# Service-interaction Descriptions: Augmenting Services with User Interface Models

Jo Vermeulen, Yves Vandriessche, Tim Clerckx, Kris Luyten, Karin Coninx

Hasselt University – transnationale Universiteit Limburg,
Expertise Centre for Digital Media – IBBT,
Wetenschapspark 2, B3590 Diepenbeek (Belgium)
{jo.vermeulen, yves.vandriessche, tim.clerckx, kris.luyten, karin.coninx}@uhasselt.be

**Abstract.** Semantic service descriptions have paved the way for flexible interaction with services in a mobile computing environment. Services can be automatically discovered, invoked and even composed. On the contrary, the user interfaces for interacting with these services are often still designed by hand. This approach poses a serious threat to the overall flexibility of the system. To make the user interface design process scale, it should be automated as much as possible. We propose to augment service descriptions with high-level user interface models to support automatic user interface adaptation. Our method builds upon OWL-S, an ontology for Semantic Web Services, by connecting a collection of OWL-S services to a hierarchical task structure and selected presentation information. This allows end-users to interact with services on a variety of platforms.

**Keywords:** Model-based user interface development, Semantic web services, Screen layout, Automatic generation of user interfaces, User interface design, Ubiquitous computing

## 1 Introduction

In this paper, we introduce a framework to design *services* that automatically present a suitable user interface (UI) on a wide variety of *computing platforms*.
The main objective of our system is to allow mobile users to flexibly interact with services in a city environment. A city environment is often very volatile. Users come and go, carrying with them different devices and having different needs for the resulting user interface (e.g. a visually handicapped person might prefer speech interaction).

By *service*, we refer to an application that provides useful functions to end-users. Users interacting with these services use a variety of devices with different operating systems and user interface toolkits. A *computing platform* is the combination of a device, operating system and toolkit. The user interface for a service thus runs on a computing platform.

The city environment we described roughly corresponds to the vision of ubiquitous computing [26]. Its goal is for users to move through their environment, finding resources and services as they go, and to have those services provided in the context of their physical environment. This vision is slowly becoming a reality with the increasing market penetration of ever more capable mobile devices, the availability of advanced sensors and cheaper network access.

Semantic service descriptions are more and more used to describe services in a ubiquitous computing environment. Discovering, invoking and even composing these services based on their semantics has already proven effective. Unfortunately, the resulting user interface was left out of the equation. Usually, the user interface for interacting with a service is still designed by hand. This seriously decreases the flexibility of the system. Designing each user interface by hand requires prior knowledge of the available services, their inner workings and possible service compositions, not to mention the computing platform where the user interface has to be deployed and the context-of-use.

People will use services as they become available. However, the designers of a service may have never anticipated the user's device as a target platform. It is not reasonable to require services to have a custom-made user interface available for each possible situation, neither is it reasonable the other way around, to require each target platform to support every possible service. A more general solution is needed.

Our approach uses existing metadata about semantic web services and custom, high-level annotations about the resulting user interface, to allow for advanced adaptation to any target platform. These custom annotations link user interface models with the logical components of the service. We call the resulting service description a *service-interaction description*.

We describe three contributions in this work:

- The combination of semantic service descriptions with a model-based user interface development approach. While annotating service descriptions with user interface information has been explored before, the use of model-based techniques results in a higher degree of abstraction, enabling adaptation to any target platform.
- The creation of an extensible semantic network[1] of presentation information which is used to model an extra layer of abstraction on top of the User Interface Markup Language (UIML). By providing the link between the abstract and concrete presentation information, we are able to perform an automatic mapping from the former to the latter.
- A hierarchical and reusable graphical layout model that describes layout on a concrete level while keeping the interface flexible. We obtain this through the use of spatial constraints and by connecting the layout to the abstract user interface. With this we attempt to comply to the plasticity requirements inherent in user interfaces for services that have to be deployed on a variety of platforms.

The remainder of the paper is organized as follows. The next section discusses related work. Then, we give an architectural overview of our approach. Subsequently,

[1] A semantic network is a form of knowledge representation, consisting of concepts and semantic relations between these concepts.

the details of service-interaction descriptions are discussed (Sect. 4). Sect. 5 gives an overview of how high-level user interface models are transformed into a concrete user interface. First, we describe the central model in our approach: an annotated task model which will be used to extract the dialog model (Sect. 5.1). Secondly, we introduce the semantic network built on top of UIML and explain how it can be used to perform automatic widget selection (Sect. 5.2). Next, we discuss layout templates which can be used to position the selected widgets for the graphical modality (Sect. 6). After presenting the main ideas, we provide a walkthrough of the design of a photo sharing service using our system (Sect. 7). Finally, we draw some conclusions while looking ahead for possibilities in future work.

## 2 Related Work

Much work has been done in combining service descriptions with user interface information. We do not aspire to give a complete overview of the existing work in this area. Yet, we have selected a couple of notable examples which we feel are most relevant for this paper. We believe our approach is unique in that the use of high-level user interface models results in a higher level of abstraction while still offering the possibility for manipulating the final presentations. In addition, by building on semantic web services it is possible to leverage the existing work in automated discovery, invocation, composition and monitoring of these services.

XWeb is inspired by the architecture of the World Wide Web. It allows a variety of interactive platforms to communicate with services by means of a uniform protocol [8]. Service providers specify XViews that define the interaction with the data of the service, in a device-independent manner. The clients themselves decide how to render these XViews. A drawback of XWeb is that each client must know when to request the correct XView. There is no information about the structure of the user interface, and in which way an end-user will interact with the service. We, on the other hand, do provide this information through the task and dialog model.

