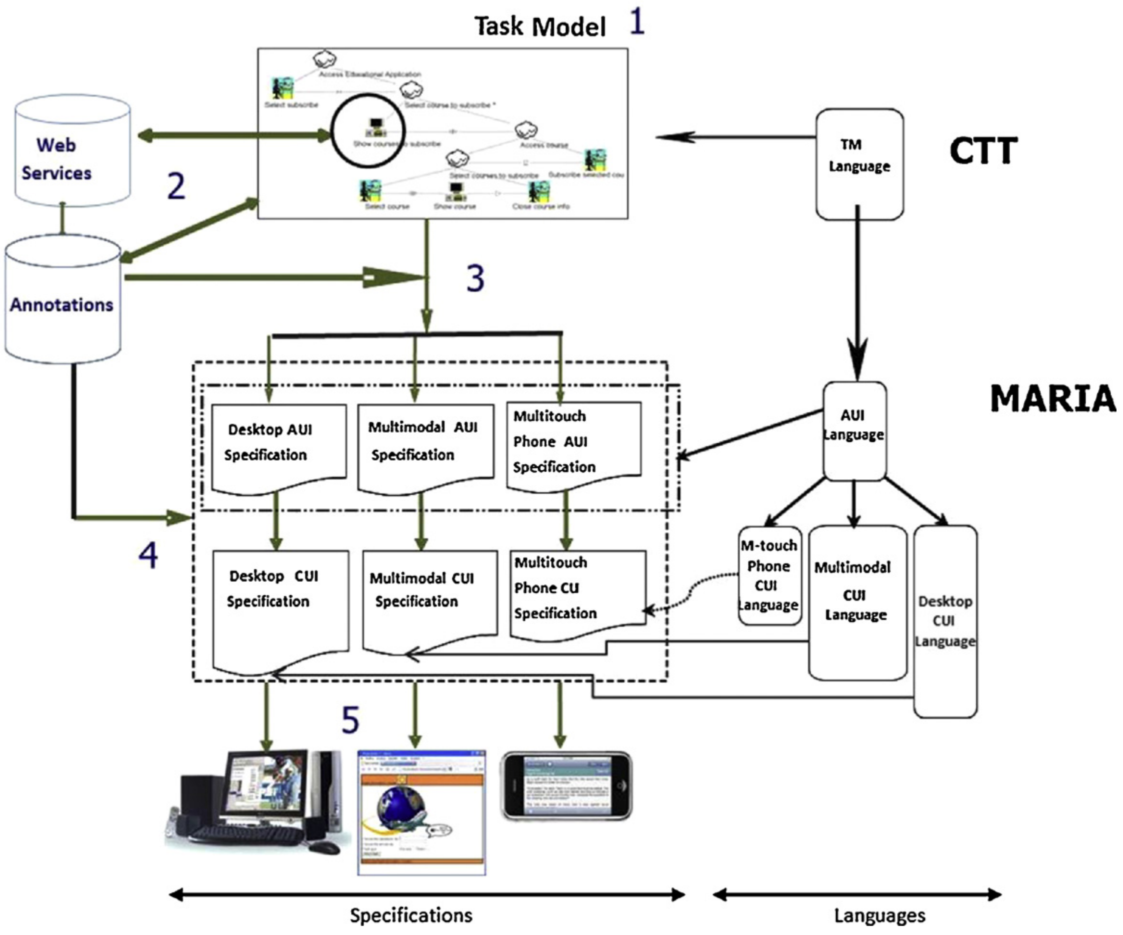

Task Model
1
Web Services
2
Annotations
3
TM Language
CTT
MARIA
AUI Language
Desktop AUI Specification
Multimodal AUI Specification
Multitouch Phone AUI Specification
4
Desktop CUI Specification
Multimodal CUI Specification
Multitouch Phone CU Specification
M-touch Phone CUI Language
Multimodal CUI Language
Desktop CUI Language
5
Specifications
Languages

Fig. 1. The approach proposed.

5. Binding task models with services and their annotations

5.1. Method

This step takes as input the task model (describing the activities to support in the interactive application), a set of Web Services (which have to be included in the design since they provide basic functionalities), and also possible annotations associated with such Web Services (which provide hints for the generation of the corresponding UI).

On the one hand, since the task model expresses how the interactive application assumes that the activities will be carried out, it also indicates the *system* functionalities to support. On the other hand, the Web Services are expected to describe such system functionalities (using a different specification). Therefore, Web Services and application tasks can be seen as two ways of representing application functionalities. However, differently from Web Services, which are provided as an unstructured list (no particular relationship is specified between Web Services operations), the tasks belonging to the task model (and then, also the application tasks) are connected each other through the hierarchical structure of the model and its temporal operators. Consequently, if some elementary (system) tasks are associated with relevant Web Services, we can provide useful indications about how the Web Services should be composed thanks to the structuring inherited from the task model. This composition is carried out by binding together system tasks and Web Services operations.

Some rules can be followed in order to perform task/service associations in a consistent way. As said, since Web Services are application functionalities, they are associated with *system* tasks. In particular, it is important to endow the task model with 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 subtask will be associated with an operation defined in the Web Service. Thus, if a Web Service supports three operations, then there can be three basic system tasks representing their execution. In general, it is not

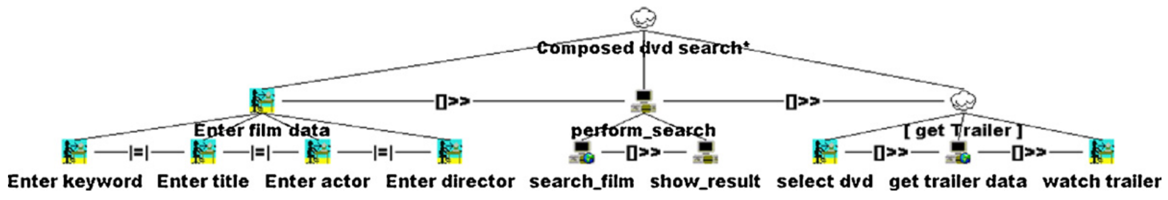


Fig. 2. The task model for the DVD access.

necessary to consider all the operations of a Web Service, but it is sufficient to include in the task model only the ones needed to support the activities associated with the application that has to be designed.

Once such associations are performed, it is necessary to specify how the input for the considered system operation is provided. Designers can indicate this by either connecting *interaction tasks that already exist in the task model* (and which can enable the execution of the Web Service operation, represented as a *system* task) with concerned system tasks, or adding some *new interaction* tasks to the model in order to build a more detailed description of the task model. Such model enhancements or refinements have to be carried out to ensure that the parameters needed by a certain operation have been correctly collected before the operation is invoked.

The typical access to a Web Service operation can be modelled in the task model in the following way: interactive tasks for providing input, one system task for the operation execution, and one task for presenting the results. Then, some specific information is derived from the associations between the Web Services operations and the *system* tasks, which can be useful to complete the abstract description:

- The abstract interactors corresponding to interactive tasks should contain a data model entity for storing the value entered by the user and then passed to the Web Service operation, whose type is derived from the WSDL.
- We need to include an *activator* interactor for modelling the triggering of the access to the Web Service.
- The interactor that presents the result of the Web Service execution in the UI should contain a data model entity, whose type is still derived from the WSDL.

Therefore, the goal of this step of the methodology is to obtain a complete and consistent task model (also through performing some refinements and/or modifications to it) in such a way that the information included in the WS can be properly included and integrated. This phase can be summarised in the following main points:

1. Binding the service operations with the corresponding basic *system* tasks.
2. By applying the CTT semantics, calculate the set of possible *interaction* tasks that have to be accomplished to enable each *system* task associated with a Web Service operation. This can be done by following the list of parent nodes in the CTT model starting with the considered *system* tasks and, checking whether the current node is the right operand of an *enabling* expression. The set will contain all the *interaction* tasks contained in its left operand.
3. The designer specifies which *interaction* task provides the data for each parameter of the operation. If more than one parameter corresponds to the same task, it can be refined into two or more subtasks, one for each parameter considered.
4. The designer specifies which tasks receive the output of the operation (if any). The set of potential receivers is calculated applying the CTT semantics in order to discover which tasks are enabled by the execution of the system tasks bound to the operation.
5. If the annotations provide information about possible dependencies between the Web Service operations (such as auto-completion or validation functionalities), it is also possible to refine the model (either automatically or manually) by adding the temporal relations that express such dependencies. For instance, in the case of an auto-completion dependency specified in the annotation model for the input of an operation, the interactive task that provides such input can be refined through the iteration of a sequence of tasks for entering data, invoking the suggestion functionality, displaying the list of produced results and selecting one value from such a suggestion list.

5.2. Tool support

It is worth pointing out that the MARIAE tool supports some analysis of task models. For this purpose, it includes a task model simulator (see Fig. 3), which allows interactive execution of the model. At the beginning, the tasks initially enabled by the temporal relations defined in the model are shown: when the designer selects one of them, the tool will show the next tasks enabled after the performance of the selected task. This can be useful for checking how the activities described in the task model dynamically evolve according to the designers' choices.

