Service Discovery Supported by Task Models

Kyriakos Kritikos
HIIS Lab, ISTI-CNR
Kyriakos.Kritikos@isti.cnr.it

Fabio Paternò
HIIS Lab, ISTI-CNR
Fabio.Paterno@isti.cnr.it

ABSTRACT
We propose an approach that takes as input a task model, which includes the user's view of the interactive system, and automatically discovers a set of categorized and ranked service descriptions for each system task of the model. In this way, a set of service operations can be used to implement an application's part or whole functionality so that its development time is significantly reduced.

Author Keywords
Service front-ends, semantic service discovery, interactive application design, term to ontology concept matching.

ACM Classification Keywords
D.2.2. Design Tools and Techniques: User Interfaces. H.3.5. Online Information Systems: Web-based Services.

General Terms
Design, Algorithms

INTRODUCTION

Nowadays, an increasing number of applications is designed based on services. This is due to the ability to combine services into new integrated functionalities through service composition. In this way, a whole application can be built from scratch or existing intra- or inter-organizational applications can be integrated together in a seamless and loosely-coupled way.

Services have to be discovered before integrated into an application. To this end, many service discovery approaches have been proposed, using techniques from Information Retrieval (IR) and the Semantic Web (SW), able to match a user-provided *service query* with a set of *service advertisements* and to produce a categorization of the matched advertisements based on their matching degree. The most prominent approaches [8,12] present higher accuracy due to the use of semantics in the service description terms taken from domain ontologies. However, they assume that the requester is able to provide a semantic service description and knows well the application domain.

Concerning service composition, various approaches have been proposed [1,4,15] that focus on producing a concrete service composition plan by selecting those services that together realize the overall requested functionality.

While service discovery and composition approaches can build new functionalities from scratch without writing any single line of code, often the corresponding front-ends do not interact with users in a satisfactory way. For this reason, HCI approaches [2,7,11,13,14] have been proposed able to build service-based applications with a UI customized according to the context-of-use [9] that follow a Model-Driven Approach (MDA), which starts with a high-level model (such as a task model) and after various model-to-model transformations generates the final UI code [3].

Apart from the management of the various models produced, which can be tedious and time-consuming, another big disadvantage of the above HCI approaches is the limited automatic support. In particular, concerning the application's functionality, the designer has to manually select those services able to fulfil it, thus is required to know every possible service able to fulfil any possible functionality, which is quite impossible.

We argue that the usage of more formal representations [9], capturing the terms semantics, and reasoning mechanisms exploiting these representations, leads to the automation of the various designer-performed activities, thus overcoming the above limitation. To this end, we propose a solution that automatically selects those services able to fulfil the functionality of the application. Our solution relies on an MDA approach that automatically produces a service-based task model, called concrete service model, from an initial task model and a domain ontology providing the semantics of the tasks information. The produced model can be used to obtain interactive applications with some application logic implemented through external services.

There are various advantages in adopting our solution. First, designers are provided with useful automatic support without being required to produce the domain ontology and to annotate the task model with its concepts. Second, the application's development time is significantly reduced as a part or its whole functionality is realized through external services. Third, the use of semantics increases the service discovery results accuracy. Fourth, tasks not being supported either completely or partially by existing services in terms of input-output (I/O) parameters can be identified. Thus, existing model-based HCI approaches can use our solution as a particular automated component by replacing the corresponding non-automated ones and by providing useful hints to the designer. In this way, a value-added

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EICS'10, June 19–23, 2010, Berlin, Germany.
Copyright 2010 ACM 978-1-4503-0083-4/10/06...$10.00.

approach can be obtained that considers both UI and functional aspects to effectively design an interactive and context-of-use-aware service-based application.

The next section presents an enhancement of the interactive applications design process. Then, the proposed MDA method is analyzed, followed by a particular case study showing its benefits. The last section concludes the paper.