Khushraj et al. [14] also use OWL-S service descriptions and augment them with user interface information to generate personalized user interfaces. Their system is oriented towards automated form-filling based on context information, which means the user interface annotations are too concrete to be useful for the problems that are targeted in our approach.

ICrafter [23] is an architecture to select, generate and/or service user interfaces at runtime. The authors also state that they support aggregation of service UIs. User interface generators be written for patterns of services, which are services conforming to a common programmatic interface. In fact, this comes down to providing the same user interface for a collection of services with similar semantics, instead of merging two existing service UIs. Semantic web services already solve the problem of composing the functional descriptions of two services, but in a more generic way. A disadvantage of ICrafter is the fact that the appliance-specific UI generators have to be programmed by hand. This means that whenever a new target platform has to be supported, a corresponding UI generator needs to be created. The use of a concrete abstraction layer (UIML in our approach) solves this problem.

Manolescu et al. [21] describe a model-driven design and deployment process for integrating web services with web applications that have a predefined user interface. Another example of the combination of WSDL service descriptions and user interface models is the CATWALK framework [25]. This framework mainly concentrates on the creation of the actual web pages that interact with the services.

## 3 Architectural Overview

The work we present in this paper enables users to flexibly interact with services. Our approach is centered on the combination of semantic service descriptions and high-level user interface models [10]. Fig. 1 illustrates how the system can create a suitable user interface to allow an end-user to interact with a particular service.

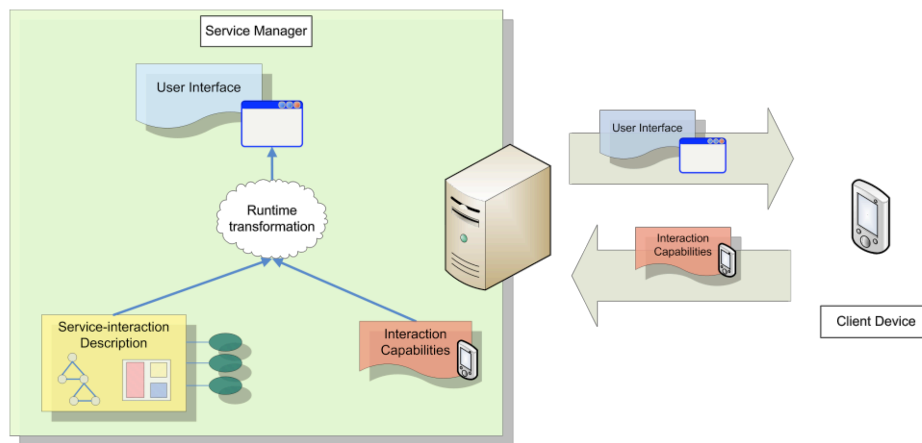



**Fig. 1.** Architectural overview.

The client device on the right wants to make use of a particular service. To do so it sends a *service-interaction request* to the *Service Manager*. This request consists of a description of the client platform's interactive capabilities, together with a reference the service it wants to address. First, the Service Manager looks up the correct service-interaction description which consists of both the functional description and the user interface information. Then, the service's high-level user interface information is combined with the knowledge of the client's interaction capabilities to form a concrete user interface for the client platform. This transformation is performed *at runtime*. Finally, the Service Manager sends a *service-interaction response* to the client, containing the user interface for the requested service.

The next section will discuss the creation of service-interaction descriptions. Afterwards, Sect 5 and 6 will explain in detail the transformation of high-level user interface information into a concrete user interface.

## 4 Service-interaction Descriptions

The first step in our approach is to extend service descriptions with user interface information. First, we define the terms *web service* and *service description*. The World Wide Web Consortium (W3C) defines a *web service* as "a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks" [15]. A web service generally provides a *service description*, which includes a description of its interface among other information (e.g. the URL[2] where the service can be reached). Most existing web services use the Web Service Description Language (WSDL)[3] for this purpose.

Although it is possible to augment WSDL with user interface information (as demonstrated by Kassof et al. [13]), WSDL's lack of semantics makes it very difficult to generate a suitable user interface. The Semantic Web is a vision of the next generation of the World Wide Web, characterized by formally described semantics for content and services [2]. These semantics are described by knowledge representation languages such as the Resource Description Framework (RDF)[4] and the Web Ontology Language (OWL)[5] RDF and OWL, in turn, refer to *ontologies*, specifications of conceptualizations [11], which enable *reasoning* through the use of logic rules. *Semantic web services* originate from the augmentation of web service descriptions with formal semantics. The extra semantics facilitate the automation of discovery, invocation, composition and monitoring of these services. It is exactly this new "extension" that enables us to automatically generate suitable user interfaces for web services. First, the added semantics are useful for selecting an appropriate presentation (e.g. the meaning of inputs and outputs). Secondly, we can easily link the service with our own semantics, which is in this case the high-level user interface information. We chose to use OWL-S[6], an OWL-based web service ontology. An OWL-S service can be mapped to a concrete realization of the service (such as a WSDL description). This means existing web services can be reused and extended with an OWL-S description.