The *Tasks-Services Binding Editor* supports the associations between the tasks included in the model, describing how to perform activities in the application to be developed, and the Web Services that the designer wants to exploit in the interactive application. 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. The tool automatically takes the WSDL of the Web Service and shows the corresponding information on the right panel (see Fig. 4): the list of operations and the associated input and output parameters.

The main part contains the CTT model using a hierarchical tree representation: the children of a task are the decomposition of the task itself. The tasks at the same level are connected using various temporal operators. As introduced before, there are four categories of tasks depending on how their performance is allocated. Each task allocation is indicated through the use of a specific icon. *System* tasks (represented by computer icons) can be bound to a Web Service operation indicated on the right part, where different services with their operations and data types are listed. After this operation selection, the tool is able to automatically calculate all the possible *interaction* tasks that can enable the execution of the specified Web Service. The designer is then able to associate a single *interaction* task to each parameter of the operation. If the same *interaction* task is selected for two or more input parameters, it is automatically refined into the composition of two or more sub-tasks connected through an order independence operator. These tasks can also be further edited by the designer after wards in order to ensure that the tool is always able to generate an appropriate UI for the operation invocation.

The right part of the tool user interface includes the Annotation Browser, where the designer can load and visualize annotations that enhance the WSDL definition. The set of annotation models loaded is exploited during the AUI/CUI generation.

During the binding phase the tool aims to gather as much information as possible, which can then be useful for the generation of the corresponding abstract user interface. Thus, when the developer binds a system task and a Web Service operation (in the example in Fig. 5 the *Check Login Data* system task has been associated with the *login* Web Service operation), it looks at the input and output parameters of the Web Service and tries to identify what interactive task can provide the input parameters and what tasks can be associated with the output parameters. In this search it considers that (i) the interactive tasks associated with the input parameters should be enabled just before the occurrence of the task corresponding to the Web Service execution and (ii) the tasks providing the output parameter should be performed immediately afterwards. The task presenting the result of the Web Service can be either an interactive task (in this case the user is able to manipulate it) or a system task (in this case the user interface should only display it, without allowing further editing).

Thus, in order to facilitate the designers' work, the tool shows (see Fig. 5) the available options using pull-down lists (when there

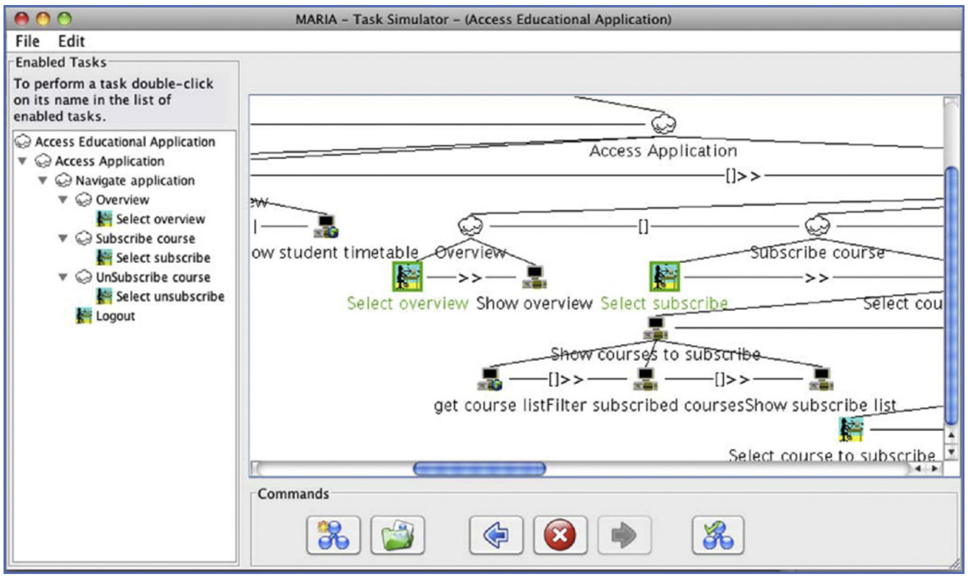


Fig. 3. The task simulator included in MARIAE.

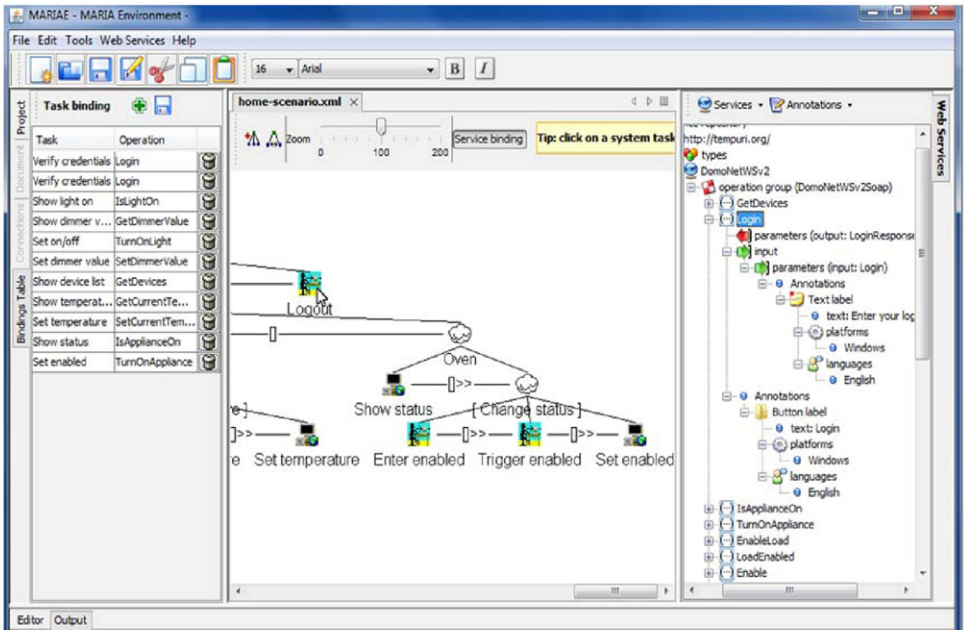
are multiple choices). In addition, if the user indicates that the same task is used for providing multiple input parameters for the Web Service, the tool automatically refines such task into as many subtasks as input parameters, connected through an order independence temporal operator. In this way, we obtain a more refined task model, which facilitates the creation of the corresponding abstract user interface that will include one interaction element for each basic interaction task. The mappings between basic tasks and operation parameters are useful in the generation of the abstract user interface because the parameters have XSD (XML Schema Definition) types that are defined by the WSDL, and thus indicate what type of data objects the corresponding abstract interactors should manipulate.

The CTT model enhanced with the Web Service binding and annotations (which are respectively displayed on the left part and on the right part of the tool, see Fig. 5), is used to generate the first AUI (Abstract User Interface) model for the interactive application, which can be modified by the designer using the AUI/CUI (Concrete User Interface) editor.

5.3. Example application

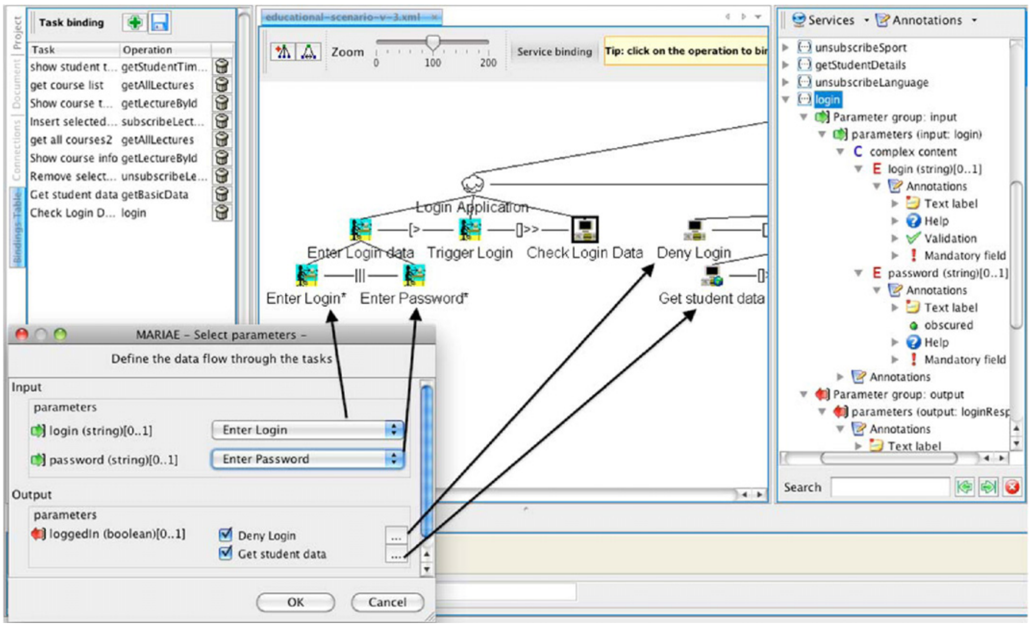
In the example considered, we suppose the following information coming from both the WSDL and the annotations for the operations:

- ItemSearch [input] *keywords*
  - *Type:* string, max length: 80 characters
  - *Text label*: "Keyword"
- ItemSearch [input] *actors*
  - *Type:* string, max length: 80 characters
  - *Text label*: "Actor"



Task	Operation
Verify credentials	Login
Verify credentials	Login
Show light on	IsLightOn
Show dimmer v…	GetDimmerValue
Set on/off	TurnOnLight
Set dimmer value	SetDimmerValue
Show device list	GetDevices
Show temperat…	GetCurrentTe…
Set temperature	SetCurrentTem…
Show status	IsApplianceOn
Set enabled	TurnOnAppliance

Fig. 4. The tasks-services binding editor.