APPROACH

Service-based applications are not interactive enough. Moreover, functionality is not well supported during an interactive application's design. Thus, we believe that an interactive application's design process should equally consider both the UI and functionality aspects and their mutual implications, and be split into two parallel model-driven aspect-specific paths driven by the context-of-use.

Figure 1 depicts our envisioned design process. Its starting point is a task model conveying important information, such as the application's interactive and system tasks, and their temporal order and interactions. Such information can be transformed into two distinct parts: the *Abstract User Interface* (*AUI*) and the *service model*. In this way, the design of the UI and functionality aspects can be processed in parallel following model-to-model transformations on the two resulting models under the designer's supervision. In the end, the parallel design paths' low level models are joined together in the interactive application's code, which is implemented by the developer and consists of: the *Final User Interface* (*FUI*), the implemented functionality code, and the selected services' invocation code. The low-level models production follows a reification transformations path, while the propagation of changes from low to higher-level models follows an abstraction transformations path.

The UI's model-based design usually follows a particular path [3], producing the following set of models with an increasing dependency on the context-of-use: a) the *AUI*, b) the *Concrete User Interface* (*CUI*), dependent on the user's platform and interaction modality, and c) the *FUI*, an operational UI running on a particular platform either by interpretation or by execution.

The functionality design path starts with the *service model*, defining the application's abstract functionality as a service orchestration. This model is represented by a hierarchical tree having similar structure to the task model. In particular, the higher the tree node level, the more composite services are represented, so that the leaves correspond to service operations and the next level nodes to simple services. Moreover, each non-root tree node is connected with its siblings by the task model's temporal operators (e.g. of CTT [10]) in order to preserve the application's temporal semantics. There are two main reasons for using such model and not a typical service orchestration model (e.g. BPEL). First, the service model has a similar structure to a task model, so it can be used in possible abstraction transformations to produce a new task model's functional part. Second, a typical service orchestration model cannot represent high-level user goals, which can be exploited by service discovery engines to discover composite services able to fulfill their functionality. The formal definition of the proposed service model formalism is omitted as it is almost equivalent with the CTT task models one, where there is a renaming of some task types, the distinction between input and output task objects, and the inclusion of service discovery results into the task description.

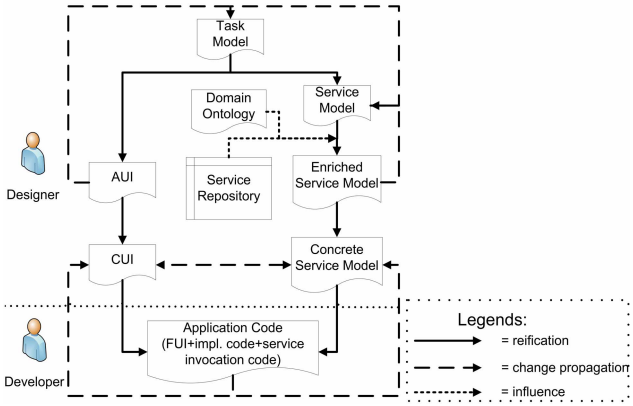

Figure 1: Envisioned design for service-based interactive applications

An *enriched service model* is produced when a service model's nodes are associated with those services realizing their functionality through service discovery. However, only some nodes of this new model will be associated with services, depending on the services world pragmatics. In particular, a part of an application's functionality may not be realized with existing services. Moreover, it is impossible to discover services corresponding to the service model's higher-level nodes, as this would require the description of the services operations' temporal semantics. The enriched service model can indicate interesting facts to the designer and developer: a) the missing functionality to be implemented and b) particular places where UI changes must be performed, e.g., additional application objects need to be attached to particular interaction tasks in case input required from the associated services is missing.

