Task Model-Driven Realization of Interactive Application Functionality through Services

K. KRITIKOS and D. PLEXOUSAKIS, IS Lab, ICS-FORTH, Heraklion, Greece
F. PATERNÒ, HIIS Lab, ISTI-CNR, Pisa, Italy

The Service-Oriented Computing (SOC) paradigm is currently being adopted by many developers, as it promises the construction of applications through reuse of existing Web Services (WSs). However, current SOC tools produce applications that interact with users in a limited way. This limitation is overcome by model-based Human-Computer Interaction (HCI) approaches that support the development of applications whose functionality is realized with WSs and whose User Interface (UI) is adapted to the user's context. Typically, such approaches do not consider various functional issues, such as the applications' semantics and their syntactic robustness in terms of the WSs selected to implement their functionality and the automation of the service discovery and selection processes. To this end, we propose a model-driven design method for interactive service-based applications that is able to consider the functional issues and their implications for the UI. This method is realized by a semiautomatic environment that can be integrated into current model-based HCI tools to complete the development of interactive service front-ends. The proposed method takes as input an HCI task model, which includes the user's view of the interactive system, and produces a concrete service model that describes how existing services can be combined to realize the application's functionality. To achieve its goal, our method first transforms system tasks into semantic service queries by mapping the task objects onto domain ontology concepts; then it sends each resulting query to a semantic service engine so as to discover the corresponding services. In the end, only one service from those associated with a system task is selected, through the execution of a novel service concretization algorithm that ensures message compatibility between the selected services.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*User-centered design*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

General Terms: Design, Algorithms

Additional Key Words and Phrases: Service front-ends, service discovery, interactive application design

ACM Reference Format:
K. Kritikos, D. Plexousakis, and F. Paternò. 2014. Task model-driven realization of interactive application functionality through services. ACM Trans. Interact. Intell. Syst. 3, 4, Article 25 (January 2014), 31 pages.
DOI: http://dx.doi.org/10.1145/2559979

1. INTRODUCTION

Web Services (WSs) are modular, self-describing software entities that can be discovered and invoked through the Internet. One of their most significant advantages is that they can be combined into new services offering new integrated functionalities. In fact, through service combination and composition, a whole application can be built

Authors' addresses: K. Kritikos and D. Plexousakis, IS Lab, ICS-FORTH, N. Plastira 100, Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece; email: kritikos,dp@ics.forth.gr; F. Paternò, HIIS Lab, ISTI-CNR, Via G.Moruzzi 1, 56124 Pisa, Italy; email: Fabio.Paterno@isti.cnr.it.

DOI: http://dx.doi.org/10.1145/2559979

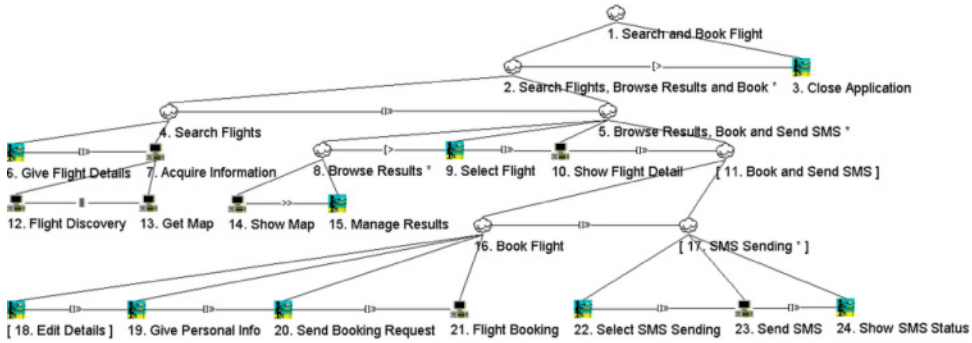

Fig. 1. The task model of the flight-booking application.

from scratch, or existing intra- or interorganizational applications can be integrated together in a seamless and loosely coupled way.

However, before services are combined, they first have to be discovered. Many research approaches have been proposed in service discovery using techniques from various disciplines, including Information Retrieval (IR) and Semantic Web (SW); such approaches are characterized by different performance and accuracy capabilities. The most accurate approaches use semantics taken from application domain ontologies in the service description. In particular, the service operations' input-output (I/O) parameters are mapped to ontology concepts and then reasoning mechanisms are used to infer the service request-to-advertisement similarity. However, these approaches rely on the requesters' ability to provide a semantic service description by assuming that requesters are familiar with SW technology and have a good knowledge of the application domain.

Concerning service composition, the proposed approaches produce a service composition plan from functional and nonfunctional user goals or requirements. Some of these approaches can automatically produce service plans but rely on service specifications, describing additional service features (e.g., pre- and postconditions), that are not always present in practice, while others assume the availability of an abstract service plan and just perform service selection based on the user goals for the plan tasks.

As a running example used throughout the article, suppose that the interactive flight-booking application depicted in Figure 1 must be designed in which a user can search for flights, browse the search results, and book the respective flights before closing it. The search for flights is performed by supplying the flight details and then acquiring the related information, including the relevant maps for the area between the initial and final destination. The search results browsing is performed by repeatedly visualizing the map and managing the results, selecting the flight, and showing the flight details. The booking of a flight is finally performed by providing appropriate details and sending the booking result with an SMS. Based on this application, the designer should exploit suitable service discovery and composition tools to discover the appropriate services (e.g., flight search services) and select the best service for each required functionality.

Unfortunately, the applications built from such service tools usually neglect users and their requirements. These applications usually interact only with automated programs or agents. In some cases, they can also interact with human users but in a limited way, as their User Interfaces (UIs) may not be applicable to the user platform, redundant input must be provided by the user, or users with limited capabilities are not considered. Thus, the resulting application turns out to be of limited usability.

Another disadvantage is that the produced application may not be *robust* [Lécué et al. 2009; Pires et al. 2003] due to lack of compatibility checks between the selected services, so it is not guaranteed to function correctly. In particular, a concrete service plan may contain two services exchanging semantically equivalent but syntactically incompatible information; incompatibility means that the information message type that one service receives is richer or totally different from that of the service producing it. For instance, concerning the running example, the functionality of retrieving and visualizing a map could correspond to two different services that produce and consume maps of a different format, respectively. Such a problem can be partially solved by enriching the domain ontology with a different concept to represent the richer message type (that is subsumed by the poorer message type's concept). The remaining problem is that a semantically equivalent but syntactically different representation of the same concept may lead to precision loss for some message types, while message adapters must be developed to have the application run properly.

Human-Computer Interaction (HCI) models can play an important role [Fonseca 2010; Vanderdonckt 2005] in better supporting the interactive applications design. The proposals adopting such models often follow a Model-Driven Architecture (MDA) [Calleros et al. 2010] starting with a description of a task model and ending with the final UI code generation. Such an approach is generic and assists in deriving UIs that adapt to the user device peculiarities. Moreover, it also considers that some system tasks may be associated with external services. However, its biggest disadvantage is the lack of automatic support concerning functionality as the designer has to manually select the appropriate services for particular system tasks (e.g., tasks with IDs 12–14 in Figure 1). Besides, when system tasks cannot be totally realized by existing services, the developer usually produces the respective code without reusing an imperfectly matching service's functionality. For instance, concerning the running example, if there is an internal service implemented for booking other items, such as books, then such service could be modified to realize the functionality of booking a flight (see task 21). Such activities require a great deal of effort and particular knowledge of both the domain entities and the services that can be associated to the application tasks/domain operations. Another weakness is the inability to exploit the domain knowledge associated with their application.

Thus, a new solution is needed inheriting the main features of the aforementioned HCI MDA. This solution should better consider the interactive application's functional design and should produce robust service compositions supporting the application's functionality. It must also be as automated as possible and sufficiently reduce the effort required by application designers and developers. To this end, this solution should rely on the use of semantics and the reasoning mechanisms that support it. It should also intelligently manage the domain knowledge produced during the application design to reuse the solutions found.

This article fills this gap by proposing a novel method for model-based service-oriented interactive application design, which focuses on functionality aspects and considers their implications on the UI. It also proposes a semiautomated environment that realizes the MDA's functional part by respecting the dependencies between the UI and functional models, taking as input an interactive application task model and producing as output a *concrete service model*. The latter model contains additional information useful to obtain robust interactive applications with some application logic implemented through WSs.

As a task model describes an abstract service plan, the proposed method achieves its aim by performing the usual steps of service discovery and concretization. However, its main advantage over related work is that apart from exploiting state-of-the-art semantic service matchmakers with optimal performance and accuracy, it produces

semantically and syntactically robust service compositions, so that the application is correctly executed without any information loss. The robust service compositions are produced via *a novel service concretization algorithm,* able to select the *best* matching service for each system task while minimizing the information loss that may occur between one service's rich output message and another service's respective poorer input.

The service matchmaking process has efficient accuracy, as the method relies on automatically discovering the formal semantics of application object terms by selecting an appropriate domain ontology from an ontology set already used to annotate the service advertisements stored in our environment's repository and mapping the task-manipulated objects to domain ontology concepts. In this way, the domain-specific terminology in describing both tasks and services is fixed, thus enabling their precise matching. As such, semantic system tasks map to application domain operations, while advertisements map to services realizing such operations. The designer is also not required to select the domain ontology or its concepts to properly annotate the task model's application objects.

The proposed solution can be used in model-based HCI approaches as a semiautomated component to replace those functionally equivalent parts requiring manual intervention and provide additional support to the interactive application design process. This support regards three aspects: first, discovering those services relevant for end-users that realize the system tasks' functionality; second, selecting one service from those associated to a system task to produce a robust concrete service model; and third, identifying those tasks that can be either partially or not at all supported by existing services. As such, the developer is informed about which system tasks must be implemented from scratch and which can be realized by also exploiting the tasks' partially matched services.

The article is structured as follows. Section 2 reviews related work. Section 3 provides useful background information for proper understanding of the article. Section 4 analyzes an MDA for interactive service-based application design and the respective functional models. Section 5 provides an overview of the proposed method and a specific use case of its application. Section 6 analyzes the main method steps. Section 7 presents the empirical evaluation performed on the system's components. Finally, Section 8 discusses our method's main advantages and disadvantages and draws further research directions.