Task binding

Task	Operation
show student t...	getStudentTim...
get course list	getAllLectures
Show course t...	getLectureById
Insert selected...	subscribeLect...
get all courses2	getAllLectures
Show course info	getLectureById
Remove select...	unsubscribeLe...
Get student data	getBasicData
Check Login D...	login

MARIAE – Select parameters –

Define the data flow through the tasks

Input

parameters

login (string)[0..1] Enter Login

password (string)[0..1] Enter Password

Output

parameters

loggedIn (boolean)[0..1] [✓ Deny Login] [✓ Get student data]

Fig. 5. Interactive association of basic tasks and operation parameters.

- ItemSearch [input] *title*
  - *Type:* string, max length: 80 characters
  - *Text label*: "Title"
- ItemSearch [input] *director*
  - *Type:* string, max length: 80 characters
  - *Text label*: "Director"
- ItemSearch [output] *smallImage*:
  - *Type:* byte array
  - *MIME type:* image-jpeg
  - *Text label:* "DVD cover"
- ItemSearch [output] *Title*
  - Type: string
  - *Group:* "Result_entry"
- ItemSearch [output] *Actors*
  - Type: string
  - *Group:* "Result_entry"
- ItemSearch [output] *Rank*
  - *Type:* float with min value = 0 and max value = 5
  - *Group:* "Result_entry"
  - Special data type: rating
- SearchVideo[input] *movie_title*
  - Type: string
  - *Semantic data type relation*: equals to Title output parameter of the ItemSearch operation
- SearchVideo[output] *URL*
  - *Type:* byte array
  - *Group:* "Result_entry"
  - MIME type: video-flv

In such example the service composer binds the system tasks to the Web Services operations using the Web Service pane of the editor, specifying the connections between the system tasks and the WS operations. In particular, the *ItemSearchRequest* is bound to the *search_film* task, while the *SearchVideo* operation is bound to the *get_trailer_data* task. Moreover, the composer also selects the tasks that will provide the input parameters to each WS operation and the ones that will exploit their outputs. In order to clarify the association between the input parameter of the operations and the interactive tasks, the *ItemSearch* operation has a single parameter (the *ItemSearchRequest*), but it consists of many elements that can be specified (or not) for the service invocation. The editor is able to show the complex type structure and to specify that a certain *interaction* task provides the content for a particular element in this complex type. For instance, the designer can specify that one or more fields among the *keyword*, *actor*, *title* and *director* should be specified in order to send the DVD request.

The authoring environment (see Fig. 6) supports such specifications through the Web Service browser, which allows the developer to specify the URL of the WSDL file for inspecting operations (with input and output parameters) and data types defined for invoking the service. Then, it is possible to load annotations for the selected Web Service (if any) for supporting the logical user interface generation process.

6. From enriched task model to abstract user interface

6.1. Method

Since the task model has been designed together with the end users, it considers the user perspective in identifying the possible solutions, and the resulting user interface specifications (the abstract and the concrete representation) should preserve such perspective. The procedure for generating the abstract description starts with the task model and derives a first partition of all the leaf tasks consisting of a number of sets called *Presentation Task Sets* (PTSs). In particular, PTS collection is obtained following the CTT temporal relation semantics. Each PTS contains a set of basic tasks, which should be supported at the same time according to the temporal relations in the task model. Therefore, PTSs are good candidates to identify the abstract UI presentations, which are associated with the sets of user interface elements presented at a given time (in the case of Web applications a presentation is an abstraction of a Web page).

The algorithm for this transformation (from task model to PTSs) constructs a binary tree representation of the task model. Each tree node represents an elementary task or a composite task expression (composed of a left operand, a temporal operator, and a right operand). The leaves of the task model are the leaves of the binary tree. The PTSs are calculated by visiting this tree, and maintaining

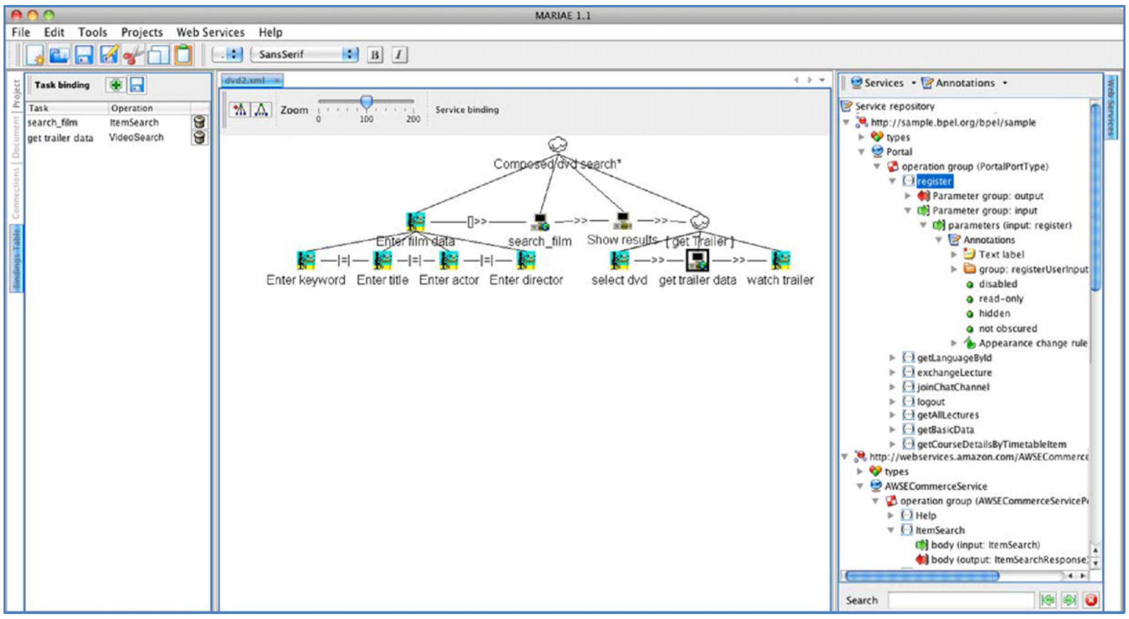


Fig. 6. The example in the authoring environment.

a *current_PTS* variable, in which tasks are added if there is no need to create a new PTS. This variable is initialized with the tasks that are enabled at the application start-up. The algorithm proceeds as follows, it performs a depth-first visit of the tree and:

- If the node is a leaf then it is added to the *current_PTS*.
- If the node is an *enabling* expression, the PTS for the left operand are recursively calculated starting with the *current_*PTS, while the PTS for the right operand is calculated starting from a new PTS. In this way the resulting PTSs for the two operands are different. The *enabling* is reflected in the UI structure, by generating a connection between these two PTSs.
- If the node is a *suspend-resume* or a *disabling* expression, the algorithm performs a separate calculation for each operand. For the left operand it continues with the *current_*PTS. After that, the tasks in the right operand that are enabled when the user starts to execute the expression are added to all the PTSs coming from the left operand, in order to let the user *suspend* or *disable* the left operand expression. The other remaining tasks are recursively analysed.
- If the node is a *concurrency* expression, all the tasks that are enabled when the user starts to execute one of the tasks belonging to such expression are added to the *current* PTS. After that the calculation continues recursively on the left and the right operands.
- If the node is a *choice* or *an order independence* expression, the *current* PTS does not change and the two operands are recursively analysed.

Such procedure can include the same task in more than one PTS: the rationale is that the same task may be supported in different points during the application execution. In addition, the first PTS identification step can produce sets having a very low cardinality. Since in the end each PTS will correspond to a dedicated presentation, we should avoid having presentations supporting a very limited number of user interface elements. To this aim, within the tool four different types of heuristics have been provided to the designer in order to join PTSs together and obtain more compact results (which can also be changed during later modelling steps):

- If two (or more) PTSs differ by only one element and their elements are at the same level connected by an enabling operator, they can be joined together.
- If a PTS is composed of just one element, it can be included within another superset that contains such element.
- If two (or more) PTSs share most elements, they can be unified in order not to duplicate elements that are already available in another presentation.
- If there is an exchange of information between two tasks, they can be put in the same PTS in order to highlight such relation.