We should note however that there is an important difference between an end-user's perception of a service and what is described in an OWL-S service description. For example, the Google search WSDL description[7], defines three basic operations: `doGetCachedPage`, `doSpellingSuggestion`, and `doGoogleSearch`. If we convert this WSDL file to OWL-S, we end up with three different OWL-S services (one for each operation). It is not possible to describe these operations as a single OWL-S service since they each have different inputs and outputs. After all, an OWL-S service advertises itself by its functional description which includes the accepted inputs and outputs. Nevertheless, the end-user views the entire WSDL description (the combination of search, spelling suggestions and cached pages) as a single service provided by Google. To prevent confusion, our notion of a service

---

[2] Uniform Resource Locator
[3] http://www.w3.org/TR/wsdl
[4] http://www.w3.org/TR/rdf-concepts/
[5] http://www.w3.org/TR/owl-features/
[6] http://www.w3.org/Submission/OWL-S/
[7] http://api.google.com/GoogleSearch.wsdl

should correspond to the one of the end-user. The high-level user interface information for this kind of service will thus often cover multiple OWL-S services. This means a number of OWL-S services will have to be coupled into a custom service description which is in turn linked to the abstract user interface. We define this as a *service-interaction description*, since it contains the necessary information to allow both machines and humans to easily interact with a particular service.

The abstract user interface of service-interaction descriptions is based on a hierarchical task model which describes the tasks that can be performed by users in order to reach a goal. We describe this task model with the ConcurTaskTrees (CTT) notation [22]. Tasks can be decomposed into subtasks, resulting in a hierarchical tree structure. The deeper we go into the hierarchy, the more concrete the tasks are. The task model can be used to extract more concrete models, such as the dialog model and presentation model [19]. Elements from the dialog and presentation models are associated with leaf tasks[8]. The designer also has to link these leaf tasks to service components, which as a result provides the link between the user interface models and the service descriptions. The next section provides more details on how this allows the abstract user interface information to be translated to a concrete user interface.

Fig. 2 shows the different components of a service-interaction description. It combines a hierarchical task model with a layout model and a number of OWL-S services. These services can be grounded into a single WSDL description for easy invocation by the concrete user interface.

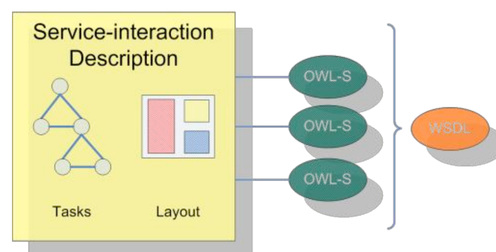



**Fig. 2.** An overview of the components of a service-interaction description.

## 5 Producing the concrete user interface

The previous section described how semantic web services were augmented with high-level user interface models. These models provide enough abstraction to be applicable for every computing platform. However, to be actually useful, they have to be translated into a concrete user interface for a specific platform. This section will discuss how we perform this transformation.

First, we give an overview of the four levels of abstraction for multi-platform user interfaces, as defined by the CAMELEON Reference Framework [4] (sorted from the

[8] Leaf tasks are the most concrete tasks: they cannot be decomposed further into subtasks.

most concrete to the most abstract level): (1) the *Final User Interface* (FUI) is the operational UI; (2) the *Concrete User Interface* (CUI) expresses any FUI independently of any markup or programming language; (3) the *Abstract User Interface* (AUI) expresses any CUI independently of any interaction modality (e.g. graphical, vocal, tactile, ...) via the mechanism of Abstract Interaction Objects (AIOs) as opposed to Concrete Interaction Objects (CIOs) for the CUI and Final Interaction Objects (FIOs) for the FUI; and finally (4) the *Task and Concepts* level, which describes the various interactive tasks to be carried out by the end user and the domain objects that are manipulated by these tasks.

The service-interaction descriptions contain a hierarchical task model in the ConcurTaskTrees (CTT) notation [22], which corresponds to the Tasks and Concepts level. We assume that each client device knows how to transform a CUI to a FUI. This means the transformation process ranges only from the Task and Concepts level to the CUI. First, the task model should be transformed into an AUI, whereafter this AUI is transformed into a CUI. The next section discusses the first mapping, while Sect. 5.2 provides more details about mapping the AUI to a CUI, for which we use the UIML language.

### 5.1 Annotating the task model

In order to ease the transition from the task model to an AUI, we annotate leaf tasks with service components and AIOs. This requires the task model to be decomposed up to the level that each leaf task can be connected to a single AIO and service component. A *service component* can be an input or output of an OWL-S service or the service itself.

An important step in the transformation to the AUI is the extraction of a dialog model. The dialog model is a State Transition Network (STN), modeling the possible states of the user interface. In each state, a “dialog” is conducted between the user and the system. We use the annotated task model to generate a corresponding dialog model [19]. Each state in this model is an *Enabled Task Set (ETS)*. An ETS is a collection of tasks that are enabled during the same time period, which means they should be presented to the user simultaneously, i.e. in the same dialog [22].

In conclusion, our AUI consists of the annotated task model and the extracted dialog model. We now know of which states the user interface is comprised and which leaf tasks belong to these states. The fact that these tasks are annotated with AIOs and service components will prove useful in the next section.

### 5.2 Widget selection through enhanced UIML metadata

The next step is to transform the AUI into a CUI. As described earlier, we assume that each client device knows how to present a CUI to the user. For the CUI level, the User Interface Markup Language (UIML) [1] is used.

UIML is an XML-based language to describe the structure, style, content, and behavior of a user interface. Unlike other user interface markup languages, UIML does not use metaphor-specific tags (such as `window` or `button`), but only generic

tags (e.g. `part`, `property`, ...). These generic tags can be associated with a set of abstractions, defined in the `peers` section. The `peers` section specifies how these abstractions can be translated into a final presentation. Basically, the abstractions define a *vocabulary* of classes and names to be used with a UIML document. Since the vocabulary is specified separately, new devices and UI metaphors can be supported when they become available in the future. The CIOs will be defined by this vocabulary.