2. RELATED WORK

Service Discovery. A service description is produced by using a particular service specification language (e.g., OWL-S). The most common service description parts in different languages are the service name, I/O parameters, and textual description. Other parts allowed by particular languages are the service operations, preconditions and effects, and process models. Service discovery is the process of finding a set of related service descriptions (i.e., service advertisements) published by service providers in a repository based on a service requester-specified service description (i.e., service request). An initial approach attributed to service discovery was proposed by Zaremski and Wing [1997], matching components in a software library by using signature and specification matching. The path pursued by the following approaches, such as Stroulia and Wang [2005], was to compare service requests and advertisements based on their structure and the similarity of the words contained in the pure text description of the service or symbolizing the names of the service and its I/O parameters and operations (e.g., by using thesauri-enhanced VSM techniques). However, those approaches' results were not so accurate, as the terms' semantics were not formally captured. As such, new approaches were proposed, first mapping the service description terms to either ad hoc concepts [Dong et al. 2004] or domain ontology concepts [Klusch et al. 2009; Plebani and Pernici 2009] and then using IR or both IR and SW techniques to assess the match

between service descriptions. Their accuracy has been proven to be increased. Thus, we selected the second type of the latter approaches to discover those services able to fulfill an application task's functionality.

As there can be services that match based on their I/O parameter concepts but expose a different functionality, the latter approaches' accuracy cannot be ideal. Only if services are additionally described with preconditions and effects can the service discovery accuracy become nearly perfect. As such, several full IOPE (I/O, preconditions, and effects) approaches have been proposed [Sycara et al. 2002; Keller et al. 2004], which cannot be exploited as the preconditions and effects of semantic services are not usually described.

Service Composition. Functional service composition approaches produce a specific concrete composition plan by applying techniques of various fields, such as AI planning [Pistore et al. 2005] and model-driven composition [Brogi et al. 2008]. Our work could exploit the latter approach to return service compositions for failed task/service queries that are not satisfied by any single service based on the requested I/O. For instance, concerning the running example, the functionality corresponding to task 12 might not be available if the user provides as input the names of (arrival and departure) cities and not the names of the respective airports. Thus, this would require the discovery of another service that can provide the name of the airport for a particular city in addition to the service that discovers available flights based on the arrival and departure airports. Another approach [Lau and Mylopoulos 2004] proposes a refined Tropos [Bresciani et al. 2004] design process exploiting Actor and Goal diagrams describing the stakeholders' needs and capabilities to produce a set of OWL-S queries, which are used to discover services that satisfy the stakeholder's goals.

The nonfunctional service composition work assumes that an abstract plan exists and performs service selection to instantiate it based on global nonfunctional constraints. It uses optimization techniques (e.g., Integer Programming [Zeng et al. 2004; Ardagna and Pernici 2007] or Genetic Algorithms [Canfora et al. 2008]) to guarantee the global constraints' fulfillment. However, it is conservative or inaccurate by considering the worst- or most-common-case service composition scenario [Canfora et al. 2008] or satisfies constraints statistically [Jaeger et al. 2005]. It cannot express nonlinear constraints and handle overconstrained user requirements; it requires that all execution paths must satisfy the global constraints—even the improbable ones—and considers only worst or average service metric values and the independence of metrics. Ferreira et al. [2009] have proposed a novel approach that solves above problems.

Interactive Service-Based Application Design. Various HCI approaches have been proposed in interactive service-based application design. Vermeulen et al. [2007] propose a model-based UI development approach, where the designer first creates a hierarchical task model based on the ConcurTaskTrees (CTT) [Paternò 1999] notation and manually links its leaf tasks to concrete service components and abstract interaction objects. Khushraj and Lassila [2005] enhance OWL-S service descriptions with UI annotations and automatically transform them into form-based UIs used both to invoke the desired services and to display the results.

Broll et al. [2006] propose to enrich semantic WS descriptions with UI annotations for the automatic generation of adapted UIs to support physical mobile interaction. Ruiz et al. [2006] perform transformations on an HCI task model to design a particular service and service discovery to find services with operations matching those of the designed service. This work manually maps service operation I/O parameters to domain entities.

Sasajima et al. [2008] propose using task-oriented menus for service navigation. A task ontology method is used to build these menus, supporting the description of user activities, services associated to such activities, obstacles occurring before or during user activity execution, and ways to solve them. In this work, very simple UIs are

Table I. Comparison with Related Work

Research work	Features				
	Automatic service discovery	Domain knowledge reuse	Service query enr/ent	Robust service selection	UI aspects cons.
Zaremski and Wing	√				
Stroulia and Wang	√				
Dong et al.	√		∼		
Klusch et al.	√	∼			
Plebani and Pernici	√	∼			
Brogi et al.	√	∼		∼	
Sycara et al.	√	∼			
Keller et al.	√	∼			
Elgedaway et al.	√	∼		∼	
Pistore et al.	√	∼		∼	
Brogi et al.	√	∼		∼	
Lau and Mylopoulos	√	∼		∼	
Zeng et al.	√			∼	
Ardagna and Pernici	√			∼	
Canfora et al.	√			∼	
Jaeger et al.	√			∼	
Ferreira at al.	√			∼	
Vermeulen et al.		∼			√
Khushraj and Lassila		∼			√
Broll et al.		∼			√
Ruiz et al.	√				√
Sasajima et al.		∼			∼
Manolescu et al.					√
USDS	√	√	√	√	√

Note: The table symbols are interpreted as follows: √ and ∼ mean that the work fully and partially satisfies the feature, respectively, while a blank entry means that the work does not exhibit the feature. The relevant references are included in the bibliography section.

built, service providers must annotate their service descriptions with the tasks they fulfill, and users must navigate to find the exact activity to perform.

Manolescu et al. [2005] extend the WebML declarative model to specify data-intensive web applications, which implement new, complex WSs from existing ones.

The previous HCI approaches support specific interactive application types or do not exhibit the appropriate automation level, especially in terms of automatically selecting services able to fulfill the application functionality. As such, a generic design approach for interactive applications is needed to provide automatic support to the designer. Our work follows this direction by automatically discovering services for system functionalities indicated by a task model without losing the generality offered by model-driven approaches. It can replace the manual, functionality-oriented parts of HCI approaches so that the resulting system provides additional automatic support to the designer. Finally, it produces only robust concrete service models, a novelty not offered by other approaches.

Comparison. Table I compares our solution/system, named the *User-Centered Service Discovery System* (USDS), with the analyzed related work based on the following criteria:

- *Automatic service discovery*: the ability of automatically discovering a set of service advertisements that match a specific service query given as input.

- *Reuse of domain knowledge*: the ability to reuse the domain knowledge dynamically produced. The use of static knowledge, such as domain ontologies, is considered as partial criterion satisfaction, while the exploitation of dynamic knowledge, such as the application object-to-ontology concept mappings, is regarded as full satisfaction. The reuse of domain ontologies leads to increased accuracy, as both system tasks and services can be functionally described in the same way. The exploitation of dynamic knowledge leads to shorter execution time; as such, knowledge is not required to be reproduced for each new case of application design. For instance, concerning the running example, if the user uses the same name for identical application objects (e.g., Flight List) of different tasks (e.g., tasks 12 and 14), then such objects could be mapped to the same domain ontology concept (e.g., ListOfFlights), where the mapping could be performed just once for the first occurrence of the object, thus enabling the fast annotation of application tasks.
- *Service query enrichment*: the ability to semantically enrich the user input (e.g., service queries or task models) so as to provide more accurate service discovery results through the use of a common vocabulary in service query and advertisement specifications. This criterion is fully satisfied if the appropriate domain ontology is selected automatically and a term-to-concept mapping algorithm is exploited. If only one from the previously mentioned two cases holds, then the work partially satisfies the criterion. Continuing the previous example, by mapping the input object of task 14 to a specific ontology concept (i.e., ListOfFlights), the service discovery accuracy is increased by discovering services (annotated with the same concept) that not only show maps but also visualize in them the list of flights that have been discovered.
- *Robust service selection*: the ability to perform service selection such that the resulting service plan is semantically robust. Approaches just performing service selection or considering the plan's syntactic robustness partially satisfy this criterion.
- *UI aspects consideration*: this criterion considers if UI-related information is considered to produce an interactive application based on the current context of use. Approaches just producing UIs with a limited usability (e.g., for the average user) without considering the current context of use partially satisfy the criterion.

Table I shows that our work fills a gap in the state of art by addressing the indicated set of aspects. So, our work provides crucial support in designing service-based interactive applications. It can be categorized as multidisciplinary as it aims to solve a research problem that is multidisciplinary in nature and provides novel solutions by exploiting techniques from various fields, such as HCI and IR. As Table I shows, so far there has been little overlap between such communities but the increasing need for making Internet services accessible to a wide variety of users is stimulating new research in order to integrate them. In our case, we use task models to better support service discovery and selection for the functional part of interactive applications. This allows designers and developers to obtain services that are relevant and useful for the end-users and that can be better integrated in a development process based on user-oriented models.

This work substantially extends and further improves our previous work [Kritikos and Paternò 2010a, 2010b] as follows: (a) the term-to-ontology mapping algorithm is further extended, (b) the term-to-ontology concept associations are reused, (c) service selection was performed by selecting the service having the greatest semantic and structural similarity with the service query while now the novel service concretization algorithm is used, (d) the proposed interactive application design method is substantially extended and considers the dependencies between the UI and functional models and the various design issues involved, and (e) the proposed method is experimentally evaluated.

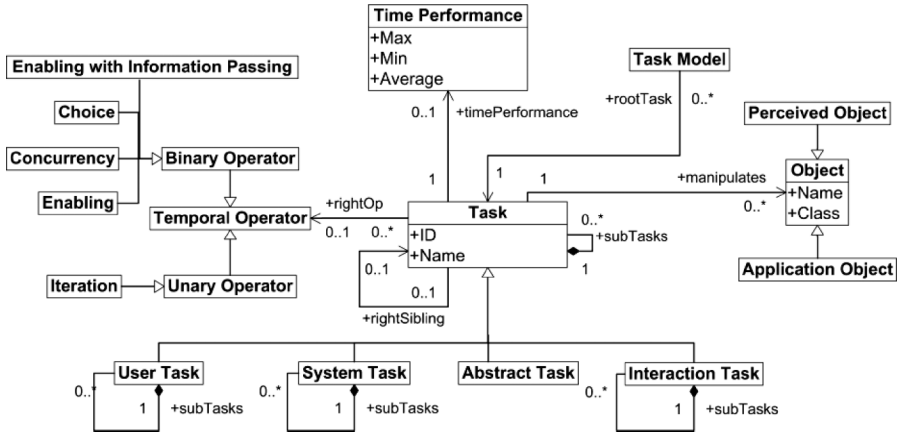

Fig. 2. The class diagram of the CTT task model.

3. BACKGROUND

This section provides background knowledge useful for proper understanding of the article. In particular, Section 3.1 considers task models (i.e., the initial models exploited by our method) and introduces the concepts of the task model representation language used. Section 3.2 describes the Semantic Service Matchmaking Engine exploited to match the produced service queries with the service advertisements stored in our system repository.