In the process of merging PTSs the designer can take into account the target platform: in the case of desktop systems more merging can be useful with respect to mobile versions. After the calculation of the abstract UI presentations, the corresponding interactors and interface behaviour have to be generated. For this purpose, three types of rules are applied to the task model description:

- Temporal relations among tasks indicate requirements for the UI dialogue model because user actions should be enabled in such a way as to follow the logical flow of the activities to perform.
- Task hierarchy provides information regarding grouping of UI elements: if one task is decomposed into subtasks, it is expected that the interactions associated with the subtasks are logically connected, and this should be made perceivable to the user.
- The type of task provides useful information to identify the most suitable interaction technique for the type of activity to perform (e.g. control tasks are represented through activators, selection tasks through selection interactors, etc.).

The binding between system tasks and Web Service operations is exploited in order to generate the logic that enables the invocation of the operation. Indeed, if the system task is enabled by one or more control tasks, the generator adds an event-handler to the corresponding activators. If this activator does not exist, it is automatically created and added to the presentation. The handler collects the input values from the interactors (more precisely those that correspond to the tasks specified by the designer as input providers for the operation), invokes the external function corresponding to the bound operation and updates the interactors that are intended to receive the output (still specified by the designer at the binding step).

The interactor corresponding to the input/output parameters of the operation are generated according to the service data type definitions, finding the most appropriate interactor that supports the editing or the visualization of values (e.g. *TextEdit* interactor for editing strings, *NumericalEditFull* for editing integers, etc.). If the data type is a complex one, its definition is recursively explored, creating groupings corresponding to the inner elements, until the simple ones are reached. For arrays and list the generator creates *Repeater*s for output (the repeater is a composition operator that enables the presentation of a dynamic homogeneous list defining a template for the generic element), while for the input creates the interactor for adding and removing elements.

In addition, it is also possible to take into account the information contained into the annotation model during the generation. More precisely, the following annotations have an impact on the AUI:

- *Format*: the generator procedure adds the logic for verifying the format.
- *Group*: the generator adds a grouping that contains the interactors that corresponds to the annotation-specified service elements, preserving the order.
- *Units*: the generator adds a selection interactor for enabling the measure change and generates the logic for performing the conversion on selection change.
- *Enum*: if the service element is an input, the generator creates a *selection* interactor, with the *Enum* elements as choices.
- *Rendering properties*: the information specified is reflected into the interactor rendering (e.g. if hidden is specified, the interactor flag *hidden* is set to true).
- *Validation*: the generator creates the logic for performing the validation of the input fields, according to the annotation specification.
- *Form Completion*: the generator creates the logic for performing the completion.
- *Synchronous update*: the generator sets the *continuous_update* flag to true and the *continuous_update_function* attribute to the external function corresponding to the annotation-specified update operation.
- *Default value*: the input field value is set to its default value.
- *Mandatory field*: the generator creates the logic for checking that the user enters a value for the corresponding input interactor.
- *Semantic data type relation*: the information is exploited in order to improve the usability of the generated UI avoiding multiple inputs of equivalent fields etc.

In general, in the abstract user interface description it is possible to compose UI elements for identifying groups of logically connected elements and relations among groups of elements. Compositions and elements are included in presentations and determine their structure. In addition, entire presentations can be composed through connections: a connection included in a certain presentation specifies the UI element that triggers the navigation, together with the corresponding target presentation.

6.2. Tool support

Once the task-Web Service association has been carried out, the tool produces a first draft of the corresponding Abstract User Interface (AUI) description. In this process it is possible to exploit some annotations associated with the Web Services, which provide further information about how the functionality included in the Web Service should be finally rendered in the UI.

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

Fig. 7 shows the interface for editing a concrete description (the editing of abstract descriptions is similar): the left part contains an interactive tree diagram of the presentations describing the user interface and the corresponding elements. In the central part of the editor interface there is a panel for editing the model, where each interactor composition is a container for various interactor representations. All the elements of the model are classified according to their semantics taking into account the target platform. For instance, a choice with low cardinality in a desktop CUI will be represented as a radio button, showing all the possible choices with the default option selected. All the interactor representations can be dragged from one interactor composition to another, depending on their semantics. In the central part of the window it is possible to work either through a graphical representation of the model or the corresponding XML file. Changes in one representation are automatically updated in the other.

The right part of the editor interface is a toolbox from which new instances of interactors can be added to the current model (depending on the currently selected element within the model, it shows only the elements that can be included within it). In addition by selecting the appropriate tabs in the right part of the editor, it is also possible to specify the values of the selected element properties, attributes and events.

6.3. Example application

As we already discussed, the first transformation creates a draft of the AUI by calculating the Presentation Task Sets, which are the sets of tasks enabled over the same period of time according to the temporal relations defined in the task model.

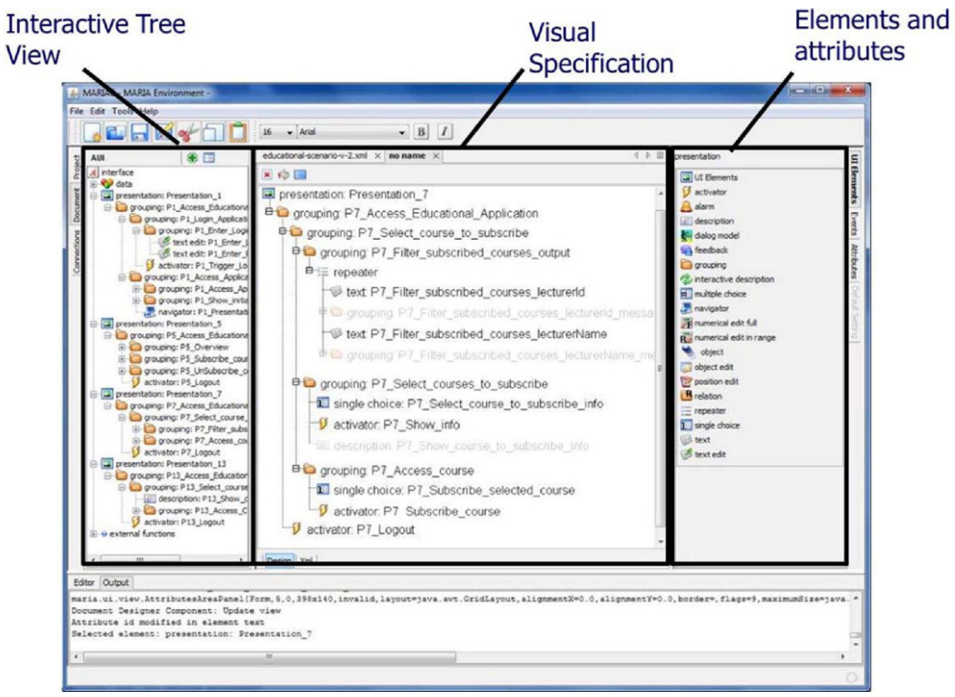
In our example the result is a partition of the tasks into three presentation task sets, which are identified automatically. They can be used to identify corresponding presentations in the user interface when we transform the task model into an abstract user interface description.

In this approach, on the one hand the service operation bindings will be used to:

1. Generate the list of the external functions (reference pairs <service URL, operation name>).
2. Generate the abstract script for calling the Web Service (to be "translated" into real code when building the final implementation).
3. 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 will be used for (i) generating user friendly attributes (such as labels) for presentations, groupings and interactors; (ii) identifying the correct interactor types according to the service developer suggestions.