We use UIML solely for the CUI level, because its level of abstraction is not sufficient for covering different platforms with widely varying interaction mechanisms. The vocabulary can only provide a very thin layer of abstraction above the target platform since it uses a one-to-one mapping of an abstraction to a final widget. If we situate UIML in the CAMELEON framework [4], it only covers the concrete and final level. The vocabulary can thus be seen as a one-to-one mapping from concrete interactors (CIOs) to final interactors (FIOs). Although it is possible to describe abstract interactors (AIOs) with UIML, we would then have to map them directly to FIOs. This is too big of a step to be feasible for every possible platform and interaction mechanism.

The remaining problem now is how to perform a smooth transition from the AUI to UIML. Most tools (e.g. DynaMo-AID [6]) often only define this mapping internally. In our opinion, it is better to specify this information externally in a machine-readable way.

An interesting approach to connect the different levels of abstraction is described by Demeure et al. [9]. They have exploited a semantic network of the concepts and relationships that are involved at each level of abstraction to pose interesting questions about a running user interface. For example, one could ask "What are the alternative CIOs for the CIO *ListBox*?" This would allow us to perform automatic widget remapping just by reasoning about the semantic network. Adaptation rules would not have to be hard-coded into the software or into the user interface design. Demeure's semantic network is defined in a custom format, which complicates interoperability with other software. With the advent of the Semantic Web [2] however, the Resource Description Framework (RDF) has been widely accepted as the standard format for representing knowledge.

**A semantic network built on top of UIML.** We adopt the approach presented in [9] by Demeure et al., and adjust it to our system. We will use RDF to describe the UIML `peers` section and link it with an external AIO classification, thereby building our own semantic network. An additional advantage of using RDF is the easy integration with service-interaction descriptions, which are also described with RDF. Since the UIML vocabulary covers the concrete and final levels, the first step is to express this information with RDF.

We defined a *peers* ontology[9] by performing a straightforward mapping from UIML tags to OWL classes. The four concepts (and therefore OWL classes) defined in this ontology are: *Presentation*, *DClass*, *DProperty*, and *DParam*. A simple tool was developed to convert an original UIML vocabulary to its RDF representation and vice-versa.

---

[9] This ontology is available at http://research.edm.uhasselt.be/~uiml/peers/elements/0.1

In order to connect the concrete and abstract levels, we extend the ontology with the concept of an *AIO* and the relationship *reifies*. The *reifies* relationship works on a DClass and an AIO instance, to indicate that the former is a concretization of the latter. Note that we do not explicitly define an ontology of AIOs. Our ontology only defines the AIO concept, and the relationship that links it with a DClass. This approach is necessary to provide the same level of flexibility for the abstract level as the UIML vocabulary provides for the concrete level. It allows AIO classifications to be specified separately in external ontologies. The only requirement for this is that the different AIOs are specializations of our AIO concept, so that they can be linked with a DClass instance.

Of course, in order to actually link CIOs with AIOs, we first need to define a set of AIOs that we can use. According to the definition from [4], AIOs should be modality-independent. We will use a very high-level, minimal set of AIOs that are differentiated according to the functionality they offer to the user: (1) *input* components allow users to enter or manipulate data; (2) *output* components provide data from the application to the user; (3) *action* components allow a user to trigger some functionality; and finally (4) *group* components group other components into a hierarchical structure. We define these four AIOs (*Input*, *Output*, *Action* and *Group*) in an external ontology as the only instances of the AIO concept of our peers ontology.

**Adding data types.** A disadvantage of the generic, modality-independent AIO classification we just discussed is the fact that each AIO applies to a large number of CIOs. This means that extra information is required in order to select the correct CIO for a given AIO. The service description provides us with the associated data type, which allows us to narrow down the number of possible CIOs. Consider for example the AIO *Input*. This AIO can map on different CIOs such as a combo box, a spin box, a text entry, a check box, a radio button, or a calendar. However, if we add the constraint that the data type should be a *boolean*, our choice is automatically limited to the check box and radio button.

The concept *DataType* and the relationship *hasDataType* was added to our peers ontology, in order to relate DClass instances with a data type. Again, data types can be defined externally, to allow for maximum flexibility. We created a data type classification, based on XML Schema[10]. The ontology consists of the primitive types of XML Schema (e.g. `decimal`, `string`, `void`, etc.) in addition to a number of data types which are often used in user interfaces (e.g. `Image`, `Color`, etc.).

The leaf tasks that are annotated with an AIO and a service component provide the necessary information to be mapped on a concrete interactor. Sect. 5.1 defined a service component as an input or output of a service, or the service itself. Inputs and outputs have an associated type, while the service can be linked with the data type `Void`. However, inputs and outputs of a OWL-S service are often associated with *semantic types*, which are arbitrary concepts (e.g. *Price*). It would be unreasonable to require each OWL-S service to use our own data types. We therefore allow a service developer to link semantic types with their corresponding data type (e.g. *Price* could be linked with `Float`). This technique allows us to associate inputs and outputs of a

[10] http://www.w3.org/TR/xmlschema11-2/

service with elements from the data type ontology, while retaining the semantics of the existing OWL-S service. To do so, we extend the peers ontology with the relationship *associatedDataType* that can links arbitrary concepts with a *DataType* instance. When a leaf task is associated with an entire service (linked to the data type `Void`), it will also be coupled with the AIO *Action*. This is to indicate that a leaf task invokes a certain service. An example of a CIO that is associated with the AIO *Action* and the data type `Void` is a *Button*.