A *concrete service model* is produced from an enriched service model by performing *service concretization*, i.e. by selecting only one of the associated services of the enriched service model's leaf nodes. Service concretization may be performed using various criteria, but the most relevant are those pertaining to the syntactic and semantic similarity between the service's description and the operation's node one and the compliance to the context-of-use constraints. In this way, different concrete service models can be produced from an enriched service model at different time points, depending on the domain's dynamicity, i.e. the pace with which new services with equivalent functionality are developed or existing services are altered, the type of services considered (e.g. mobile or web), and the current context-of-use (e.g. different mobile services can be

discovered depending on their proximity to the device running the application). In our approach, services are only selected based on their similarity with the corresponding task. A concrete service model may also provide useful hints to the designer and developer, as some candidate services may not satisfy the selection criteria. This model is tightly coupled to the application's corresponding CUI. First, as they both reference each other's content. Second, as they are finally transformed to the application's code.

METHOD

The UI-specific path of the proposed design process was realized in our previous work [11]. The functional-specific reification path is realized by this paper's MDA approach, which automatically produces a concrete service model from a task model. Our approach's key characteristic is the use of semantics during the service-to-enriched service model transformation. In particular, a term-to-ontology concept matching algorithm is exploited to map a task's application objects to the concepts of a domain ontology, which is selected by the designer from a set of ontologies already used to annotate the service advertisements stored in our system's service repository. As a result, a semantic service query is produced from each system task description that is sent to a semantic service discovery engine, which in turn accurately produces a set of semantic service descriptions that are associated to the particular task.

The *User-Centered Service Discovery System* implements our approach and consists of the following components: *Controller*, *Transformer*, *Term Matcher*, and *Service Matchmaker*. The *Controller* coordinates the other components and interacts with a task model editor, thus enhancing the editor's capabilities and enabling it to model the service-based applications front-end. Next subsections analyze the rest of the components functionality.

Transformer

The *Transformer* performs four main transformations: a) task-to-service model, b) service model-to-service queries, c) service model and discovery results-to-enriched service model, and d) enriched-to-concrete service model, which are explained in detail in the following subsections.

Task-to-Service Model

The service-to-CTT task model transformation is performed by retaining only particular high-level and all system tasks of the task model and respecting its hierarchy and temporal semantics. System tasks are retained as they correspond to the system's functionality, while high-level tasks are retained only when they have system tasks as children. The latter can be justified by the following alternative cases. If the children system tasks correspond to service operations, then their parent, which represents a higher-level goal, should correspond to a service. Otherwise, if these system tasks correspond to services, then their parent should correspond to a more composite service.

The following set of assumptions should hold to guarantee the success of task-to-service model transformations and should be considered as guidelines during the design of task models. These assumptions are made as it cannot be determined in a CTT model which application objects manipulated by a system task are its input or output.

A1: The application objects manipulated by each task should be specified. **A2**: Interaction tasks manipulate application objects that can be considered as input to the remaining system tasks, if the latter tasks have equally-named objects. **A3**: A system task's application objects matching those of previous system tasks can be considered as its input. **A4**: A system task's unmatched application objects can be considered as its output. **A5**: If a system task modifies an application object (i.e. this object is both input and output), then this task's specification should contain a description of two equally-named objects, where the first object is considered as its input and the second as its output.

During a task model's transformation to a service model, its nodes are visited in a Depth First Search (DFS) fashion. Based on the node type, the following cases are considered:

Rule 1: Nodes representing user, interaction, or abstract tasks with no children are removed. Interaction tasks' application objects are kept in a Global Object List (GOL).

Rule 2: System tasks' tree position determines their renaming. Leaves are renamed as *operation* type tasks (**Rule 2.1**) and non-leaves as *service* (**Rule 2.2**). Their description and name are retained, while the following rules may fire when iterating over a leaf system task's objects: **Rule 2.3**: If an object has a name matching a name of a GOL object, then its node is renamed as *input*; **Rule 2.4**: If an object has an identically named sibling already matched, then its node is renamed as *output*; **Rule 2.5**: If an object's name is unmatched, then its node is renamed as *output*. Leaf system tasks' renamed objects are copied into GOL.