The resulting generated abstract user interface presentations are shown in Fig. 8. The first generation result is the typical page splitting for a search application: the first page shows only the interface for entering the search criteria, the second one shows the search results and the third allows the user to watch the video. However, our application is not aimed at presenting a large number of videos to the user, rather it should provide help for selecting


Interactive Tree View
Visual Specification
Elements and attributes
MARIA - MARIA Environment -
File Edit Tools Help
16 Arial B I
AUI
interface
data
presentation: Presentation_1
grouping: P1_Access_Educationa
grouping: P1_Login_Applicati
grouping: P1_Enter_Logi
text edit: P1_Enter_
text edit: P1_Enter_
activator: P1_Trigger_Lo
grouping: P1_Access_Applica
grouping: P1_Access_Ap
grouping: P1_Show_initia
navigator: P1_Presentati
presentation: Presentation_5
grouping: P5_Access_Educationa
grouping: P5_Overview
grouping: P5_Subscribe_cour
grouping: P5_UnSubscribe_c
activator: P5_Logout
presentation: Presentation_7
grouping: P7_Access_Educationa
grouping: P7_Select_course_
grouping: P7_Filter_subs
grouping: P7_Access_co
activator: P7_Logout
presentation: Presentation_13
grouping: P13_Access_Education
grouping: P13_Select_course
description: P13_Show_c
grouping: P13_Access_C
activator: P13_Logout
external functions
Project
Document
Connections
educational-scenario-v-2.xml
no name
presentation: Presentation_7
grouping: P7_Access_Educational_Application
grouping: P7_Select_course_to_subscribe
grouping: P7_Filter_subscribed_courses_output
repeater
text: P7_Filter_subscribed_courses_lecturerId
grouping: P7_Filter_subscribed_courses_lecturerId_messa
text: P7_Filter_subscribed_courses_lecturerName
grouping: P7_Filter_subscribed_courses_lecturerName_m
grouping: P7_Select_courses_to_subscribe
single choice: P7_Select_course_to_subscribe_info
activator: P7_Show_info
description: P7_Show_course_to_subscribe_info
grouping: P7_Access_course
single choice: P7_Subscribe_selected_course
activator: P7_Subscribe_course
activator: P7_Logout
Design
Xml
presentation
UI Elements
activator
alarm
description
dialog model
feedback
grouping
interactive description
multiple choice
navigator
numerical edit full
numerical edit in range
object
object edit
position edit
relation
repeater
single choice
text
text edit
UI Elements
Events
Attributes
Editor
Output

```
maria.ui.view.AttributesAreaPanel[Form,6,0,398x140,invalid,layout=java.awt.GridLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=9,maximumSize=java.
Document Designer Component: Update view
Attribute id modified in element text
Selected element: presentation: Presentation_7
```

Fig. 7. The editor of the user interface models.

a DVD showing the movie trailer. Such presentation structure is already suitable for a mobile device. However, the developer can decide to merge the three presentations into one (and then dedicate one part for entering keywords, one part for showing the results and the last one for watching the trailer) in case of a desktop platform that has a large screen (see Fig. 9).

The first generation step exploits the annotations in order to select the interactor types to be included in the UI, while the task model structure is used in order to compute the groups and presentations for the generated abstract UI model.

In particular, at the abstract level, the annotations affect the interactor generation in the following ways:

- The *Repeater* interactor composition is generated because the search service returns an array of results. The semantic of this model element is that its content is repeated for each element contained into the specified list (in this example, it is the result list).
- The *Group* annotation is used to group together the elements into the *Repeater*.
- The *Semantic data type relation* of equivalence between the title output parameter of the *ItemSearch* operation and the *movie_title* input parameter of the *SearchVideo* operation, enables the generator to implement the selection of one result directly into the rendering of the list through a dedicated activator for each list entry, avoiding the replication of the list inside a *single_choice* interactor (which is usually generated if the annotation information is not available).
- The *Special data type* annotation is used to generate an image rendering of the rating, using a star for each rating point.

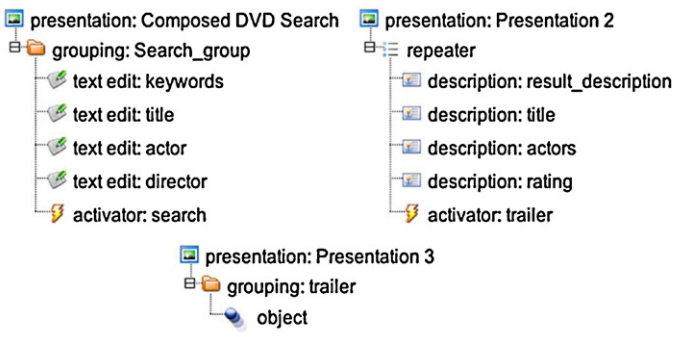


Fig. 8. Generated abstract UI presentations.

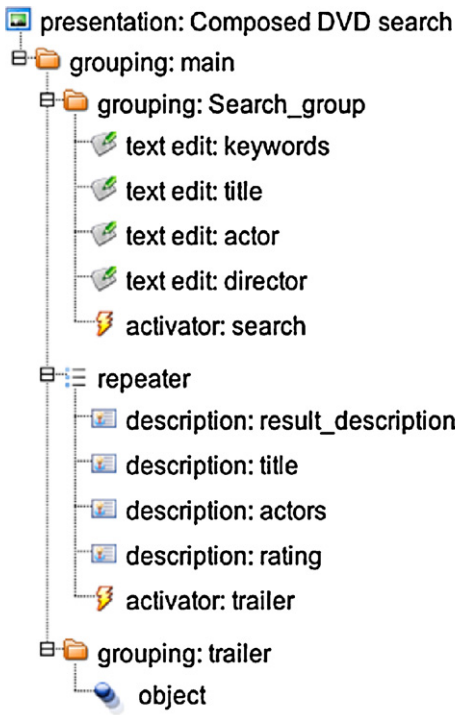


Fig. 9. Modified AUI.

7. From abstract user interface to concrete user interface

7.1. Method

Once the Abstract User Interface has been obtained, the next step consists of deriving the Concrete User Interface description. Since we use multiple transformations, the associations between tasks and Web Services (which implicitly also define the connection with the annotations), and between tasks and corresponding UI elements created during the abstract user interface generation are stored in order to be reused during the concrete user interface generation.

More precisely, the transformation refines the abstract user interface choosing a concrete implementation for each abstract interactor (e.g. a text link for a navigator etc.).

The data type of a service element correlated with an interactor facilitates the selection of an appropriate concrete refinement. For instance, for both a *boolean* data type and an *enumeration* data type having ten elements the abstract interaction will be the *single_choice*. At the concrete level for the first one a *radio_button* will be selected (due to the low cardinality of the two selectable values), while for the second one the refinement will be a *drop_down_list*.

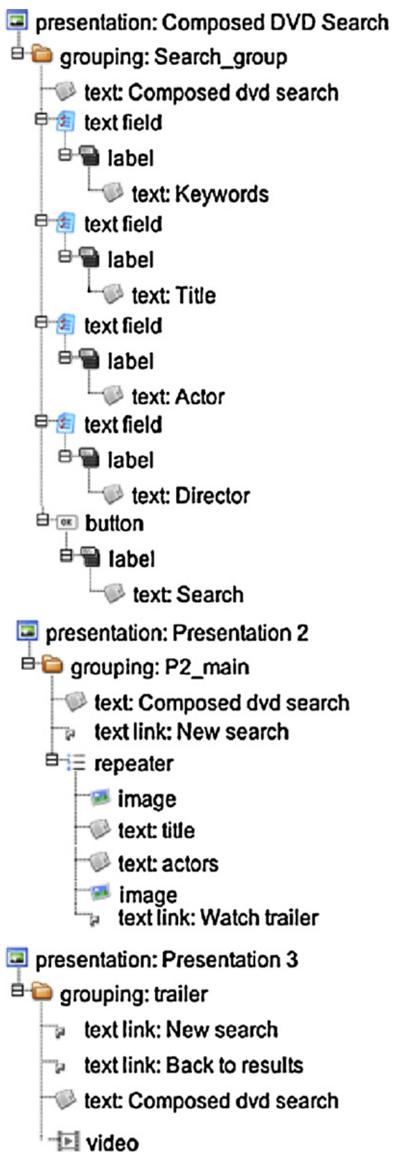
Moreover, the annotations have the following effects on the concrete refinement:

- *Feedback*. The information about *labels*, *help, error, warning* and *status* messages are included into the concrete refinement. The label is directly visible, the help information is available on user request, while warnings, error and status messages are visible only according to validation or execution status.
- *MIME Type*. It is exploited in order to generate the correct interactor (e.g. an image, a video etc.).
- *Special Data Type*. It is exploited in order to generate a dedicated widget for entering some special values such as dates (calendar), colours (colour picker) etc.
- *Rendering Property*. Used to set the correspondent interactor rendering attributes (e.g. obfuscate a password field).
- *Suggestion*. It is exploited to fill the suggestion drop down list for a text field.
- *Conditional Rendering Rules*. The effects specified into these event–condition–action rules are reflected into the corresponding event handlers associated to UI elements.

The annotations considered at the abstract level can also be reused at the concrete level, because it is possible that they contain information useful for the following steps (e.g. the interactor type can be selected at the abstract level, but its label can be different for different platforms).

7.2. Example application

The next generation step is the selection of a concrete platform (for example the graphical desktop platform) and a transformation for creating the corresponding Concrete User Interface. In this step the corresponding transformation selects the concrete interactor that will refine 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). Then, the developer can also fine-tune attributes for the presentations, such as background colour, fonts etc. If we consider the AUIs shown in Figs. 8 and 9, the generation result for the mobile platform is shown in Fig. 10, while the result obtained for the desktop platform is visualized in Fig. 11. As you can see from the latter two figures, the two concrete models are different from a structural point of view. The mobile one has smaller presentations and contains links in order to support navigation. Nevertheless, the contents look very similar because, though they belong to different meta-models, the vocabulary for graphic controls like buttons, images etc. is shared. This is one of the most powerful advantages of the MDA approach: the concepts that are similar remain similar across platforms, even if the final implementation uses very different technologies (e.g. the desktop CUI can be implemented by using Java Swing, while the mobile one by using Objective C).


```
presentation: Composed DVD Search
  grouping: Search_group
    text: Composed dvd search
    text field
      label
        text: Keywords
    text field
      label
        text: Title
    text field
      label
        text: Actor
    text field
      label
        text: Director
    button
      label
        text: Search
presentation: Presentation 2
  grouping: P2_main
    text: Composed dvd search
    text link: New search
    repeater
      image
      text: title
      text: actors
      image
      text link: Watch trailer
presentation: Presentation 3
  grouping: trailer
    text link: New search
    text link: Back to results
    text: Composed dvd search
    video
```
Fig. 10. CUI for the mobile platform.