3.1. Task Models

Due to the plethora of platforms and interactive devices, the UI design can be supported through various abstraction levels so as to focus on the semantic aspects and not on the low-level details associated with the respective implementation languages. To this end, at the highest level, task models are adopted as a way of describing application functionality and interaction with one or more users in terms of activities required to reach the users' goals. They provide a user view of the interactive application and a description more focused on user interactions than approaches such as BPMN. As such, they are exploited by service front-end authoring environments [Paternò et al. 2011] to produce the final executable code of the service-based interactive application, including the final UI (FUI).

Task models usually contain a hierarchical task description and a temporal operators set specifying the task execution order. Various notations and associated tools were proposed to represent and reason about task models. Our work exploits the CTT notation, which has been widely used[1] in many applications (e.g., adaptable web museum applications, air traffic control, serious games, etc.) and is supported by a toolset [Mori et al. 2002] that includes an editor providing advanced editing and analysis features. Figure 2 shows a UML class diagram representing the main CTT task meta-model entities and their relations.

A task model is associated to one root task. Tasks are classified into four categories: *abstract*, *user*, *interaction*, and *system*. Abstract tasks represent the most general task category and comprise subtasks that may belong to different categories. In the other task categories, a task can only have subtasks of the same category. User tasks are

[1]List of organizations that have used CTTE at http://giove.isti.cnr.it/tools/CTTE/external.

internal cognitive activities performed by the end-user without interacting with the application (e.g., taking a decision). Interaction tasks represent interaction activities between the application and end-user. System tasks are performed solely by the application. Each task is primarily characterized by an identifier, name, and category. It is also associated with a set of objects it manipulates. Each object is primarily characterized by its name and class (such as *string*, *text*, or *object*). Objects can be classified as *Perceivable* or *Application* objects, which are related to the UI or the application domain entities, respectively.

Temporal operators are unary or binary. The most usual binary operators are *choice* (one of the tasks is selected and executed), *enabling* (the tasks are sequentially executed with no information exchange), *enabling with information passing* (the tasks are sequentially executed and the first provides information to the second), and *concurrency* (the tasks are performed concurrently). The most usual unary operator is *iteration* (indicates a task's repetition). As the CTT Editor can save a task model in XML format, this model may be processed and transformed into different contents possibly expressed in different formats.

3.2. Semantic Service Matchmaking Algorithm

The OWLS-MX [Klusch et al. 2009] prominent Semantic Service Matchmaking Engine (SSME) is exploited in our system to match an OWL-S [Martin et al. 2004] service descriptions repository with an OWL-S service query and to produce results with high accuracy (precision and recall [Baeza-Yates and Ribeiro-Neto 1999]). Its main advantage is that it produces not only an extensive categorization of the discovery results but also their ranking in each category based on their textual IR similarity with the service query.

OWLS-MX produces the following five types of matchmaking results in decreasing order of significance for a given OWL-S query:

1. **Exact**: Contains those services that perfectly match the query.
2. **Plug-in**: A service is a plug-in match if each of its input parameter concepts matches a respective query input parameter concept and if each query output parameter concept matches a respective service output parameter concept. One concept matches a second concept if it subsumes or is equal to the second one. In this category, at least one matching concept pair must exist with the first concept subsuming the second.
3. **Subsumes**: The difference with respect to the previous category is that the concept of at least one service output parameter must be a nondirect subclass of the concept of the corresponding query output parameter.
4. **Subsumed-by:** A service is a subsumed-by match if each of its input parameter concepts matches a respective query input parameter concept and if each query output parameter concept is matched by a respective service parameter output concept while its similarity with respect to the service query is above a certain threshold. This similarity concerns the I/O parameters and can be computed according to different VSM-based similarity metrics.
5. **Nearest-neighbor**: Contains those services not matching the aforementioned cases, whose similarity with the query is above a specific threshold.

4. INTERACTIVE SERVICE-BASED APPLICATION DESIGN

This section presents our flexible MDA for interactive application design. It also defines a new and more appropriate service model for interactive service-based applications compared to traditional service models.

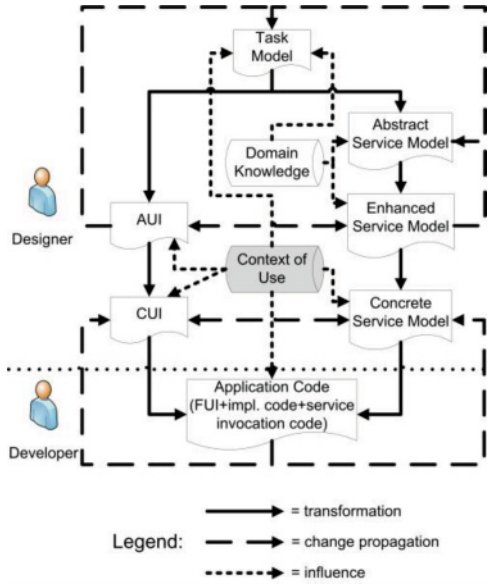

Fig. 3. Integrated design of a service-based application.

4.1. MDA for Designing Interactive Service-Based Applications

Most applications are both interactive and service based so their design should address equally the UI and functionality aspects and their implications. For this purpose, the design process should be split into two parallel paths according to these two aspects.

A task model conveys important information to be used as a starting point for the parallel design of an application's UI and functionality, as it describes all application interactive and system tasks and their temporal order and interactions. Interactive tasks correspond more closely to the UI aspect, while system tasks correspond to the functionality one. As such, by following an MDA, the task model can be transformed into two different aspect-oriented parts: an abstract UI (AUI) (UI aspect) and an abstract service model (functionality one). These two parts can be processed in parallel but not independently based on the target platforms and the domain knowledge and eventually be joined together to produce the application code.

Figure 3 shows the MDA to an interactive service-based application design that considers both the functional and UI aspects, which has been proposed in our previous work [Kritikos and Paternò 2010a]. It also shows which factors influence the design of which model and where the responsibilities of the application designer and developer lie. As our focus is on the functional design path, this path is analyzed in the following along with a thorough analysis of the functional models and their dependencies with the UI models. The interested reader can refer to (Online) Appendix A for a more detailed analysis.

4.2. Service Model

A transformation of an HCI task model (see Section 6.1), which provides a high-level view of the application functionality, generates the service model by considering only the system tasks and particular abstract tasks. The rationale of selecting these task types is as follows. Leaf system tasks of the same parent have complementary and interrelated functionalities whose combination leads to achieving a specific functional goal. Thus, these leaf system tasks should map to a subset, if not all, of the operations

$$service\_model(modelID, name, type, \underline{task}) \quad (D.1)$$

$$task(taskID, \underline{\underline{name}}, \underline{\underline{description}}?, type, iterative, optional, \underline{parentID}?, \underline{rightID}?, \underline{temporalOperator}?, \underline{input}*, \underline{output}*, \underline{results}*, \underline{TimePerformance}, \underline{task}*) \quad (D.2)$$

$$input(appl\_object) \quad (D.3)$$

$$output(appl\_object) \quad (D.4)$$

$$appl\_object(OID?, appl\_name?, concept?, property?, ref?) \quad (D.5)$$

$$results(\underline{exact}?, \underline{plugin}?, \underline{subsumes}?) \quad (D.6)$$

$$exact(\underline{service}^{+}) \quad (D.7)$$

$$plugin(\underline{service}^{+}) \quad (D.8)$$

$$subsumes(\underline{service}^{+}) \quad (D.9)$$

$$service(URI, name, \underline{inputParam}*, \underline{outputParam}*, score) \quad (D.10)$$

$$inputParam(name, concept, OID, \underline{MsgType}) \quad (D.11)$$

$$outputParam(name, concept, OID, \underline{MsgType}) \quad (D.12)$$

$$MsgType(MID?, name?, elemName?, simple, type?, ref?, \underline{MsgType}*) \quad (D.13)$$

$$TimePerformance(Max?, Min?, Average?) \quad (D.14)$$

Fig. 4. The service model.

of an existing WS, while their parent should map to the actual WS. As such, high-level system and abstract tasks, with system tasks as children, map to existing single or composite WSs.

Based on this analysis, the service model has a representation similar to the one of an HCI task model with particular additions addressing the representation of services fulfilling the functionality of the selected abstract and system tasks. This representation is XML based and conforms to a specific language definition, which is shown in Figure 4 with some symbols altered or omitted for the sake of readability and brevity. In particular, in an element definition, the symbol "?" represents a single optional occurrence of a specific element or attribute, while symbols "$*$" and "$+$" represent zero or more and one or more multiple optional occurrences of a specific element, respectively. Single underlined names represent elements and double-underlined names represent textual elements.

The service model consists of 14 element definitions. To follow, each element is shortly analyzed by explaining the most relevant definition aspects (e.g., by omitting referencing element IDs and names) and omitting some aspects that are common with the CTT model. The interested reader can refer to Appendix A for a more detailed analysis:

- D.1 specifies the *service_model* element. This element has the *type* attribute taking the values {*abstract*, *enhanced*, *concrete*} and contains the root *task* element.
- D.2 corresponds to the *task* element. This element has the *type* attribute that can take one among the {*operation*, *service*} values. It contains the next elements: *description* textually describes the task functionality; *parentID* and *rightID* determine the task's parent and singling node identifiers, respectively; *temporalOperator* specifies the CTT temporal operator connecting the task with its right sibling; *input* and *output* map to the task input and output application objects; *results* provides the task service discovery results; and *task* specifies the task children recursively.
- Definitions D.3–4 are identical but correspond to the *input* and *output* elements, respectively, defined with respect to the *appl_object* element. The latter concerns an application object, defined (see D.5) by the URIs of the *concept* and *property* to which it may be mapped, or by a reference *ref* to another application object's ID.
- D.6 specifies the task-based service discovery results. It contains *exact*, *plugin*, and *subsumes* results depending on their semantic type. The latter result types are

determined by the next three (D.7–9) identical element definitions containing the service advertisements results specifications represented by the *service* element.
- A service element is specified by D.10. It contains the *URI* of its OWL-S semantic description, its *inputParam* and *outputParam* elements, and *score* defining its overall similarity with its matching task.
- D.11–12 identically specify the *inputParam* and *outputParam* elements, which contain the URI of the *concept* they have been annotated with, the ID (*OID*) of the application object they have been matched with, and their message types (*MsgType*) as defined in the corresponding service WSDL file.
- D.13 specifies the *MsgType* element containing as attributes the name (*elemName*) of its XSD element definition, the *simple* attribute determining if this type is simple or complex, and the basic XML *type* for simple message types, or a reference *ref* to another message type. If it is complex, it can contain a set of enclosing *MsgType*s.