Rule 3: Abstract tasks with system task descendants are renamed as *service*.

Figure 3 depicts a transformation example of a task model's part into the service model's respective part. The task model's part concerns the procedure of sorting a physical addresses array. It contains one leaf interaction task, retrieving the addresses array and the sorting criterion from the user, followed by one leaf system task, sorting the addresses array according to the provided criterion. Both tasks contain a set of two identical objects (addresses array, sort criterion). Moreover, the system task contains two copies of the same object (the addresses array). Only the system task has been retained in the resulting service model, that is renamed to operation and contains four I/O objects: a) one input and output nodes created from its two equally named objects matching a particular interaction task object, b) one input node created from the sort criterion object, c) one output node created from its unmatched object (sorting steps number).

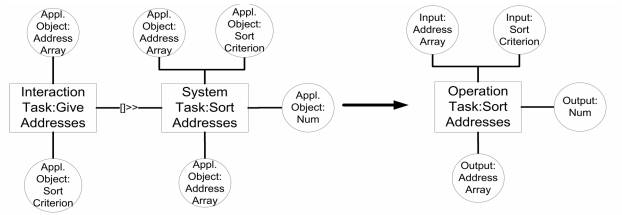

Figure 2: Objects-to-I/O parameters trans/tion example

There can be two alternative cases for removing a non-root node: **Rule 4.1**: When it is the leftmost or rightmost child of its parent, its right or left temporal operator has to be removed, respectively; **Rule 4.2**: Otherwise, its right operator has to be removed and its left sibling has to be connected with its right one using its left operator.

When a node is revisited, its number of children should be checked. If it is one, the node's child must replace it (**Rule 5.1**). If it is zero, the node must be removed (**Rule 5.2**).

Service Model-to-Service Queries

This transformation produces a service query for each service model's operation task. Each produced query has a service description with one operation, where the operation is named after the transformed task and the service is named after the task's parent, while the operation's I/O parameters are named after the task's I/O objects.

Service Model and Discovery Results-to-Enriched Service Model

This transformation merges the service queries discovery results to the service model, producing an enriched one. In particular, a service query's discovery results are enclosed within the corresponding operation task's description in various categories according to the results' semantic degree of match with the query. Each result is represented by the service's name, URI, WSDL URI, and rank (according to its structural similarity to the respective service query).

Enriched Service Model-to-Concrete Service Model

This transformation produces a concrete service model from an enriched one through service concretization. In particular, for each enriched service model's operation task the highest-ranked result of the best matching category is retained, while the remaining results are discarded. In this way, a one-to-one correspondence between an operation task and its service discovery result fulfilling it is achieved.

Term Matcher

The *Term Matcher* semantically enriches the service queries produced from a service model by mapping their I/O parameters to domain ontology concepts through the approach of [6], which calculates the relatedness between a parameter and an ontology concept by computing the parameter's name similarity both with this concept and its related ontology terms (its super-concepts). The Google distance metric [5], well-founded on information distance and Kolgomorov complexity, is exploited to assess the name similarity between terms, relying on the compared terms relative frequency of Web appearance calculated by invoking a Web search engine like Google.

Service Matchmaker

The enriched service queries are matched by the *Service Matchmaker* with the service advertisements stored in its registry. OWLS-MX [8] has been selected for realizing this component's functionality. It is a prominent hybrid service discovery engine using both SW and IR techniques to perform the matchmaking. Moreover, OWLS-MX is able not only to match semantic service descriptions but also to produce a categorization of the service discovery results and to rank them in each category based on their textual IR similarity with the service query. In fact, it is its ranking capabilities that enable our approach to transform an enriched service model to a concrete one. The reasons of selecting OWL-S as the semantic description formalism and not WSMO, influencing also the semantic matchmaker's selection (i.e. OWLS-MX vs. WSMX), are the following: a) there is a public available set of OWL-S advertisements called OWLS-TC, while there is no such set for WSMO, b) OWL-S has been used in automated planning approaches to produce a concrete plan for a high-level user goal or task specification while WSMO has not, c) there are few WSMO matchmakers in comparison to many OWL-S ones.