The annotations are exploited again during the CUI generation on both platforms, according to their capabilities. In particular the information exploited in this example for refining the interactors is the following:

- The *Label* annotations are used to generate the interactor label elements.
- The *MIME type* annotations are used to interpret the byte array as an image (for the *result_description* interactor) or as a video (the object contained into the trailer grouping).
- The length of the input strings is used to generate the columns of the text fields.
- The *Special data type* annotation is used to generate the five-star notation for the rating.

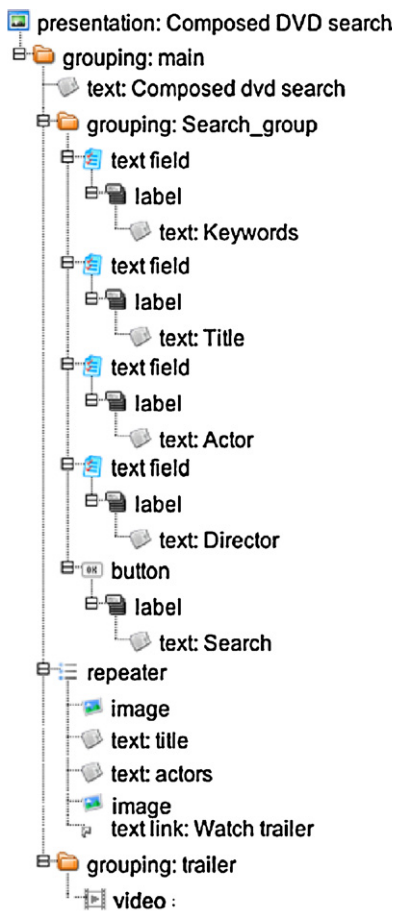


Fig. 11. CUI for the desktop platform.

8. UI generation

8.1. Method

The last step is the generation of the Final UI, through a model-to-code transformation of the CUI.

To this regard, the MARIAE tool, which is able to support the whole design process, also incorporates a set of usability guidelines. As an example of UI guidelines we can consider those described in Bastien and Scapin (1993). For instance, an example guideline for prompting criterion is the following for data entry: the user should be provided with the required formats and acceptable values, possibly providing them with additional cueing of data format (e.g. “Date (mm/dd/yy)”). In our approach, this is supported by analysing the operations and the data types associated with the input and output parameters of the Web Services considered, or through an explicit annotation, therefore enabling automatic mechanisms that control the user’s input. Indeed, knowing the data types not only enables techniques for checking input validity (which should improve the usability of the application), but also allows for inferring the appropriate UI abstract interaction object. For instance, if we have a date value, a possible UI object for effectively render it on the UI could be a calendar. This kind of knowledge is embedded in the UI transformations supported by the tool and is aimed at improving the usability of the final UI result.

The inclusion of such usability guidelines in the MARIAE environment has the advantage of supporting the provision of prototypes for which less evaluation iterations are needed, since the user interfaces that are generated through such tool have already taken into account some usability-related aspects from the early phases of their design and development. In addition, the possibility for the designer to manipulate user interfaces that are described in more abstract terms than the implementation level, together with the use of graphical and intuitive representations and UI previews provided by the tool, should help the designers to focus on semantic aspects and put them in a position to better evaluate the quality (in terms of e.g. usability) of the first draft of the user interfaces generated.

8.2. Example application

The last step is the generation from the CUI of the final UI in some implementation language (for example using XHTML). The implementation resulting from the transformation of the desktop CUI is shown in Fig. 12. The presentation consists of three parts (*Search*, *Search result* and *Trailer*) implemented by *div* HTML tags, each one containing the HTML implementation of the corresponding concrete interactors: the text fields are rendered using the standard *text-field* forms, the button is a *submit input type*. Each result entry (repeated following the *repeat-content* semantics) contains the image representing the DVD description, a paragraph for the title and for the list of actors, the image for the DVD rating and a link that activates the trailer on the right part, which is implemented as a Flash video player.

The generation procedure can be also executed on the mobile CUI, in order to get an implementation also for this platform. An example result is shown in Fig. 13.

9. Evaluation

In this section we report on an evaluation of the MARIAE authoring environment performed through two user tests.

Six people, mainly recruited from the Institute staff, all male graduates in Computer Science, performed the first test. The mean age of the participants was 27.9 ± 1.83 (mean value plus or minus standard deviation) years old. Taking into account that the scale considered (for both tests) is 1 to 5 (1 = the worst; 5 = the best), they all have a good experience in the development of user interfaces (3.8 ± 0.75). Their experience in developing models for user interface was sufficient (3.17 ± 0.98), with a heterogeneous experience at the various UI levels (some had created only CUIs, some only task models etc.). Their knowledge of the development of applications based on Web Services was rated as sufficient (3 ± 1.4). However, some users had broad experience in this area, while others had used Web Services only a few times.

For the test, users were requested to read a brief text introducing them to the development of UI models, and explaining the various levels of abstraction (Task, Abstract User Interface, Concrete User Interface, and Implementation). After that, they read the instructions for performing the tasks requested for the test. Once the tasks were completed, they had to fill in a questionnaire, and rate aspects of the methodology and the tool, with the possibility to add comments to their answers.

The test consisted of five parts, which allowed the user to assess the main functionalities of the tool. More in detail, users were asked to:

- Use CTTE (Mori et al., 2002) to build a task model of a simple interactive application to perform a book search by keyword. The system has to search the occurrences of the word in the stored data and show the results. After that, one of the results can be selected to get more details about it.
- Import the CTT model just created into the MARIAE authoring environment. After this, the user has to open an annotation file

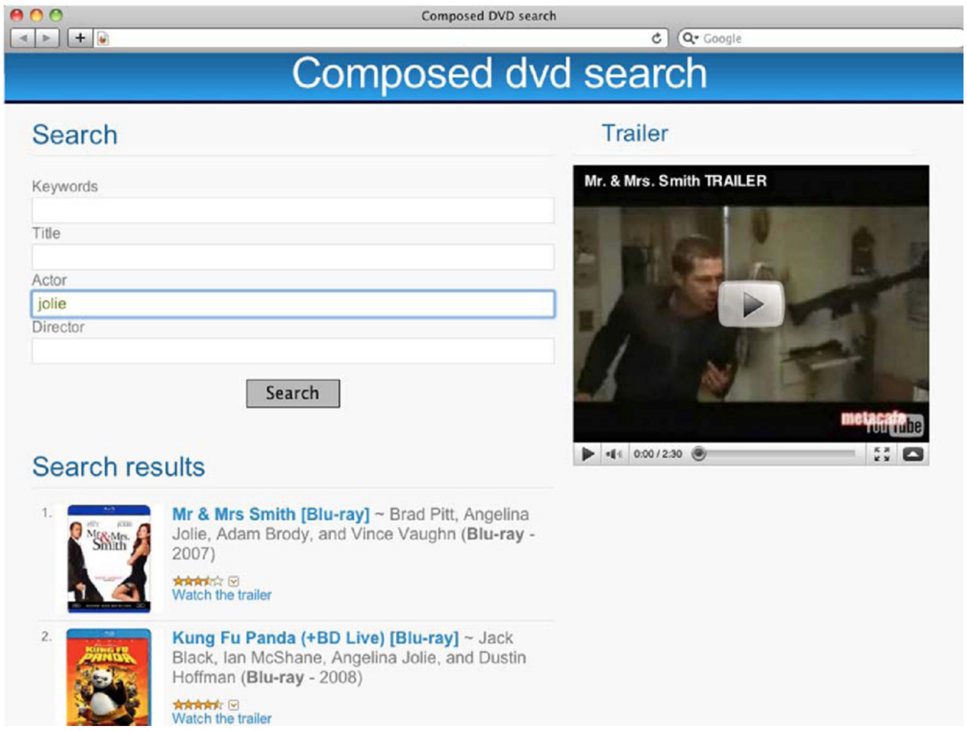

Composed DVD search
Composed dvd search
Search
Keywords
Title
Actor
jolie
Director
Search
Trailer
Mr. & Mrs. Smith TRAILER
Search results
1. Mr & Mrs Smith [Blu-ray] ~ Brad Pitt, Angelina Jolie, Adam Brody, and Vince Vaughn (Blu-ray - 2007)
Watch the trailer
2. Kung Fu Panda (+BD Live) [Blu-ray] ~ Jack Black, Ian McShane, Angelina Jolie, and Dustin Hoffman (Blu-ray - 2008)
Watch the trailer

Fig. 12. Final UI of the example.