A final requirement to translate an AIO and data type to a CIO, is to indicate through which DProperty the DClass is associated with the data type. For example, the DClass *Label* can be associated with the AIO *Input* and the data type `String`, through the property *text*. We add the relationship *hasDataTypeProperty* to the peers ontology for this purpose. A UIML renderer should know how to translate each element from the data type classification to its platform-specific data type (e.g. `String` to `java.lang.String`).

**Conclusion.** The metadata we added on top of the UIML vocabulary defines a mapping from an AIO and data type tuple to a certain CIO. Fig. 3 gives an overview of the different concepts we introduced, and the relationships between them. Note that the puzzle piece on the far right represents an arbitrary concept that can be linked to a *DataType* instance.

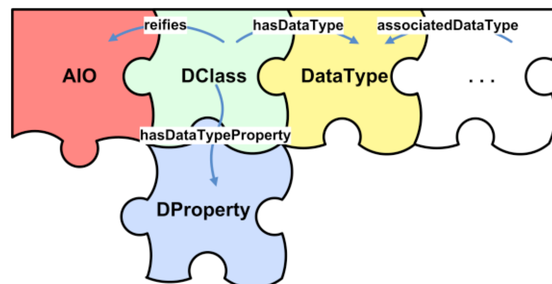



**Fig. 3.** The different concepts in the semantic network and the relationships that connect them.

The extended UIML vocabulary that was introduced here will represent a target platform's interactive capabilities. When a client device wishes to use a certain service, it sends this description to the Service Manager, as discussed in Sect. 3. In order to translate the AUI to a CUI, which is described with UIML, we use the following process. For each ETS in the dialog model, the enabled tasks are translated to corresponding CIOs, using the associated AIO and service component. To arrive at a concrete UIML user interface, a skeleton UIML description could be used, which would be filled in with the CIOs from the previous step. However, this is not an ideal solution for graphical user interfaces (GUIs). After all, static positions for the CIOs or even a standard layout will not scale between widely varying screen sizes and in addition could seriously affect the usability of the resulting user interface. The next section introduces a layout model, which we developed to overcome this problem.

## 6 Specifying the layout

This section will present a layout model, which is an extension of our approach targeted to graphical user interfaces, as discussed in the previous section. Existing work has been done in specifying the layout on the abstract user interface level, but relations between AIOs on this level are hard to map onto a concrete layout. We will therefore focus in this work on the graphical modality. The use of a layout model is still justified because there is a need for a certain amount of flexibility which cannot be obtained by a static layout specified at design time.

### 6.1 Current approaches

The most common approach to specify the layout in model-based user interface development is to group AIOs. An example of this is the hierarchically structured Logical Windows abstraction [10]. Combining AIOs under a *Group* AIO parent will guarantee that these components will stay logically grouped in the concrete user interface.

The way group AIOs are represented in the CUI will affect the eventual positions of their children. For example, group AIOs can be mapped onto a horizontal box container, which means their children are positioned on a horizontal line, from left to right. Group AIOs can be part of another group AIO, which allows a nested layout specification. However, the UI designer has only limited control over the final layout with this technique.

A pattern-based approach such as described in [24] and [18] defines layout patterns that aggregate interface elements into a specified graphical layout. In practice, these layout patterns represent simple layout containers (eg. a horizontal box). The corresponding layout model consists of layout patterns written beforehand in a template language. This technique works on a more concrete level, giving the designer a good idea of what the final UI layout will look like. However, from a modeling perspective it would be better if a designer could specify his own templates within the layout model instead of using a template language.

Another way of expressing a more concrete layout is by the use of spatial constraints on abstract UI elements [7]. This technique has been covered in many publications, such as [3] or [12]. Usually there are two approaches for obtaining these layout constraints: the designer can explicitly specify the required constraints (by means of a visual tool or by using a declarative constraint language) or constraints can be generated automatically. The latter uses either visual cues [17] or external ones such as data relationships.

Allen constraints express relationships between time and space intervals [16]. By specifying Allen relationships between AIOs we can express both spatial relationships for visual layout and temporal relationships for non-visual interfaces. Allen relationships have to be mapped onto a more concrete level, much like group AIOs. We wish to work on a more concrete level to avoid exposing the designer to this mapping problem inherent to the use of group AIOs and Allen relationships in layout design.

### 6.2 Layout Model

In this section we present a tentative approach for specifying a layout model that can be applied on the CUI level. By specifying layout on a concrete, 2D graphical level we avoid the AUI layout abstraction problem. We try to preserve the hierarchical structure introduced by group AIOs, enable reuse of patterns and allow concrete spatial constraint relations. However, we still need some of the abstractions provided by AIOs as we cannot predict the specific target platform. As seen in Fig. 4, the layout model consists of two parts, a set of *layout templates* and one of *layout instances*.

A layout template describes the structure of the layout, using hierarchical layout elements and layout relations representing spatial constraints between these elements. In Fig. 4, the root layout element of the template represented has three child elements, two leaf elements and one nested layout element which has three children of its own. The arrows between sibling layout elements represent the layout relations that exist between them. A layout template needs to be instantiated with AIOs and related with a certain state from the dialog model to be able to provide a concrete UI description. The resulting layout instance will describe the mapping between the abstract layout elements and AIOs for a single dialog. AIOs are connected to layout elements using layout instances to enable reuse of the layout templates.