CASE STUDY

This section illustrates the proposed approach's functionality through an example application, which concerns the Web Shopping domain and involves ordering a car. Figure 3 depicts the hierarchy and temporal semantics of the application's CTT task model, while Table 1 shows the model's relevant task information.

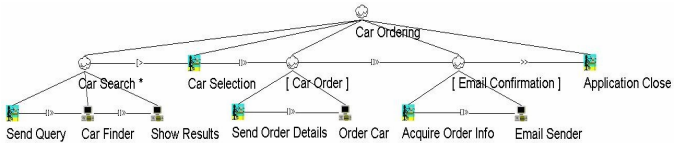

Figure 3: The car ordering application's task model

The task model's root abstract task is named "Car Ordering" and is decomposed into five sequentially executed tasks: "Car Search", "Car Selection", "Car Order", "Email Confirmation", and "Application Close".

The "Car Search" abstract task concerns the procedure of searching for the prices and colors of a car of a specific brand and model. It is decomposed into three sequential tasks: a) the interaction task "Send Query" that obtains the user's car search parameters, b) the system task "Car Finder" that retrieves the price and color lists matching the user query terms, and c) the system task "Show Results" that displays the car's discovered price and color lists.

The "Car Selection" interaction task interrupts the previous task and enables the selection of the desired car's specific

color and price. The "Car Order" optional abstract task follows representing the procedure of ordering the selected car, which is decomposed into two sequential tasks: a) the "Send Order Details" interaction task allowing the user to supply the ordering details (the car's model, brand, price and color and the user's name and credit card number) and b) the "Order Car" system task performing the car ordering.

The "Email Confirmation" abstract optional task concerns the procedure of sending a confirmation email to the user's email account and is decomposed into two sequential tasks: a) the "Acquire Order Info" interaction task enabling the user to request receiving the car order details, and b) the "Email Sender" system task that sends the requested email.

Table 1: Task information and transformation details

Appl. Object Name	In Interaction Task	In System Task (transformed object type)
Car Brand	Send Query	Car Finder (input)
Car Model	Send Order Details	Show Results (input)
		Order Car (input)
Price	Car Selection	Show Results (input)
Color	Send Order Details	Order Car (input)
		Car Finder (output)
Name	Send Order Details	Order Car (input)
CCN		
Order	Acquire Order Info	Email Sender (input)
		Order Car (output)
Email	Acquire Order Info	Email Sender (input)

The task model is sent to the *Transformer* that performs the task-to-service model transformation. The produced service model is depicted in Figure 4, where *service* tasks are represented by the *abstract* task symbol and *operation* tasks by the *system* task symbol. It is clear from Figure 4 that the task model's interactive tasks have been removed (Rule 1), while the abstract tasks (Car Order and Email Confirmation) owning just one system task child have been replaced by this child (Rule 5.1). In addition, leaf system tasks have been transformed to *operation* tasks (Rule 2.1), while abstract tasks (Car Search and Car Ordering) with more than one system task child have been transformed to *service* tasks (Rule 3).

Table 1 depicts which application objects were contained in particular interaction and system tasks and to what type of objects (input, output, or both) have been transformed to for the system tasks that contained them. As can be seen from this table, no application object has been transformed both to an input and output object for a particular system task. Thus, only Rules 2.3 and 2.5 were fired during the task-to-service model transformation.

The *Transformer* produces four OWL-S services queries from the service model, which correspond to the four service model's operation tasks. These service queries are enriched by the *Term Matcher* that maps their I/O parameters to concepts of the domain ontology ("my_ontology.owl"), which has been selected by the designer from the ontologies used to annotate the OWLS-TC service advertisements stored in the *Service Matchmaker*'s registry. Then, the enriched service queries are matched by the *Service Matchmaker* with its service advertisements and their discovery results are sent back to the *Transformer*, which merges them into the service model to finally produce the enriched service model.