(created for the test by us) associated to the Amazon Web Service, and bind the necessary service operations to the system tasks.

- Generate a first draft of the AUI based on the task model and the annotations about the Web Service operations bound to the model, and then refine it using the editing functionalities of the tool.
- Create a model-to-model transformation between the AUI model and the CUI desktop model for the elements considered in the example that they have developed. Then apply it to the AUI created.
- Edit the resulting CUI using the tool.

From the test we obtained the following results. The possibility to use annotations on Web Services for creating the front ends was particularly appreciated (4.5 ± 0.55) and all the comments further pointed out the fact that it can help in speeding up the development. The modelling of applications based on Web Services using task models was also highly rated (4.16 ± 1.16). Only one user did not find the approach intuitive: he would have preferred directly starting with the AUI model. Another one wanted to have more automatic support from the tool during the task model creation: if the developer chooses to use a Web Service operation, a first corresponding task model should be automatically generated from

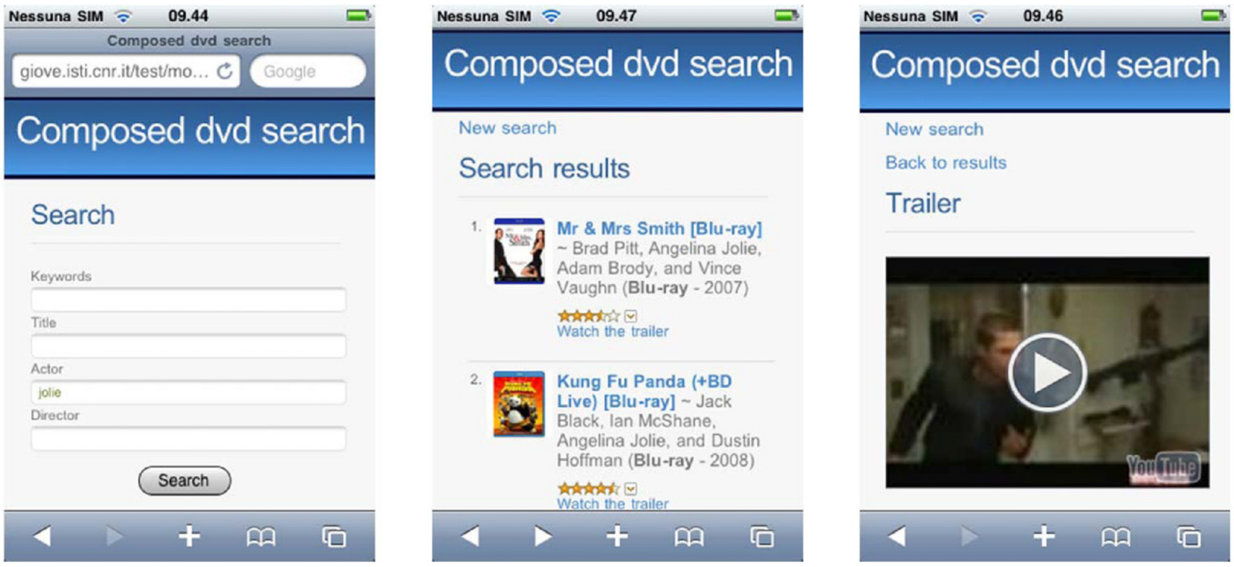


Fig. 13. Final UI for the mobile platform.

the operation definition. The users liked the interface for supporting the binding between system tasks and Web Service operations (4 ± 0.9) and the comments indicated the need for more feedback and help at first usage. All users appreciated the display of services and annotations (4.33 ± 0.51).

The automatic creation of a first draft of the Abstract User Interface starting from the task model was considered useful (4.16 ± 0.75) and the AUI editing interface was appreciated (4 ± 0.89). The users asked for a better preview mechanism of the created user interface and support for manually changing the code. The editing at the concrete level of the interface was rated very good (4.5 ± 0.55). One suggestion during the test was to integrate the representation of the Web Services and their annotations. Indeed, they were originally in two different windows (one on the left and one on the right) and users had difficulty in understanding which elements of the Web Services each annotation referred to. Thus, we changed their visual representation as shown in Fig. 4, where they are integrated in a single window: in this way, when a Web Service element is annotated, the corresponding annotation is shown immediately below.

Afterwards, we conducted a second study with designers and developers working in companies. We involved nine people working in various industrial organisations from various countries, some of them in very large European ICT companies. The average age was 32.5, with 4 graduates and 5 post-graduates. In terms of experience/time working in their companies there was one person with less than one year, 5 between one and five years, one between five and ten years, and two with over ten years. On a one-to-five scale they rated themselves 3.89 ± 1.05 in terms of experience in user interface development, and 3.67 ± 1.22 in terms of Web Service knowledge. Only five of them had experience in terms of model-based development.

Unlike the first test, the second one was remotely conducted: by email the users received the tool, some basic explanations of the goals of the test, some background information together with a short text explaining how to use relevant features of the tool along with a task model used for the test and describing the activities required to manage customers' orders. Then, they had to perform a number of tasks:

- Import the task model within the MARIAE tool.
- Display the bindings available in the project; then add further bindings to the project
- Select and apply some heuristics to obtain Presentation Task Sets corresponding to abstract presentations of the AUI that will be generated. Then, make some further refinements to the obtained AUI (by editing labels, etc.).
- Refine the AUI to obtain a Concrete UI for the desktop platform. Then, further modify/refine the CUI.
- Generate the Final User Interface in HTML language. If they were not happy with the resulting UI, they could further modify/refine the Final UI by changing the associated Concrete UI.

After the exercise users had to fill in a questionnaire through which they had to rate a number of features of the tool. The use of annotations associated to the operations of WS for developing UIs accessing the service was rated well (3.78 ± 0.97). The visualization of services and the related annotations in the tool seemed to be a bit more problematic (3.22 ± 0.97). The critical point was the panel showing the information regarding Web Services and annotations. This information can be very long because of the possible high number of available operations and developers can get lost while looking for the desired piece of information. Therefore, a search feature has been added to more easily look for specific elements in such descriptions. The technique used to carry out the binding between the operations of WS and the application tasks received a better evaluation (3.56 ± 1.42)

The transformation that automatically derives a draft of the Abstract User Interface was the most highly appreciated feature (4.00 ± 0.87). The editing modality of the abstract user interface (3.25 ± 1.16) was a bit more problematic. The underlying language (MARIA) evolved during the tool development and its attributes were increased to allow a better specification of realistic case studies. Thus, in order to address such issues, it has been necessary to reorganise this part by structuring it in panels depending on the attribute type considered. The editing modality of the Concrete UI was more intuitive and understandable (3.5 ± 1.07). The most problematic part was the quality of the obtained Final UI (2.5 ± 1.27). The problem was mainly due to some bugs in the user interface generator, which did not produce the expected results in a few cases. The bugs have been removed thanks to the test, which was useful to identify them. The rating of the overall approach proposed for the development of applications based on Web Services was good (3.67 ± 1.22).

The overall impression of the methodology and the tools for supporting it was good and the comments underlined the benefits for the creation of service front ends. There were also some requests to make the overall process more straightforward, since sometimes it can be over-elaborate. Thus, a number of other small modifications have been performed to the tool in order to improve its usability taking into account the feedback reported in the test and other informal tests. Since the process goes through various phases usually generating files with the relevant information, initially it was mandatory to explicitly save such files when created and then load them when necessary. Then, in order to limit such repetitive operations we introduced the possibility to create projects able to include all the user interface specifications and bindings created so that developers can activate them by directly selecting the project elements represented in a specific tab.

All in all, both studies have shown that the development environment, although in a prototype stage, was judged as promising in the provided features, in the capability of supporting the underlying method, and in providing designers and developers with an effective mean for exploiting Web Services in interactive applications in multi-device contexts.

10. Conclusions and future work

We have presented a method for the development of user interfaces for applications based on Web Services, together with the associated tool support, which is available for public download at http://giove.isti.cnr.it/tools/Mariae/. We have shown how task models developed with user involvement can be used to describe how activities should be performed, including those implemented through the Web Services. Then, the resulting task models can be applied to start the generation of corresponding user interface transformations that aim to preserve the usability of the corresponding models. This process can also exploit information from specific Web Services annotations, which allow their developers to provide hints about related aspects. The environment allows designers to specify and customise transformations between various levels of detail of the user interface description. We have also illustrated our approach through an example.