Based on definition D.1, three service model types exist. *Abstract service models* contain tasks not yet associated to existing services in contrast to the other two types. So, they represent an interactive application's functionality at the highest abstraction level. They resemble abstract service plans, but their main difference is that they are hierarchical and represent the different ways an application's functionality can be realized by executing particular tasks. So, an abstract service model corresponds to many abstract service plans. As these models are semantic, their application objects should be mapped onto domain ontology concepts by using a name-to-concept mapping algorithm. Thus, their generation from a task model involves various steps, which are analyzed in Sections 6.1–6.3.

Enhanced service models contain tasks that have been associated to existing services by transforming these tasks to service queries and exploiting an SSME to perform the matchmaking and discover their corresponding associations. These models encapsulate all appropriate information to perform service concretization: (a) all possible (abstract and concrete) service plans, (b) association of tasks to services realizing them, and (c) temporal task ordering. So, a service concretization process exploiting an enhanced service model could select the best abstract plan among those available and the best services realizing the plan tasks' functionality (see Kritikos and Kubicki [2011]). However, based on the pragmatics of the services world, only the leaf model nodes can be associated to existing services, as the discovery of the higher-level nodes' associated services would require describing the temporal semantics of the service operations. Thus, an enhanced service model corresponds to only one abstract (and many concrete) service plan, comprising the service model's leaf tasks, their temporal dependencies, and their associated services.

The abstract-to-enhanced service model transformation involves various intermediate steps, analyzed in Sections 6.4–6.6, while its overall procedure is analyzed in Section 5.

Concrete service models specify how an interactive application's functionality can be fulfilled by indicating which services are selected to realize the functionality of particular tasks. So, a concrete service model describes a concrete service plan as it not only defines the plan's services but also determines their temporal execution order. Such a model is produced from an enhanced service model via service selection. In this process, only one of the leaf nodes' candidate services is selected based on the selected service provisioning, the channels in which it is requested and delivered, and its semantic and syntactic similarity with the respective service model node. This process must guarantee that the selected services are compatible with each other at the semantic and syntactic (message) level, and the information loss between the involved services' message types is minimal. This requires using optimization algorithms that minimize

the information loss while preserving the service model robustness, respect the local context-of-use constraints, and select services best matching their respective tasks. Section 6.7 analyzes such an algorithm, used by our work to perform service selection. Its results are exploited by the enhanced-to-concrete service model transformation activity analyzed in Section 6.8.

The inclusion of service results within the service model description is a natural result of applying an MDA approach going from general to more specific models, which depend on the current context of use, and is needed for two reasons: (a) the association between tasks and services must be known in order to perform service selection based on the current context of use and (b) a concrete service model associates a task with the service selected to realize it, so it can be exploited via model transformations to produce the final service invocation code, while it also influences the concrete UI (CUI) design, as Section 4.2.1 shows.

The rationale for using this service model and not a normal service model (e.g., specified in BPEL or BPMN) is the following. First, a normal service model like BPEL is not user oriented as it does not represent the user high-level goals. BPEL and BPMN are not able to appropriately describe interaction tasks required for producing the application UI, so they cannot be exploited by a holistic model-driven service-based interactive application design approach, which considers both the UI and functionality aspects. It is unnecessary to use a normal service model that is automatically executable in service composition engines, as the interactive application's overall functionality is not exposed as an automatically executable service but is mainly controlled by user interactions. Finally, the proposed model retains the structure of its generating task model, so it could be used by reverse-engineering techniques, such as abstraction model transformations in MDAs, to produce a new task model when the modeling of the application functionality is changed.

4.2.1. Implications between the Design Models and Modeling Effort. The success of our method depends on the way the initial task model is specified. In particular, the designer must create a detailed task model with a very fine-grained granularity such that the system task leaves correspond to elementary domain operations, not further decomposable into smaller ones. These operations will probably be offered by existing services providing particular domain-specific functionality. Thus, leaf system tasks will be matched and eventually associated with these services. The next guidelines should be applied when specifying a task model in CTT because this notation does not indicate which objects manipulated by a system task constitute its input or output:

- The application objects manipulated by each task should be specified.
- Interaction tasks manipulate objects that can be used as input for the following system tasks, if these tasks have equally named application objects.
- A system task's object is its input object if matching a previous system task's object.
- A system task's unmatched objects can be considered its output.
- If a system task takes a particular object as input, modifies it, and finally produces it as output, then two equally named objects should be specified in its description. The first object will be considered the task's input, the second one its output.

In this way, the designer should carefully create application objects for each task and use equal names for identical objects shared between different tasks. As a result, through a particular task-to-abstract service model transformation, an abstract service model can be produced whose (task) nodes have input and output objects and the communication between them in terms of output-input objects is fixed.

These guidelines are not hard to follow. First, designers have good knowledge of the application domain and its operations. Otherwise, a modeling tool can be integrated

in USDS to provide designer support by indicating the tasks already associated to existing services. Second, designers know or have an idea about which application objects will be associated to system tasks. Apart from the guidelines, our approach does not require that the designer knows SW technology, as the appropriate domain ontology selection and the mapping of I/O application objects to ontology concepts is completely automated.

The UI and functionality aspects can be processed in parallel, but their models are not independent. The pairs of dependent design models are as follows. The first pair is the AUI and enhanced service model. In previous approaches, the AUI is inflexibly specified by defining the application objects used to get the appropriate user information to invoke services or to present the information received from the services invocation based on the already selected service message types, and by associating them to abstract interactors. This problem is remedied as follows. As application objects are mapped to domain concepts from the task-to-abstract service model transformation, the designer can be presented with the concept object and data type properties and choose those to be specified by or presented to the user. This selection leads to the filtering of those services associated to system tasks that have a message type not capturing all the concept-based user-requested information. This filtering can be performed either before the enhanced service model production by imposing a syntactic filter, apart from the semantic one, on service matchmaking, or after by filtering the services associated to leaf system tasks.

The second design model pair is the CUI and concrete service model. In this case, the concrete model influences the CUI by specifying the best service from the candidates to implement each system task's functionality. As various solutions producing a concrete service model can exist, the designer may choose some and then update the respective CUI. This requires an appropriate interaction between the designer and the modeling tool and a service selection algorithm producing many appropriate, robust solutions.

5. PROPOSED METHOD OVERVIEW AND ITS APPLICATION ON A USE CASE

The goal of this work is to support the (semi-)automatic production of a concrete service model from an initial HCI task model, thus realizing a significant part of an interactive application's functional design. To this end, a particular method is followed performing various steps, including model-to-model transformations, in a way that respects the dependencies between the UI and functional models. The USDS system realizes this method, whose architecture and implementation details are analyzed in Appendix B.

Figure 5 depicts a UML activity diagram showing the method steps' execution order and the interactions between the Service Front-End Authoring Environment (SFAE) and USDS. USDS controls the method execution and returns to SFAE all basic functional design models. The method starts by transforming an SFAE task model to an abstract service model (see *transformTaskModel* activity). In parallel, based on the task model, a specific ontology is selected (see *selectDomainOntology*) from the ontologies used to annotate the USDS repository service advertisements.

Next, the *enrichServiceModel* activity enriches the abstract service model by mapping all unique application objects to the ontology concepts by executing a specific algorithm that reuses the mapping knowledge produced. Then, *produceServiceQueries* runs the abstract model-to-service queries transformation, followed by *matchQueries* matching the queries derived with the service advertisements.

Depending on the modeler's choice provided by SFAE before *produceServiceQueries* is performed, an additional step is executed by obtaining the modeler's conceptual requirements and then filtering the produced service results according to them (see *filterServiceResults*). Then, the filtered service discovery results are merged into the service model producing an enhanced one (see *produceEnhancedServiceModel*). If the

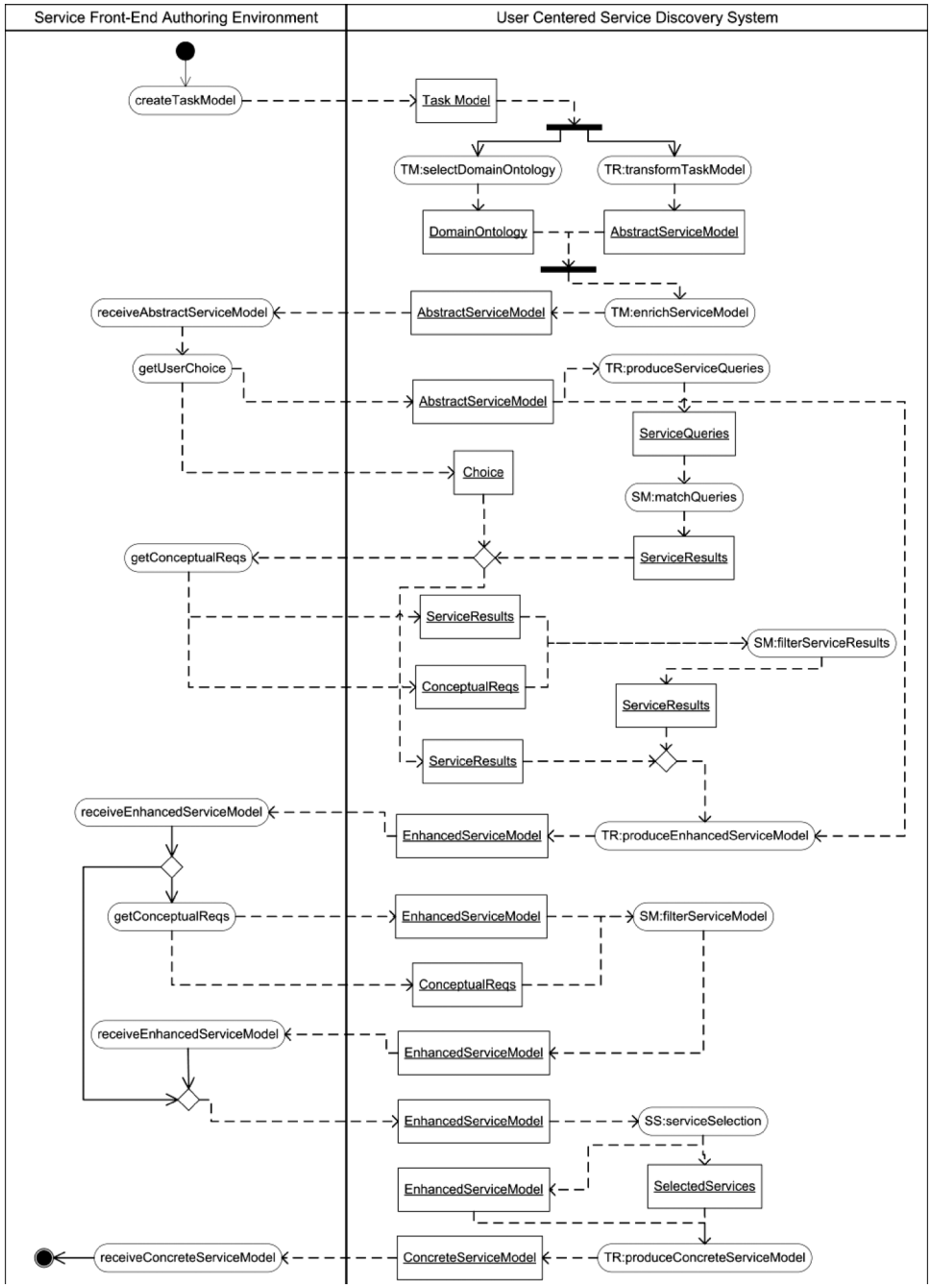

Fig. 5. The activity diagram showing the interactions between SFAE and USDS.