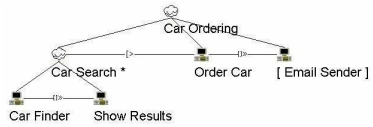

Figure 4: The task model's derived service model

```
<profile:Profile rdf:ID="_Car_Finder_Profile">
  <service:isPresentedBy rdf:resource="#_Car_Finder_Service"/>
  <profile:serviceName xml:lang="en">Car_Finder</profile:serviceName>
  ...
</profile:Profile>
  ...
<process:Input rdf:ID="_Car_Brand">
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
     http://127.0.0.1/ontology/my_ontology.owl#Car
  </process:parameterType>
</process:Input>
<process:Input rdf:ID="_Car_Model">
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
     http://127.0.0.1/ontology/my_ontology.owl#Car
  </process:parameterType>
</process:Input>
<process:Output rdf:ID="_Price">
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
     http://127.0.0.1/ontology/my_ontology.owl#Price
  </process:parameterType>
</process:Output>
<process:Output rdf:ID="_Color">
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
     http://127.0.0.1/ontology/my_ontology.owl#Color
  </process:parameterType>
</process:Output>
```

Figure 5: Snippet of the first enriched service query

Figure 5 shows a snippet of the first enriched service query corresponding to the "Car Finder" system task. As can be seen, the task's first two input objects (Car Brand and Car Model) have been mapped to the same ontology concept (Car). This is because the domain ontology contains only the Car concept and not the Car Brand and Car Model ones. In fact, the Car concept has datatype properties named *brand* and *model*, so by mapping the first two objects onto the same concept it is assured that they are both represented in the concept's description. In addition, the last two task's objects (Price and Color), corresponding to its output, have been mapped onto different concepts (Price and Color).

The first query's discovery results are shown in a specific window of the task model editor (when the "Car Finder" task is selected) depicted in Figure 6. As can be seen, three result categories have been produced with their results sorted according to their textual similarity with the service query. The first category is the best. It corresponds to an exact match and contains one service advertisement. By selecting this advertisement, its file name and I/O concepts are shown in the window's right bottom part. Figure 7 shows a snippet of the advertisement, where the service's I/O parameters names and their corresponding ontology

concepts can be viewed. The latter concepts are the same as those of the service query and correspond to the same parameter type (i.e. I/O), thereby yielding an exact match.

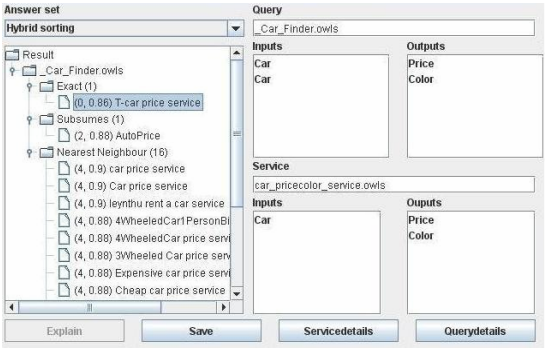

Figure 6: First query's discovery results

```
<profile:Profile rdf:ID="CAR_PRICECOLOR_PROFILE">
<service:isPresentedBy rdf:resource="#CAR_PRICECOLOR_SERVICE"/>
<profile:serviceName xml:lang="en">T-car price service</profile:serviceName>
  ...
</profile:Profile>
  ...
<process:Input rdf:ID="_CAR">
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://127.0.0.1/ontology/my_ontology.owl#Car
  </process:parameterType>
</process:Input>
<process:Output  rdf:ID="_PRICE">
 <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
   http://127.0.0.1/ontology/my_ontology.owl#Price
</process:parameterType>
</process:Output>
<process:Output  rdf:ID="_COLOR">
 <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
   http://127.0.0.1/ontology/my_ontology.owl#Color
</process:parameterType>
</process:Output>
```