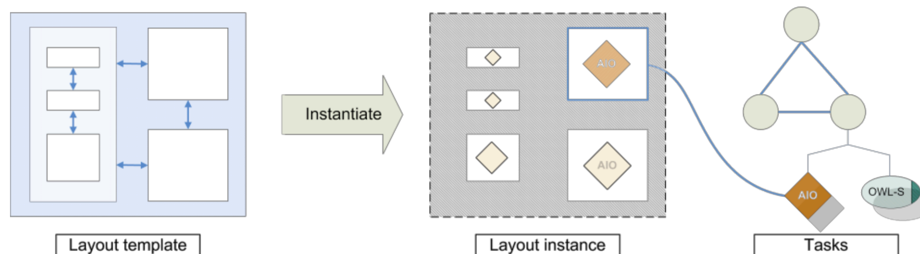



**Fig. 4.** Instantiating a layout template with AIOs.

The structure of a layout template is described by *hierarchical layout elements*. These layout elements are equivalent to group AIOs; they provide a logical window and can be nested to create a hierarchy, as explained earlier in Sect. 6.1. A logical window in this context means that layout relations can only be defined between siblings and their parent element. Layout templates differ from group AIOs in that they use geometric relations between the elements they contain to describe the actual graphical layout.

We currently use a simple set of linear geometrical constraints as an example: *align-top*, *align-center*, *left-of*, *under*, *above*, etc. In addition we also add some more complex relations: *horizontal box* and *vertical box* containers. Layout relations are abstract enough to support other types of constraints. A layout template contains a reference to a single layout element and a collection of layout relations. The referenced layout element acts as the root node of a hierarchy of layout elements. The collection of layout relations contains geometric constraints expressed between the elements of that layout element hierarchy.

A layout element in a template is a placeholder on which layout relationships such as geometric constraints are defined. During the instantiation of the layout template we can fill these placeholders with AIO elements from the abstract user interface model. A layout instance describes the mapping between layout elements and AIOs. The layout instantiation process has two main requirements. AIOs used in an instantiation have to be coupled to tasks inside the same Enabled Task Set (ETS) [22] from the dialog model. By definition, tasks in different enabled task sets cannot be shown at the same time. The designer will thus create a layout for each ETS. As a second requirement, we prohibit layout templates to be instantiated with group AIOs As mentioned earlier, group AIOs can be used to logically group AIOs on the AUI level. Since the layout inside group AIOs is unspecified, it is not possible to instantiate a layout element with a group AIO without using default layout rules. However, this would defeat the purpose of our layout specification. It is up to the designer to split the layout elements to allow a one-to-one mapping.

For this work, we use UIML to specify the concrete user interface. However, our layout model is generic enough to be mapped on other CUI representations. We generate a skeleton UIML description based on the layout instances. This skeleton contains the structure of the UI expressed as nested `part` elements. The instance's layout constraints will be mapped onto the UIML layout extension we developed in [20]. The specific `DClass` of the child parts in this skeleton will be filled in by the widget selection as explained earlier in Sect. 5.2. Our technique offers a certain amount of flexibility in the layout by the use of spatial layout constraints and a hierarchical layout specification.

## 7 Case Study

We clarify our approach by applying it to a mobile city service that allows people to share pictures with each other. Users can rate each picture of which an average rating is computed. The remainder of this section provides a walkthrough of the development of this service, which consists of four steps as shown in Fig. 5.

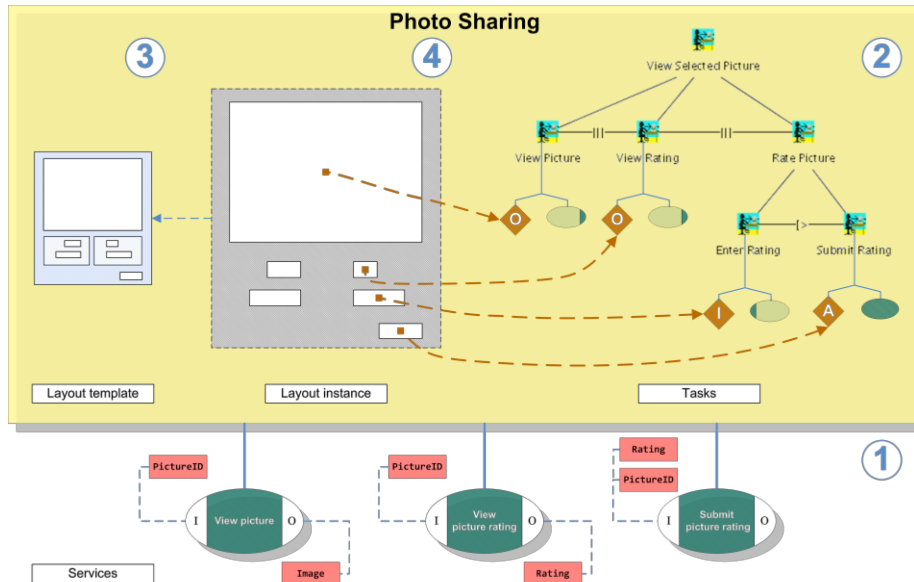



**Fig. 5.** The service-interaction description corresponding to a selected part of the photo sharing service.

To integrate the photo sharing service within our system, we need to create a service-interaction description which consists of a collection of services and a task and layout model. We extended the existing DynaMo-AID tool [6] with support for developing service-interaction descriptions. For brevity's sake, we will focus only on the functionality and corresponding user interface to show the details of a single picture. This allows users to take a look at the picture, view its average rating, and add a rating of their own.