Table II. The Information of Application Objects

Application object name	Task IDs	Information	Concept
Departure City	6,12(I)	Flight's departure city	Airport
Arrival City	6,12(I)	Flight's arrival city	Airport
Date	6,12(I)	Date of flight	DateTime
Flight	9,10(I),18,21(I)	Flight	Flight
Flight List	12(O),14(I),15	List of flights	ListOfFlights
Map	13(O),14(I)	An area map showing flight information	Map
Name	19,21(I)	Person's name	Name
Telephone Num	19,21(I),22,23(I)	Person's telephone number	TelephoneNumber
Credit Card Num	19,21(I)	Person's credit card number	CreditCardNumber
Booking Result	21(O),22	The result of booking	BookingNumber
SMS Status	23(O),24	SMS sending result	Status

previous additional step was not executed, the modeler's conceptual requirements are obtained and used to filter the enhanced service model's results (see *filterServiceModel*). Finally, a concrete service model is produced from the enhanced one, by first executing a service concretization algorithm (see *serviceSelection*) and then keeping only the selected service for each enhanced service model task (see *produceConcreteServiceModel*).

This procedure may become repetitive, where the designer updates the task model based on the previous discovery results and re-executes our method to get the new results for the modified system tasks. It may be ended based on the designer domain experience and the application semantics or when no further task model updates can be performed.

The rest of this section explains the way this method is applied on a specific application, while the next section is dedicated to analyzing each method's activity.

Suppose that the application of the running example must be designed. This application is first modeled in the CTT editor to produce a task model, which represents the flight-booking tasks at the highest level involving the application functionality and interactions required to reach the user's goal. This model can be utilized by model-based user interface generation tools (e.g., MARIAE [Paternò et al. 2011]) to produce the application final UI. Figure 1 actually depicts the flight-booking application task model, where for each task both its ID and name are visualized in the form "ID. name".

Let us now explain the meaning of the node and arc icons of the task model visualization. Cloud icons represent *abstract* tasks, human icons represent *interaction* tasks, and PC icons represent *system* tasks. The meaning of the arc labels is as follows:

- "[>" denotes that the execution of the left task can be interrupted by the execution of the right task.
- "[]≫" denotes the *enabling with information passing* binary operator.
- "≫" denotes the *enabling* binary operator.
- "|||" denotes the *concurrency* binary operator (with no information exchange).

When a node's name is between brackets, the execution of the respective task is optional. When the name is followed by a "∗", the task can be executed repeatedly. The XML representation of a task model is produced by the CTT editor via a translation of the task model graphical representation. Figure 1 does not show the complete task information, as the CTT editor uses different visual parts to display and edit such information, which cannot be all depicted in just one figure along with the task hierarchy. Besides, parts of this information do not belong to the task model but are produced as a result of applying the method. Table II fills this gap by depicting the missing and

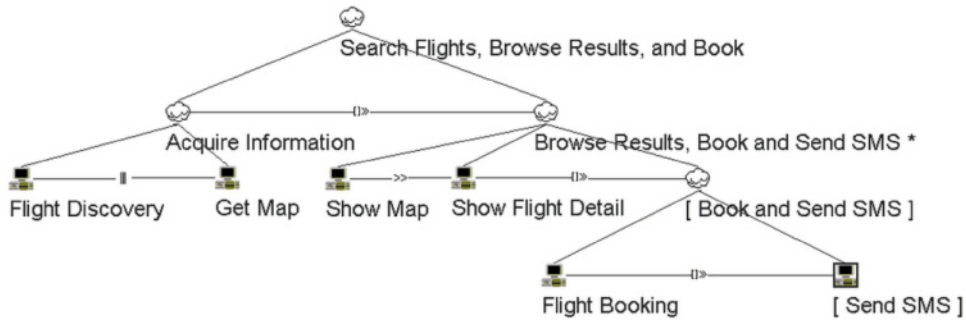

Fig. 6. The (abstract) service model produced.

additional information at the task model leaf level. This table describes each application object in the tasks, along with its I/O type (symbols "I" and "O" denote *input* and *output* objects, respectively), the information it represents (assisting the reader to understand the object semantics), and to which domain ontology concept it is mapped (extra information produced along with the object's I/O type). The focus on the leaf nodes is due to the fact that their information is exploited to perform service discovery so as to fulfill the application functionality.

Suppose that the designer, through a particular SFAE, desires to exploit our method to find services able to realize the required application functionality. To this end, she sends the task model to USDS and then the actual method begins. The various outputs produced, the task model (in XML form), and the ontologies and service advertisements were gathered in a RAR file available at http://giove.isti.cnr.it/tweb_service.rar.

Based on our method, the *transformTaskModel* activity is executed first to transform the use case's task model into the abstract service model depicted in Figure 6. This service model contains only *service* (denoted with the *abstract* tasks symbol) and *operation* (denoted with the *system* tasks symbol) tasks. By also inspecting Figure 1, it can be observed that all interaction tasks were removed, leaf system tasks were transformed to *operation* tasks, and abstract or high-level system tasks were transformed to *service* tasks.

In parallel, the *selectDomainOntology* activity is executed and selects the "NonMedicalFlightCompanyOntology.owl" ontology for annotating the application's task objects. This ontology is selected out of all ontologies used to annotate the service advertisements stored in the system repository, as it had the maximum IR similarity with the abstract service model. A part of this ontology is shown in the class diagram of Figure 7.

Next, *enrichServiceModel* enriches the abstract service model by mapping its application objects to ontology concepts. Table II shows the object-to-concept mappings produced. The mapping accuracy is 0.81, as nine out of 11 objects are correctly mapped. The incorrectly mapped concepts are highly related to the correct ones. In particular, both the departure and arrival city objects were mapped to the Airport concept as the selected ontology concept Flight is connected via respective object properties (hasDepartureAirport and hasArrivalAirport) to the Airport, while the latter is connected to the City concept. This modeling denotes the way flight discovery domain operations search for flights (i.e., by using the departure and arrival airports and not the cities with these airports). This signifies a missing system task, executed immediately before the "Flight Discovery" one, which must take a City's name and return the city's Airport name. Incorrect object mappings cause service discovery accuracy problems, as Section 7 shows.

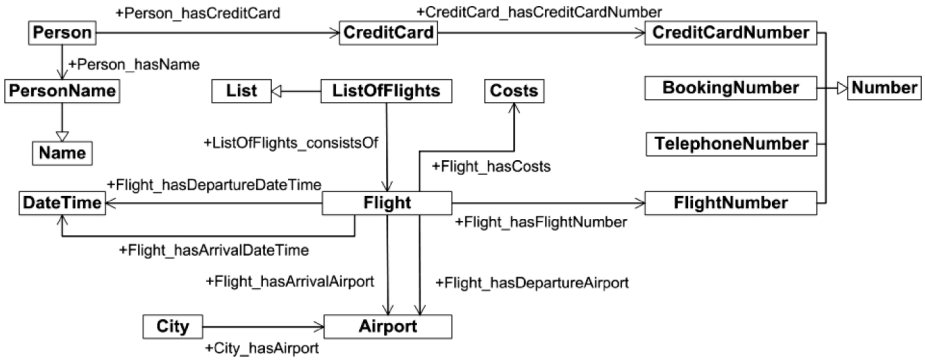

Fig. 7. Part of the NonMedicalFlightCompany ontology.

```
<profile:Profile rdf:ID="_Flight_Discovery_Profile">
   <service:isPresentedBy rdf:resource="#_Flight_Discovery_Service" />
   <profile:serviceName xml:lang="en">Flight_Discovery</profile:serviceName>
   <profile:textDescription xml:lang="en">   </profile:textDescription>
   <profile:hasInput rdf:resource="#_Departure_City" />
   <profile:hasInput rdf:resource="#_Arrival_City" />
   <profile:hasInput rdf:resource="#_Date" />
   <profile:hasOutput rdf:resource="#_Flight_List" />
   <profile:has_process rdf:resource="#_Flight_Discovery_Process" />
</profile:Profile>
<process:AtomicProcess rdf:ID="_Flight_Discovery_Process">
   <service:describes rdf:resource="#_Flight_Discovery_Service" />
   <process:hasInput rdf:resource="#_Departure_City" />
   <process:hasInput rdf:resource="#_Arrival_City" />
   <process:hasInput rdf:resource="#_Date" />
   <process:hasOutput rdf:resource="#_Flight_List" />
</process:AtomicProcess>
<process:Input rdf:ID="_Departure_City">
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://127.0.0.1/ontology/NonMedicalFlightCompanyOntology.owl#Airport
        </process:parameterType>
</process:Input>
<process:Input rdf:ID="_Arrival_City">
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://127.0.0.1/ontology/NonMedicalFlightCompanyOntology.owl#Airport
        </process:parameterType>
</process:Input>
<process:Input rdf:ID="_Date">
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://127.0.0.1/ontology/NonMedicalFlightCompanyOntology.owl#DateTime
        </process:parameterType>
</process:Input>
<process:Output rdf:ID="_Flight_List">
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://127.0.0.1/ontology/NonMedicalFlightCompanyOntology.owl#ListOfFlights
        </process:parameterType>
</process:Output>
```

Fig. 8. Snippet of the first OWL-S query produced from the use case's abstract service model.

The designer inspects the enriched abstract service model and finds all annotations relevant. She also selects to immediately filter the service results to be derived from the queries produced by the abstract service model's transformation. This transformation is performed by executing the *produceServiceQueries* activity, which results in producing six OWL-S queries, each one corresponding to a particular *operation* task. A snippet of the query corresponding to the *Flight Discovery* task is shown in Figure 8.