Two types of integration of usability aspects have been envisaged in the proposed approach to the software development process. One is from the point of view of the users, through the use of a task model validated by the end users, which should describe how the activity should be carried out. Generally the development of a task model is also useful because it drives the multidisciplinary team (usually involved in its development) to think more carefully about the relationships between the different activities. The identification of such relationships represents a good reference point for the evaluation phase, since such relationships can be compared with the actual interaction behaviour in order to understand if any deviation has occurred. The designer's perspective has also been supported in our method by including usability guidelines within the tool itself (i.e. in the transformation engines supporting progressive refinement of the UI descriptions, for instance the ability to identify suitable interactors for supporting the various activities). This should ensure that even the first user interface drafts produced by the tool account for usability issues. In this way the number of user tests (and consequent software refinements) needed to obtain high quality software should decrease.

Future work will be dedicated to further testing the usability of the authoring environment involving designers and developers from various organizations, and providing support for end-user development in order to make the authoring of interactive service-based applications possible even for people with limited programming knowledge.

Acknowledgment

We gratefully acknowledge support from the EU FP7 STREP ServFace Project (http://www.servface.eu).

References

Abrahao, S., Insfran, E., 2006. Early usability evaluation in model driven architecture environments. In: Proceedings of International Conference on Quality Software, vol. 0 ,. IEEE Computer Society, pp. 287–294, http://doi.ieeecomputersociety.org/10.1109/QSIC. 2006.26.

Agrawal, A., Amend, M., Das, M., Ford, M., Keller, C., Kloppmann, M., König, D., Leymann, F., Müller, R., Pfau, G., Plösser, K., Rangaswamy, R., Rickayzen, A., Rowley, M., Schmidt, P., Trickovic, I., Yiu, A., Zeller, M., 2007 June. Web Services Extension for People (BPEL4People), Version 1.0.

Bastien, J.M., Scapin, D.L., 1993. Ergonomic Criteria for the Evaluation of Human–Computer Interfaces. Tech. Rep. n. 156. INRIA, Rocquencourt, France.

Calvary, G., Coutaz, J., Bouillon, L., Florins, M., Limbourg, Q., Marucci, L., Paternò, F., Santoro, C., Souchon, N., Thevenin, D., Vanderdonckt, J., 2002. The CAMELEON Reference Framework, Deliverable 1.1, CAMELEON Project, http://giove.isti.cnr.it/projects/cameleon/pdf/CAMELEON%20D1.1RefFramework.pdf.

Daniel, F., Yu, J., Benatallah, B., Casati, F., Matera, M., Saint-Paul, R., Understanding UI, 2007. Integration: a survey of problems, technologies, and opportunities. IEEE Internet Comp. 11 (3), 59–66.

Diaper, D., Stanton, N.A. (Eds.), 2004. The Handbook of Task Analysis for Human–Computer Interaction. Lawrence Erlbaum Associates.

El Bekai, A., Rossiter, N., 2005. A tree based algebra framework for XML data systems. In: ICEIS 2005, Proceedings of the Seventh International Conference on Enterprise Information Systems , Miami, USA, May 25–28, 2005.

Kassoff, M., Kato, D., Mohsin, W., 2003. Creating GUIs for Web services. IEEE Internet Comp. 7 (5), 66–73.

Lepreux, S., Vanderdonckt, V., Michotte, B., 2006. Visual design of user interfaces by (de)composition. In: Proceedings DSV-IS 2006 ,. LNCS, Springer, pp. 157–170.

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Lopez-Jaquero, V., 2004. USIXML: A Language Supporting Multi-path Development of User Interfaces. EHCI/DS-VIS, pp. 200–220.

Lin, J., Landay, J., 2008. Employing Patterns and Layers for Early-Stage Design and Prototyping of Cross-Device User Interfaces. CHI, pp. 1313–1322.

Lizcano, D., Soriano, J., Reyes, M., Hierro, J., 2008. EzWeb/FAST: reporting on a successful Mashup-based solution for developing and deploying composite applications in the upcoming 'ubiquitous SOA'. In: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services , pp. 15–19.

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, Italy.

Manolescu, I., Brambilla, M., Ceri, S., Comai, S., Fraternali, P., 2005. Model-driven design and deployment of service-enabled Web applications. ACM Trans. Internet Technol. 5 (3), 439–479.

Mori, G., Paternò, F., Santoro, C., 2002. CTTE: support for developing and analysing task models for interactive system design. IEEE Trans. Softw. Eng. 28 (August (8)), 797–813, IEEE Press.

Mori, G., Paternò, F., Spano, L.D., 2008. Exploiting Web services and model-based user interfaces for multi-device access to home applications. In: DSV-IS 2008 , Kingston, Canada, July 2008. Springer Verlag, LNCS, pp. 181–193.

OASIS: BPEL – Web Services Business Process Execution Language, Version 2.0, 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

Object Management Group (OMG): Business Process Modelling Notation (BPMN) 1.2, 2009 http://www.omg.org/spec/BPMN/1.2/.

Obrenovic, Z., Gasevic, D., 2008. End-user service computing: spreadsheets as a service composition tool. IEEE Trans. Services Comp. 1 (4), 229–242.

Paternò, F., 2000. Model-Based Design and Evaluation of Interactive Applications. Springer Verlag.

Paternò, F., Santoro, C., Spano, L.D., 2009a. Support for authoring service front-ends. Proceedings of ACM EICS. ACM Press, pp. 85–90.

Paternò, F., Santoro, C., Spano, L.D., 2009b. MARIA: A Universal Declarative Language for Service-Oriented Applications in Ubiquitous Environments, ACM Transactions on Computer-Human Interaction. ACM Press, pp. 19:1–19:30.

Paternò, F., Santoro, C., Spano, L.D., 2010a. Exploiting web service annotations in model-based user interface development. In: Proceedings of ACM EICS ,. ACM Press, pp. 219–224.

Paternò, F., Santoro, C., Spano, D. (Eds.), 2010 July. Models for Service Annotations, User Interfaces, and Service-based Interactive Applications, Servface D2.9 (Final Version). , Available from http://www.servface.eu/.

Paternò F., Santoro C., Spano L.D. The role of HCI models in service front-end development. Behaviour & Information Technology, 1362–3001, First published on: 27 April 2011 (iFirst). http://dx.doi.org/10.1080/0144929X.2011.563795.

Pinna-Dery, A.M., Fierstone, J., Picard, E., 2003. Component Model and Programming: A First Step to Manage Human Computer Interaction Adaptation. Mobile HCI, pp. 456–465.

Raphael, B., Bhatnagar, G., Smith, I.F., 2002. Creation of flexible graphical user interfaces through model composition. Artif. Intell. Eng. Des. Anal. Manuf. 16 (June (3)), 173–184, http://dx.doi.org/10.1017/S0890060402163049.

Song, K., Lee, K.-H., 2008. Generating multimodal user interfaces for Web services. Interacting Comp. 20 (4–5, September 2008), 480–490.

Spillner, J., Feldmann, M., Braun, I., Springer, T., Schill, A., 2008. Ad Hoc Usage of Web Services with Dynvoker, Towards a Service-Based Internet. LNCS 5377, Springer, pp. 208–219.

Van der Veer, G., Lenting, B., Bergevoet, B., 1996. GTA: groupware task analysis – modelling complexity. Acta Psychol. 91, 297–322.

Vermeulen, J., Vandriessche, Y., Clerckx, T., Luyten, K., Coninx, K., 2007. Service-interaction descriptions: augmenting services with user interface models. In: Proceedings Engineering Interactive Systems 2007 , Salamanca. Springer Verlag.

Fabio Paternò is Research Director and Head of the Laboratory on Human Interfaces in Information Systems. He has been the scientific coordinator of various European Projects. His research interests include Ubiquitous Interfaces, Methods and Tools for Multimodal User Interface Design and Evaluation, Accessibility, Model-Based Design of Interactive Systems, and End-User Development. He has published about two hundred papers in refereed international conferences or journals. He is the Chair of the IFIP Working Group 2.7/13.4 on User Interface Engineering. He has also been appointed ACM Distinguished Scientist.

Carmen Santoro is a researcher of the Human Interfaces in Information Systems Laboratory of ISTI-CNR in Pisa (Italy). In 2004 she got a Ph.D. in Computer Science from University of Toulouse 1 (France). Her current research interests include methods and tools for the design, development and evaluation of interactive multimodal and multi-platform applications, with particular focus on exploiting model-based approaches. She has published papers for international conferences and journals on Human-Computer Interaction. Recently, she has been Co-Chair of EICS 2011 Late Breaking Results. Currently, she is in the International Editorial Review Board of "International Journal of Handheld Computing Research" journal.

Lucio Davide Spano is a researcher of the Human Interfaces in Information Systems of ISTI-CNR in Pisa (Italy) and a Ph.D. student in Computer Science at the University of Pisa. He had his Master Degree in Computer Science in 2009. His research interests currently include model-based approaches for multi-platform user interfaces, natural interaction through gestures and end user development. He published papers for HCI journals and conferences. He collaborated to the ServFace and Serenoa FP7 EU projects.