### 7.1 Collecting the required services

The first step is to import the necessary OWL-S services, which corresponds to step (**1**) of Fig. 5. The required services for viewing a picture's details are: (i) a service to retrieve a single picture; (ii) a service to get the average rating of a picture; and finally, (iii) a service to rate a certain picture. Fig. 5 shows these services and their semantic input and output types.

### 7.2 Creating the task model

After importing the OWL-S services, the next step is to create a hierarchical task model that specifies how users will interact with the photo sharing service. The task model should be decomposed up to the level where every leaf task can be annotated with a single AIO and service component. Afterwards, the task model is used to

extract a corresponding dialog model (that is constituted of a number of Enabled Task Sets).

The part of the task model we will discuss is the interaction task *View Selected Picture* and its four subtasks, as shown in step (**2**) of Fig. 5. The task *View Picture* is annotated with the *Output* AIO and `Image` data type. *View Rating* and *Enter Rating* are both linked to the data type `Rating`, while the former has the AIO *Output* and the latter the AIO *Input*. Finally, the *Submit Rating* task is annotated with the *Action* AIO and `Void` data type.

At this point, we should also map the semantic types of the inputs and outputs to our data type classification, as described in Sect. 5.2. For example, `Rating` will be mapped to `StringEnum`.

### 7.3 Creating or reusing a layout template

Before designing the layout we need the ETS containing the tasks we discussed in step (**2**) of Fig. 5. This gives us an overview of the tasks and attached AIOs that need to be presented in a single dialog. Although an existing template (or even some of its parts) could have been reused, we create a layout template from scratch here to illustrate our technique. The layout template in step (**3**) is constructed by drawing a couple of boxes which represent the layout elements. The shape and size of the boxes are irrelevant, but their relation to each other is. The nesting of these boxes represents the hierarchy of the corresponding layout elements.

After constructing the layout element hierarchy, the designer adds layout relations to the template. Layout relations are specified explicitly by selecting the target elements (for example the two middle boxes) and by applying a geometric constraint (e.g. *align right*).

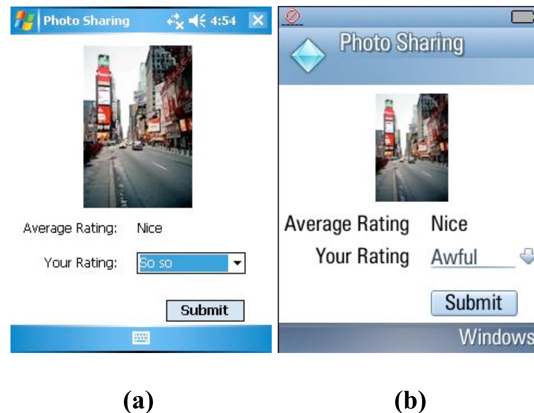
### 7.4 Instantiating a layout template

Step (**4**) in Fig. 5 depicts the instantiation of the layout template that was just created. First, the designer selects an ETS. The AIOs linked to the tasks in this ETS can then be connected to leaf layout elements in the layout template. In our example, the set of AIOs provided by the ETS is insufficient to specify the desired user interface. To add the labels "Average Rating" and "Your Rating" the designer needs to create two additional AIOs using the existing presentation model functionality in the DynaMo-AID tool [6]. The data attached to these AIOs uses the same vocabulary as explained in Sect. 5.2 to enable widget selection. This instantiation process is repeated for each ETS in the dialog model. The service-interaction description is now complete.

### 7.5 The resulting user interface

After integrating the service-interaction description in our system, users can interact with the photo sharing service. To do so, their client sends a service-interaction request along with its extended UIML vocabulary to the Service Manager,

as discussed in Sect. 3. The Service Manager then replies with a platform-specific UIML description of the corresponding user interface. Finally, the client renders the UIML code, and presents it to the user. Fig. 6 shows two examples of the resulting user interface on different platforms: (a) a PDA with the Windows Mobile operating system and Windows Forms toolkit; and (b) a Smartphone with the Symbian operating system and UIQ toolkit. Note that the photo sharing service has no specific knowledge of either of these two platforms. It just uses the metadata added to the UIML vocabulary and the specified layout instances to map the abstract user interface to a concrete one.




**(a)** **(b)**

**Fig. 6.** The final user interface for the *View Selected Picture* task on two different platforms.

## 8 Conclusions and Future Work

This paper presented *service-interaction* descriptions which combine OWL-S services with high-level user interface models in order to present a suitable user interface on any target platform. We proposed a semantic network built on top of UIML to ease the transition of the abstract to the concrete user interface. Our general approach was extended with a layout model to obtain a more visually consistent and usable UI for the graphical modality. Finally, we illustrated our approach by applying it to a photo sharing service.

We are exploring several directions for future work. First, we would like to verify the modality-independent design of the system by testing other modalities (e.g. speech). The layout model that was described in Sect. 4, would then have to be ignored since it is only useful for the graphical modality. Secondly, since it is possible to compose semantic web services, it would be interesting to investigate how the UI is influenced by this. For example, we could explore how the layout model can be modified to support this composition. In our own previous work [5] we have already taken a first step towards merging service UIs. We have shown a way to model service-aware user interfaces at the task level allowing the user interface of the main application and the one of the service to be merged into one consistent user interface.

The assumption we made was that each service would have a corresponding abstract user interface consisting of the same models as the main application. The work we presented here extends this technique at the presentation level of the user interface and explicitly links the service to the task specification.