The produced queries are matched by executing the *matchQueries* activity with the system repository's service advertisements. The service results are then filtered according to the conceptual requirements obtained from the designer based on his choice. Based on the filtering process, some correct (i.e., not containing the correct amount of concept information) and incorrect exact matches are filtered. In the end, some tasks

Table III. The Discovery Results Produced from the Service Model's OWL-S Queries

Service query	Discovery results
flight_book__Flight_Discovery.owls	Flight_Search.owls (exact) (kept,selected), Athens_Flight_Search.owls (exact) (kept)
flight_book__Get_Map.owls	Get_Map2.owls (exact) (kept,selected)
flight_book__Show_Map.owls	Show_Map3.owls (exact) (kept,selected), Show_Map.owls (exact) (kept), Get_Map2.owls (exact) (kept), Get_Map4.owls (exact) (removed), Show_Map4.owls (plugin) (removed), Show_Map2.owls (plugin) (removed)
flight_book__Show_Flight_Detail.owls	Show_Flight_Details.owls (exact) (kept,selected), Get_Map4.owls (exact) (removed), Get_Map2.owls (exact) (kept)
Flight_book__Flight_Booking.owls	Flight_Booking.owls (exact) (kept,selected), Flight_Booking2.owls (exact) (kept)
Flight_book__Send_SMS.owls	Send_SMS.owls (exact) (kept,selected), Send_SMS2.owls (plugin) (kept)

have only one (exact) match, so only a few service combinations must be checked by the *serviceSelection* activity. Table III indicates which results were returned, their semantic category, and if they were kept or removed after the conceptual requirements filtering.

By inspecting each query's results, it can be observed that two *exact* matches are returned on average and *plugin* or *subsumes* matches are rarely returned. By also inspecting those service advertisements related to the queries, the average precision calculated is 0.83 and the average recall is 0.55. This means that in most cases, the results are precise but not all related results are returned. Only system tasks showing information (i.e., with only input) have lower precision, as they are matched with services that do not produce any result but have a subset of the required inputs (so not exactly achieving what is required) or produce some result (so their functionality is different). This indicates changing the way exact matchmaking is conducted for such queries to return only services not producing any output but with the same amount of related input. The lower recall value is due to two main factors: (a) incorrect object-to-concept mappings and (b) the existence of services requiring more input than requested or containing I/O concepts related to those requested but not via subsumption. Such recall problems can be solved by adding the missing input objects or also exploiting other relations apart from subsumption in service matchmaking.

The filtered service results are then used by the *producedEnhancedServiceModel* activity to transform the abstract service model to an enhanced one. The latter model is sent to the designer for inspection. Then, it is exploited by the *serviceSelection* activity to perform the service concretization. As a result, only one service is selected for each task, as Table III shows. Finally, the service selection results are exploited to transform the enhanced service model to a concrete one by executing the *produceConcreteService* activity.

6. PROPOSED METHOD ANALYSIS

Our proposed method's activities are now analyzed in separate subsections.

6.1. Task-to-Abstract Service Model Transformation

This transformation considers only abstract and system tasks from the given task model and generates an *abstract service model*, which is consistent with the task model's hierarchy and temporal semantics. Section 4.2 justified the rationale of keeping only these task types. Besides, the task model design guidelines provided in Section 4.2.1 must be followed. The transformation follows a Depth First Search. Depending on the

type of the first-time-visited node, the next rules are considered, where each rule covers one or more distinct cases and the rules' union covers all possible cases that may be anticipated.

- **Rule 1**: If the node represents a user, interaction, or abstract task with no system task descendants, it is removed as it cannot or is illogical to be mapped to a specific service. Interaction task application objects are stored in a global object list (GOL) as they enable characterizing a system task's application objects as input or output. The user interactions (and any other high-level UI-related information) are removed from the tree, as it is regarded that their information is either redundant (already covered by the information of respective (neighboring) system tasks) or irrelevant for service discovery, but they still exist in the UI models produced by the UI aspect's model transformation path. Thus, there is no critical UI information loss.
- **Rule 2**: If a system task node is a leaf, it is renamed as an *operation* task (**Rule 2.1**) as it corresponds to a service operation. If it contains more than one system task, it is renamed as a *service* (**Rule 2.2**) as it corresponds to a simple or composite service. A system task, containing only one system task as a child, is replaced with this child as the parent task is regarded to have the same functionality with it. The node name and description are kept as they represent important information that can be exploited during service discovery. Then, the next checks are performed for leaf system tasks to cover all cases for the tasks' application objects:
  - **Rule 2.3**: If one copy of a specific object is contained, then:
    - **Rule 2.3.a:** If the object name matches the name of an object stored in GOL, then we rename this object as an *input* node. This means that the object is input to the system task as it is produced by a previous task in the task execution order (i.e., either the user provides it or the system/service automatically produces it).
    - **Rule 2.3.b:** If the object name is unmatched, the object is renamed as *output*. In this case, the object is not produced by a previous task in the execution order so it must be an output object of the task.
  - **Rule 2.4**: If more than one copy of the same object is contained, then:
    - **Rule 2.4.a:** If the object name matches the name of an object stored in GOL, the first copy is renamed as *input*, the second as *output*, while the remaining copies are discarded. This means that the first copy is an input object to the task, based on the rationale of Rule 2.3.a, while the second is an output object, as the designer desires to define a task operating on a specific object and returning its modified version. The remaining object copies are discarded.
    - **Rule 2.4.b**: If the object name is unmatched, then the first copy is renamed as *output* based on the rationale of Rule 2.3.b. The remaining object copies are discarded.

    Renamed leaf system task objects are copied to GOL as they will be used to characterize the application objects of the next tasks in the task execution order.
- **Rule 3**: Abstract tasks with two system task descendants are renamed as *service* as they represent composite or simple services. Otherwise, they are replaced by their sole system/service task descendant, which becomes repetitive or optional if its ancestor abstract task has the same, respective property.

It must be noted that when a node is replaced by its child node, the child node's content is copied to the parent's one so that the parent's connections with its siblings are kept.

During a nonroot node removal, its placement with respect to its siblings is considered to produce consistent task/service models based on the next three cases, depicted in Figure 9:

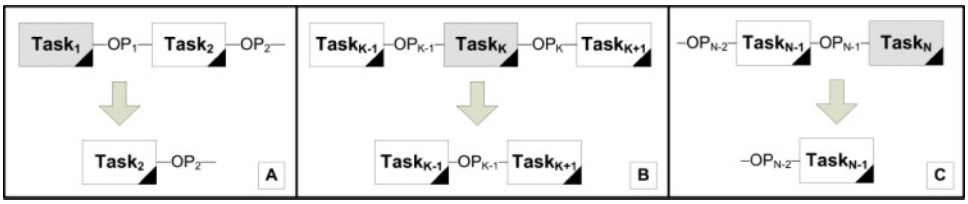

Fig. 9. The three cases of node removal.

- *Leftmost*. Both the node and its right operator are removed (**Rule 4.1**).
- *Intermediate*. Both the node and its right operator are removed and its left sibling is connected with its right one using its left operator (**Rule 4.2**). In other words, the node is replaced by its right sibling so the right operator connecting them is replaced by the left one as the priority of the temporal operators goes from the left to the right.
- *Rightmost*. The node is removed along with its left operator (**Rule 4.3**).

6.2. Domain Ontology Selection

This activity must select an ontology best capturing the task model domain among an ontology set already used to annotate our system repository service advertisements. The information used in service discovery must be limited and accurate and not contain terms causing reduction in a service advertisement's similarity with a service query. That is why the irrelevant user, interaction, and particular abstract task information is deleted in a task model-to-abstract service model transformation. However, the interaction task description may reference domain-related terms, so it can positively influence the appropriate ontology's selection as the task model's similarity with this ontology will be increased.

To this end, both the task model *TM* and each domain ontology *Oi* are considered as documents containing terms, and IR vector-based techniques [Baeza-Yates and Ribeiro-Neto 1999] are used to find the document pair (*TM*, *Oi*) having the greatest similarity so as to select the appropriate domain ontology. The procedure considered is the following:

1. Preprocessing step:
   a. From each ontology *Oi* a set of words are extracted representing ontology terms' (i.e., concepts, properties, and objects) names. Each word is split into character strings called *terms*. For instance, "ListOfFlights" is separated into {"list","of","flights"}. Each *term* is then either removed, if it is very common (e.g., "of"), or otherwise stemmed. Next, each stemmed term t is associated with a weight w according to the tf-idf term-weighting scheme. Thus, each ontology *Oi* is finally represented as a document containing a set of term-weight pairs, where a zero weight denotes the term's absence.
   b. The same procedure is followed for the *TM*. The words extracted either represent task and application object names or are inside task descriptions. In the end, the *TM* is represented in the same way as the ontologies.
2. Selection step: The similarity between the *TM* and each ontology document *Oi* is computed according to the cosine metric. The greatest similarity pair is selected and the corresponding ontology best captures the abstract service model domain.

6.3. Abstract Service Model Enrichment

The mapping of abstract service model application objects to a domain ontology concept relies on the approach of Gracia and Mena [2008] (see Equation C.1 in Appendix C) and is performed by the following procedure. Each application object name is first split into

character strings that are joined using the space character. For example, the object name "Flight_2\List" is converted to "Flight List." This procedure has already been followed for all ontology terms. The new string produced represents the object's word. If this word has already been matched (e.g., from a previous service model enrichment), the matched concept is fetched along with the matched data type property (if it exists). Otherwise, the similarity of each ontology term's word with the object word is calculated based on Equation C.1 and the term with the greatest similarity is selected. A small adjustment is performed in the equation when matching an ontology concept with no parent classes by nullifying the equation's second part, setting the first part weight (w_o) to one.

If the greatest similarity s found is below a threshold, a string in the application object word is either grammatically incorrect or not strongly related to the other strings. In this case, the previously mentioned procedure is re-executed for all meaningful object word substrings. For example, if the object word is "Credit Card Num," the word-to-ontology term matching is performed for "Credit," "Card," "Num," "Credit Card," "Credit Num," and "Card Num." The term best matching these subwords with the greatest similarity s' is picked up. If s' is greater than s, the new term is selected. Otherwise, the term with similarity s is kept.

Service discovery algorithms match I/O parameter concepts of service descriptions, so application objects transformed to service queries' I/O parameters must be mapped to ontology concepts. So, depending on the selected term's type, the next cases are checked:

- If the term represents a concept, then it is the matched concept.
- If the term represents an instance, then its respective concept is the object of search.
- If the term represents a property, one of the following two subcases may hold:
  - The object property's range concept is the matched concept as it models the particular information aspect of the domain concept that is desired by the user. The property is not selected, as it cannot be used to get more details about the range concept.
  - Both a data-type property and its domain concept are kept as the domain concept exists and is highly related to the property, or the application object is an input to an information search service, so the property must also be kept, as it also leads to the domain concept searchable information.

The matching found is finally stored. The abstract service model is enriched by adding to its application objects' description the matched ontology concepts and properties' URIs.