Figure 7 - Snippet of the matched service advertisement

CONCLUSIONS

This paper presents an MDA approach providing automatic support to the interactive service-based applications design by identifying those services that best support system tasks. This approach automatically produces a concrete service model from a designer-provided task model and domain ontology that specifies the way services can be combined to realize the application's functionality. The concrete and enriched service models provide support to the application's designer and developer: a) they indicate missing functionality to be implemented and the functionality realized by existing services, thus reducing the application development time; b) they determine specific places where UI changes must be performed. Our approach can be integrated in model-based HCI approaches to obtain a complete authoring environment for interactive service-based applications. We plan to perform this integration in the near future.

ACKNOWLEDGEMENTS

This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme and was supported by the ServFace (http://www.servface.eu) ICT EU project.

REFERENCES

1. V. Alevizou and D. Plexousakis. Enhanced specifications for web service composition. In *ECOWS*, pages 223–232, Zurich, Switzerland, 2006. IEEE Computer Society.
2. G. Broll, S. Siorpaes, M. Paolucci, E. Rukzio, J. Hamard, M. Wagner, and A. Schmidt. Supporting Mobile Service Interaction through Semantic Service Description Annotation and Automatic Interface Generation. In *SemDesk at ISWC*, Athens, GA, USA, 2006.
3. G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A Unifying Reference Framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, June 2003.
4. F. Casati, S. Ilnicki, L.-j. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and dynamic service composition in eflow. In *CAiSE*, pages 13–31, Stockholm, Sweden, 2000. Springer-Verlag.
5. R. L. Cilibrasi and P. M. B. Vitanyi. The Google Similarity Distance. *IEEE Trans. on Knowl. and Data Eng.*, 19(3):370–383, 2007.
6. J. Gracia and E. Mena. Web-Based Measure of Semantic Relatedness. In *WISE*, pages 136–150, Auckland, New Zealand, 2008. Springer-Verlag.
7. D. Khushraj and O. Lassila. Ontological Approach to Generating Personalized User Interfaces for Web Services. In *ISWC*, volume 3729 of *LNCS*, pages 916–927, Galway, Ireland, 2005. Springer.
8. M. Klusch, B. Fries, and K. Sycara. OWLS-MX: A hybrid Semantic Web service matchmaker for OWL-S services. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2):121 – 133, 2009.
9. N. Partarakis, C. Doulgeraki, A. Leonidis, M. Antona, and C. Stephanidis. User interface adaptation of web-based services on the semantic web. In *UAHCI*, pages 711–719, San Diego, CA, 2009. Springer-Verlag.
10. F. Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, 1999.
11. F. Paternò, C. Santoro, and L. D. Spano. Support for authoring service front-ends. In *EICS*, pages 85–90, Pittsburgh, PA, USA, 2009. ACM.
12. P. Plebani and B. Pernici. URBE: Web Service Retrieval Based on Similarity Evaluation. *IEEE Trans. on Knowl. and Data Eng.*, 21(11):1629–1642, 2009.
13. M. Ruiz, V. Pelechano, and O. Pastor. Designing Web Services for Supporting User Tasks: A Model Driven Approach. In *ER Workshops*, volume 4231 of *LNCS*, pages 193–202, Tucson, AZ, USA, 2006. Springer.
14. J. Vermeulen, Y. Vandriessche, T. Clerckx, K. Luyten, and K. Coninx. Service-Interaction Descriptions: Augmenting Services with User Interface Models. In *EIS*, pages 447–464, Salamanca, Spain, 2007. Springer-Verlag.
15. D. Wu, B. Parsia, E. Sirin, J. A. Hendler, and D. S. Nau. Automating DAML-S Web Services Composition Using SHOP2. In *ISWC*, volume 2870 of *LNCS*, pages 195–210, Sanibel Island, FL, USA, 2003. Springer.