A difficult problem concerns inconsistencies between service UIs, since the average user cannot master more than a few different user interfaces. The layout relations used in this work have been fairly straight-forward. Alternative ways of obtaining and expressing layout could be found to make the layout design process both easier and more expressive. Finally, it would be useful to extend the semantic network to allow for more advanced CIO matching. For example, CIOs could be annotated with their required size, allowing us to automatically switch to a smaller CIO when the available screen space decreases.

## 9 Acknowledgments

Part of the research at EDM is funded by ERDF (European Regional Development Fund), the Flemish Government and the Flemish Interdisciplinary institute for BroadBand Technology (IBBT). The CoDAMoS (Context-Driven Adaptation of Mobile Services) project IWT 030320 is directly funded by the IWT (Flemish subsidy organisation).

## References

1. Marc Abrams and Constantinos Phanouriou. Uiml: An xml language for building device-independent user interfaces. In *XML '99*, Philadelphia, USA, 1999.
2. Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34-43, 2001.
3. Alan Hamilton Borning. *Thinglab-a constraint-oriented simulation laboratory*. PhD thesis, 1979.
4. Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Nathalie Souchon, Laurent Bouillon, Murielle Florins, and Jean Vanderdonckt. Plasticity of user interfaces: A revised reference framework. In *Proceedings of 1st International Workshop on TAsk MOdels and DIAgrams for user interface design*, pages 127-134, 2002.
5. Tim Clerckx, Jan Van den Bergh, and Karin Coninx. Modeling multi-level context influence on the user interface. In *PerCom Workshops*, pages 57-61, 2006.
6. Tim Clerckx, Chris Vandervelpen, Kris Luyten, and Karin Coninx. A Prototype-Driven Development Process for Context-Aware User Interfaces. In *Proceedings of the 5th International Workshop on TAsk MOdels and DIAgrams for user interface design*, Diepenbeek, Belgium, October 2006.
7. K. Coninx, K. Luyten, C. Vandervelpen, J. Van den Bergh, and B. Creemers. Dygimes: Dynamically generating interfaces for mobile computing devices and embedded systems. In *Mobile HCI 2003: Proceedings of the 5th International Symposium on Human-Computer Interaction with Mobile Devices and Services*, Udine, Italy, September 8-11, 2003.
8. Dan R. Olsen Jr., Sean Jefferies, Travis Nielsen, William Moyes, and Paul Fredrickson. Cross-modal interaction using xweb. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 191-200, 2000.

9. Alexandre Demeure, Gaëlle Calvary, Joëlle Coutaz, and Jean Vanderdonckt. The comets inspector: Towards run time plasticity control based on a semantic network. In *Proceedings of the 5th International Workshop on TAsk MOdels and DIAgrams for user interface design*, Diepenbeek, Belgium, October 2006.
10. Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta. Applying model-based techniques to the development of UIs for mobile computers. In *IUI '01: Proceedings of the 6th international conference on Intelligent user interfaces*, pages 69-76, New York, NY, USA, 2001. ACM Press.
11. Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199-220, 1993.
12. Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. On Computer-Human Interaction*, 8(4):267-306, 2001.
13. Michael Kassoff, Daishi Kato, and Waqar Mohsin. Creating guis for web services. *IEEE Internet Computing*, 7(5):66-73, 2003.
14. Deepali Khushraj and Ora Lassila. Ontological approach to generating personalized user interfaces for web services. In *International Semantic Web Conference*, pages 916-927, 2005.
15. Yves Lafon. Web Services Activity Statement. http://www.w3.org/2002/ws/ Activity, 2002.
16. Quentin Limbourg. Multi-Path Development of User Interfaces, Ph. D. Thesis. PhD thesis, september 2004.
17. Simon Lok, Steven Feiner, and Gary Ngai. Evaluation of visual balance for automated layout. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, pages 101-108, 2004.
18. Frank Lonczewski and Siegfried Schreiber. The fuse-system: an integrated user interface design environment. In *Proceedings of the Second International Workshop on Computer-Aided Design of User Interfaces 1996,* pages 37-56, 1996.
19. Kris Luyten, Tim Clerckx, Karin Coninx, and Jean Vanderdonckt. Derivation of a dialog model from a task model by activity chain extraction. In *Proceedings of DSV-IS 2003: 10th International Conference on Design, Specification and Verification of Interactive Systems,* pages 203-217, Funchal, Portugal, 2003.
20. Kris Luyten, Jo Vermeulen, and Karin Coninx. Constraint adaptability of multidevice user interfaces. In *Workshop on The Many Faces of Consistency*, *CHI'2006 workshop*, April 2006.
21. Ioana Manolescu, Marco Brambilla, Stefano Ceri, Sara Comai, and Piero Fraternali. Model-driven design and deployment of service-enabled web applications. *ACM Trans. Inter. Tech.*, 5(3):439-479, 2005.
22. Fabio Paterno. *Model-Based Design and Evaluation of Interactive Applications*. Springer/Verlag, London, UK, 1999.
23. Shankar Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. Icrafter: A service framework for ubiquitous computing environments. In *UbiComp '01: Proceedings of the 3rd international conference on Ubiquitous Computing*, 2001.
24. Daniel Sinnig, Ashraf Gaffar, Daniel Reichart, Ahmed Seffah, and Peter Forbrig. Patterns in model-based engineering. In *Proceedings of Fourth International Conference on Computer-Aided Design of User Interfaces*, pages 195-208, 2004.
25. Steffen Lohmann, J. Wolfgang Kaltz and Jürgen Ziegler. Dynamic generation of context-adaptive web user interfaces through model interpretation. In *Proceedings of Model Driven Design of Advanced User Interfaces 2006*, October 2006.
26. Mark Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):75-84, 1993.