6.4. Abstract Service Model-to-Service Queries Transformation

In this transformation, for each abstract service model *operation* node, a respective OWL-S service description is created as follows. The name of the OWL-S service query file is first produced from the service model file name (without the ".xml" suffix) and the operation node name (i.e., the *Name* element's content). Moreover, the service-query-produced content mainly applies to the *ServiceProfile* and *ServiceModel* parts (where an atomic process is created as each query represents a simple service request). In particular, the service is named (i.e., the *serviceName* element's content in the *ServiceProfile* part) after the node's name, the service ID and its profile and atomic process IDs are produced from the node's name, while the service textual description (i.e., the *textDescription* element's content) is copied from the node's *Description* element content. In addition, process input and output nodes (i.e., *process:Input* and *process:Output* elements) are produced with IDs named after the name of the operation node's corresponding I/O application objects (i.e., the *appl_name* attribute's value of the *Input* or

Output respective element's *appl_object* element), which have the content of their *parameterType* element pointing to the I/O application objects' ontology concept URI (i.e., the *concept* attribute's value of the *Input* or *Output* respective element's *appl_object* element). The grounding information, as it does not play an important role in service matchmaking, is almost empty (only information needed for correct service description processing is created).

6.5. Service Matching

The SM component (see Appendix B) is responsible for performing service specification management and matching service queries with the service advertisements stored in its repository by also returning back the results in appropriate categories. It exploits the OWLS-MX SSME, offering its functionality as a black box, to achieve its functionality.

SM communicates with the SSME to perform the actual matchmaking and propagate service advertisement updates to the SSME inner structures. Special treatment is reserved for an advertisement modification, which is realized through a deletion followed by an insertion, so as to correctly update the SSME inner structures (e.g., to avoid duplicates).

SM extends the SSME functionality in two ways. First, when a set of service queries, pertaining to a particular abstract service model, are issued to SM, they are sent one by one to OWLS-MX for matchmaking. Then, the SM translates the results returned based on the service model's *results* element and sorts them into each category based on their score. This score represents the overall (semantic, syntactic, and I/O) similarity of the result's service with the issued query. Appendix D explains how this score is calculated.

Second, SM can impose a syntactic filter on *operation* task results at the message level. In particular, through the SFAE the designer can specify which message types must be associated to the operation task parameters. Each message type contains a specific amount of information describing a particular concept and can be used to filter those services using message types with less information for this concept. For instance, if a requested composite message type contains M message types, a service with a respective composite message type including N message types, where a subset of N is mapped to an M subset, will be filtered. Appendix E analyzes the process of matching message types.

As our approach dynamically updates the service advertisement space, it can provide support to dynamic service binding in two ways. First, if a service selected for a specific task fails or is unavailable, the respective application's execution can continue by rebinding to the task's next best matching service. Second, to guarantee the application's robustness, service selection can be performed by the proposed selection algorithm for all remaining application tasks (including the "problematic") based on the application temporal semantics, as the initial application tasks' services may have been executed.

6.6. Abstract Service Model and Service Discovery Results-to-Enhanced Service Model Transformation

In this transformation, for each abstract service model operation node, its respective query's results, represented by the *results* element, are enclosed within its description. The created element contains a specific number of elements corresponding to the number of categories being returned from SM. As only *exact*, *plugin*, and *subsumes* matches interest us, the maximum element number is three. Each category's results are added to its respective *service* element containing three attributes (*URI*, *name*, and *score*) and two elements (*inputParam* and *outputParam*). The latter elements specify the service I/O parameters. Their content is derived from the service OWL-S and WSDL files and the query.

6.7. Service Selection

This activity transforms the enhanced service model into a reachability graph and then, based on the graph and service model, solves an optimization problem. This problem aims to minimize the average information loss and maximize the average similarity of the services selected for the operation tasks by considering task selection and execution path robustness constraints. This section shortly analyzes how service selection is performed. Appendix E analyzes the whole activity and thus the respective transformation.

Modeling of the Service Selection Problem. Suppose that the enhanced service model contains a set of (leaf) tasks T. After generating the reachability graph, its execution paths are enumerated. Let P denote the set of all execution paths enumerated. Each path p in P is represented as a task set $Tp = (\text{Ti,Tj,}\ldots\text{,Tk})$, where Ti belongs in T (i.e., with a set of nodes representing tasks). These nodes' order indicates the path's task execution order.

Based on the execution path set P and the available information on the tasks involved in P and on the services associated to them, a specific constraint satisfaction optimization problem [Rossi et al. 2006] (SCOP) is created and solved so as to robustly select the best service for each enhanced service model task.

The main variables modeled in the SCOP are as follows. Three variables, namely, $score_p$, $loss_p$, and $match_p$, are modeled for an execution path: (a) $score_p$ provides the path's overall score, (b) $loss_p$ stores the average information loss in a particular path, and (c) $match_p$ stores the average matching score calculated over all the path's tasks. For each task t_i in the leaf task set T of the considered service model, two sets and one variable are modeled: sets $inputs_{t_i}$ and $outputs_{t_i}$ contain the task's input and output parameters, respectively, inherited from the task's selected service, while $match_{t_i}$ is the task's matching score, produced based on the task's selected service. A binary variable set z_{ij} is also modeled indicating for each task t_i which services s_{ij} from its discovered set Si has been selected.

Four functions are exploited in this SCOP: (a) loss calculates the information loss between output and input (O/I) parameter sets by using (b) *mloss* calculating the information loss between an O/I parameter pair's message types, (c) incomp checks if there is a matching pair of incompatible O/I parameters, and (d) matches finds the matching pairs between O/I parameter sets.

The main SCOP objective is to minimize the sum of all the paths' scores. A path score is computed by the weighted difference between the path information loss and its matching score through the Simple Additive Weighting technique [Yoon and Hwang 1995], which exploits the fact that the subscores are normalized as they are averaged. As such, the main objective translates to minimizing the overall information loss and maximizing the overall matching similarity. The subscores' weight signifies their relative importance and can be set by the designer based on his or her expertise in the application domain.

The information loss of a path equals 2 if the information loss in one of the path's task transitions is maximal (i.e., there is a transition where the two tasks involved exchange incompatible messages). Otherwise, it equals the average information loss in all the path's transitions. The information loss for a path's transition is calculated in turn by applying the loss function on the unique outputs produced in the previous transitions and the inputs of the current transition. This function returns 2 if there is one matching output/input (O/I) pair in the function parameters such that the application of this pair to the *mloss* function returns the value of 1 (i.e., the pair's message types are incompatible). Otherwise, it returns the average from the values returned by applying the *mloss* function on all O/I matching pairs. Function matches is used to compute the matching O/I pairs.

A path's matching score is calculated by taking the average of its tasks' matching scores.

The SCOP expresses two critical constraints. First, the average loss in each execution path must be less than one to guarantee the path's robustness in terms of encompassing consecutive services that do not exchange incompatible messages. Second, for each task t_i, only one from its associated services s_{ij} must be selected, that is, $\sum_j z_{ij} = 1$.

Apart from the critical constraints, other constraints exist, propagating information (e.g., the matching score and the O/I parameters) from the service selected to its realizing task.

For a complete analysis of the SCOP and an explanation of how the *mloss* function is calculated, the interested reader can refer to Appendix E.

6.8. Enhanced-to-Concrete Service Model Transformation

This transformation exploits the services selected by the service selection activity along with the enhanced service model to produce the concrete service model. It is performed by keeping only the selected service in each operation's task results of the enhanced model.

7. EVALUATION

This evaluation focuses on assessing the USDS system realizing the proposed method, especially with respect to the relevance of the services discovered and selected to realize the interactive application's functionality and the robustness and optimality of the produced design solution. As such, our work claims related to the USDS's distinguishing features are validated. Apart from accuracy and optimization issues, USDS performance was also assessed to show its added value with respect to the time that a human designer, who may cause design errors and derive nonoptimal solutions, might need to manually perform the respective service discovery and selection steps. Please note here that we have not performed user tests so as to assess what is the time that a user would require to design an application with or without USDS. We are more interested in how quickly and efficiently USDS leads to a particular service solution that could be exploited by the designer to also produce the application FUI. We argue that no matter how efficient and experienced the designer is, he or she will not be able to reach a correct, robust, and optimal service solution in such a short time as USDS does. Of course, we are not advocating that the designer will be totally replaced by USDS, but that he or she will be supported in the service-based application design through the efficient and effective recommendation of the best robust service solution for the application at hand.

Four task models (available in the use case RAR file), including the use case model, were exploited to evaluate USDS performance. The first one, named "educational-scenario," concerns the procedure of viewing a student's subscriptions in a semester and updating them by making new subscriptions to courses or deleting them. The "AmazonSearch" model concerns the procedure of searching for a book with either the author name or title and viewing the book results returned. The third model, named "HomeAutomation," concerns the procedure of viewing and controlling a set of devices.

Eight parameters were used to evaluate USDS according to its matchmaking accuracy and performance. Two parameters indirectly influence the service matchmaking accuracy: (1) the percentage of correct domain ontologies selected and (2) the average mapping accuracy measured as the percentage of correctly mapped concepts to the task model's application objects. The six direct-influence parameters were as follows: (1–2) the overall maximum and average matchmaking time, (3–4) the average and minimum service discovery precision measured as the number of correct results divided by the

number of those returned, and (5–6) the average and minimum service discovery recall measured as the number of correct results returned divided by the total number of correct results.

The USDS performance and optimality with respect to service selection were evaluated according to three parameters: (1–2) the maximum and average execution time and (3) the average accuracy measured by the total number of times the SS component returned the best correct result divided by the total number of times SS was executed.

The advertisement set used in the evaluation comprised advertisements that were both randomly produced and belonged to OWLS-TC. OWLS-TC is a collection of 1,007 OWL-S advertisements and 29 queries and general and domain-dependent ontologies. Each advertisement randomly created had from one to four parameters, representing possible domain operation implementations, as there were no OWLS-TC respective services matching some task model tasks. The randomized semantic advertisements were accompanied with respective WSDL files created so that I/O parameters annotated with the same concept were mapped to a different message type representation. As such, we ensure that there exist services associated to tasks of the four task models, exchanging semantically equivalent but differently represented information.

The rationale for selecting the particular advertisement set is as follows. First, composite services are not properly described, so they cannot be used for discovery purposes for a task model's nonleaf tasks. Thus, only services corresponding to simple operations had to be matched to a service model's leaf tasks. Second, apart from OWLS-TC, the other two semantic service collections are incomplete. The first, produced for the SWS Challenge (http://sws-challenge.org/wiki/index.php/Main_Page), is limited to three matchmaking scenarios with a small semantic service advertisement number, while the second, the Jena Geographic Dataset (http://fusion.cs.uni-jena.de/professur/jgd), contains only 50 semantic service descriptions annotated with concepts of two ontologies. There exist sites providing WSDL information for a large existing WS set. However, this information must be collected and converted to OWL-S, and the I/O of each OWL-S description must be mapped to domain ontology concepts. Even if this is performed, for some evaluated task models, no existing service matching their tasks can be discovered.

The assessment phase was automatically performed according to the proposed method without the message-based filtering step. For each task model, the following sequential steps were repeatedly performed for 10 times:

1. The best-matching OWLS-TC domain ontology was manually identified for the task model. Then, ontology selection was performed and the ontology was compared with the correct, best-matching one.
2. The correct ontology concepts to be mapped to each task's application objects of the model were identified. The application object-to-domain ontology concept mapping algorithm was then executed and the mapping accuracy was measured by inspecting the concepts mapped to the model application objects against the correct ones.
3. The task model was automatically transformed into two abstract service models, the correct and the method's one, which were manually annotated with the correct and discovered annotations, respectively, and then automatically transformed into two sets of OWL-S queries corresponding to the model's tasks.
4. A semantic service set was randomly created and merged with OWLS-TC. The tasks' respective queries annotated with the correct concepts were then associated to their related results manually by inspecting the whole repository and these results were exploited to manually discover the task model's best robust service composition.
5. For 10 times, the following sequential steps were repeatedly performed:

Table IV. The Service Matchmaking Evaluation Results According to 4 Task Models

Task model	Avg. map. acc.	Min prec.	Avg. prec.	Min rec.	Avg. rec.	Avg. exec. time (sec)	Max exec. time (sec)
Amazon Search	0.7500	0.2954	0.5318	0.2500	0.5753	33.50	36.8
Educ. Scenario	0.8181	0.2000	0.5157	0.1000	0.5018	86.60	91.0
Home Auto/tion	0.8181	0.0700	0.6480	0.1500	0.3632	95.70	97.9
Flight Discovery	0.8181	0.3333	0.8300	0.1428	0.4936	50.60	52.8
Per Task						08.54	09.2
Overall	0.8011	0.0700	0.6313	0.1000	0.4834	66.60	97.9

a. The tasks' respective queries annotated with the discovered concepts were matched and the matchmaking precision and recall for each query/task was computed by comparing the results returned with those marked as relevant. In parallel, the execution time for matching each task was measured.
b. The (method's) abstract service model was manually updated with the service discovery results of each leaf application task to produce the enhanced service model. Then, the discovery results and the abstract service model were used to produce the enhanced service model.
c. The service selection activity transformed the enhanced into a concrete service model, which was compared to the correct one, while its execution time was also recorded.

At each repetition end of the three substeps' execution in step 5, the average and maximum/minimum parameter values were updated, while in each task model processing end, the final values for the task model's respective evaluation parameters were produced. Finally, the overall parameters values were assessed based on those of the individual task models.

Table IV summarizes the assessment results for the service matchmaking evaluation parameters without presenting the ontology selection accuracy, which was always ideal. The mapping accuracy varied between 0.75 and 0.8181, with an average value of 0.80. Moreover, the precision had a minimum value of 0.07 and an average value of 0.63, while the recall had a minimum value of 0.1 and an average value of 0.48. Finally, the matchmaking time had a peak and average value of 97.9 and 66.6 seconds, respectively.

Thus, the domain ontology selection and term-matching steps are performed with the highest accuracy. While the service matchmaking precision and recall are moderate, two remarks must be made. First, the top results in each result category are always the most relevant. Second, recall is low as services partially fulfilling a query's goal are also considered relevant. Inaccurate term matching is one factor contributing to the low precision and recall values. Another factor is the nature of specific domain goals queries. In some cases, this nature is restrictive so some returned results are irrelevant. In other cases, this nature is quite generic, demanding more results than those actually returned.

As service matching is performed for all leaf system tasks, the service matchmaking time increases linearly with respect to the number of such tasks in each task model. Moreover, as the difference between the average and maximum time for each task model is not very high, our system is quite robust with respect to external environment interference. However, the difference between the overall average and maximum matchmaking time is high. This is mainly due to the way these measures are computed from their components and the fact that the task models did not have the same leaf

Table V. The Service Selection Performance Assessment According to 4 Task Models

Task model	Avg. acc.	Avg. exec. time	Max exec. time
Amazon Search	1	212 sec	267 sec
Educ. Scenario	1	409 sec	487 sec
Home Auto/tion	1	386 sec	433 sec
Flight Discovery	1	287 sec	359 sec
Overall	1	323.5 sec	487 sec

system task number. Finally, by dividing the average and maximum matchmaking time of a task model with the number of leaf system tasks, the respective measure values per system task were computed and presented in the fifth row of Table IV. As it can be also easily derived, the task level results do not have major differences with those at the task model level.

Table V summarizes the service selection assessment results. First, the service selection activity always returns the correct robust concrete service model. Service selection time is also of greater magnitude than total service matchmaking time for all tasks of a service model. This is justified by the fact that the service selection problem is NP hard, while the service-matching problem is polynomial to the size of the advertisement set. The service selection time increases exponentially with the increase on the system task number. Finally, the service selection time is so high that it dominates the execution time of all other USDS components. This means that different optimization techniques may be required to reduce this time but maybe with the cost of having a lower accuracy.

Based on the evaluation results, the maximum time spent in the service matchmaking and selection steps is around 600 seconds. This time is far less than the time the designer would spend to manually discover the relevant services for each task and then perform service selection to produce a concrete service model. Besides, as designers are humans, it could be quite easy to introduce mistakes in the design process or produce a nonoptimal or robust solution. So, our system provides added value to the interactive service-based application design process by reducing the overall design time, eliminating errors that human intervention could create, and producing optimal design solutions. The first advantage clearly leads to reduced time to market for interactive business applications.

8. DISCUSSION

In this article, a new model-driven design method for service-based applications was proposed, exploiting task models that consider both the UI and functionality aspects and transforming these models to concrete service models. Such service models describe how existing services can be robustly combined to fulfill the interactive application's functionality. Our method also exploits the domain knowledge modeled and produced by using semantics from domain ontologies to enable automating some of its steps.

A semiautomatic environment realizing this method was also developed, exhibiting the advantages of being seamlessly integrated into existing model-based HCI design methods to complete the development of interactive service front-ends and providing useful indications to application designers and developers, such as the places of missing functionality to be implemented and the application objects' data types. Finally, a novel service selection algorithm is proposed, which minimizes the overall information loss between pairs of information-exchanging services and selects the best possible service for each leaf system task based on its semantic, syntactic, and I/O similarity with this task.

Based on the evaluation results, two main problems hinder our method's success. First, the application object-to-ontology concept mapping accuracy is 0.8. This means

that some application objects are mapped to wrong ontology concepts and thus the services associated to the respective task will be irrelevant. This problem is created mainly by the abbreviations that the mapping algorithm does not properly handle. To this end, two alternative directions can be pursued: first, performing abbreviation expansion, and second, advising the designers not to use abbreviations at all. However, before following one of these directions, to enable the accurate system functioning, we propose an intermediate interactive step after executing the mapping algorithm, in which the designer inspects the mappings discovered and either asks for the problematic mappings modification through the mapping algorithm re-execution or identifies the appropriate domain ontology concepts by viewing the domain ontology selected. The mapping algorithm re-execution will propose different mappings for the same application object and it will be the task of the designer to select the most appropriate one.

The second problem is the medium precision values in service matchmaking. While this problem is remedied by the fact that the top results in each result category are always correct, it propagates to robust service selection, where the irrelevant results participate as candidates for task selection. This problem can be solved by exploiting pre- and postconditions to filter the irrelevant service results produced and appropriately extending the SSME used. This means that both the semantic service and task specifications must be additionally described with this aspect. Before solving this problem, the correct functioning of our environment can be enforced by discovering the exact percentage of topmost relevant results in each category that can be retained for the service selection phase. If such a percentage cannot be discovered as it may vary for each domain or operation, an alternative solution is to interact with the designer who can inspect the discovery results for each task and retain only the relevant ones. While the latter solution imposes additional requirements on the designer, it is far better than the case where the designer inspects the entire solution space to find the relevant solutions for specific tasks.

As the domain ontology is used to provide semantics to the task's application objects, its quality constitutes a crucial factor by influencing the service results accuracy produced by the SSME exploited. This ontology must be as complete as possible, capturing any possible domain entity and its relations and properties. Otherwise, it can lead to the incorrect mapping of a task's I/O application objects to wrong ontology concepts, thus associating the task to wrong service results. As indicated in Section 1, the ontology should be accurate by creating a new subconcept when one concept's information is enriched to avoid producing syntactically nonrobust service compositions. Such an ontology could be difficult to design. However, well-designed ontologies accurately and almost completely capturing a domain's semantics do exist. Even if such ontologies are incomplete (e.g., a required concept is missing), the mapping algorithm can associate this concept to an existing and highly related one to obtain relevant results again.

Concerning future work, many directions can be pursued: first, the integration of our method in existing service front-end environments; second, the extension of the mapping algorithm for annotating WSDL-based service advertisements; third, the investigation of how to change service matchmaking in order to enable property matching; finally, the method extension to handle nonfunctional constraints (e.g., usability, performance) for service matchmaking [Kritikos and Plexousakis 2009] and concretization and the improvement of the service concretization activity's performance.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

R. Baeza-Yates and B. Ribeiro-Neto. 1999. *Modern Information Retrieval*, 1st ed. Addison Wesley.

A. Brogi, S. Corfini, and R. Popescu. 2008. Semantics-based composition-oriented discovery of Web services. *ACM Transactions on Internet Technology* 8, 4, 1–39.

P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. 2004. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems* 8, 203–236.

G. Broll, S. Siorpaes, M. Paolucci, E. Rukzio, J. Hamard, M. Wagner, and A. Schmidt. 2006. Supporting mobile service interaction through semantic service description annotation and automatic interface generation. In *Proceedings of the Semantic Desktop and Social Semantic Collaboration Workshop at ISWC 2006*.

J. M. G. Calleros, G. Meixner, F. Paternò, J. Pullmann, D. Raggett, D. Schwabe, and J. Vanderdonckt. 2010. *Model-Based UI XG Final Report*. W3C Incubator Group Report, W3C. 04 May. Retrieved from http://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui/.

G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. 2008. A framework for QoS-aware binding and re-binding of composite web services. *Journal of Systems and Software* 81, 10, 1754–1769.

F. Casati, S. Ilnicki, L.-j. Jin, V. Krishnamoorthy, and M.-C. Shan. 2000. Adaptive and dynamic service composition in eflow. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*. Springer-Verlag, 13–31.

X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. 2004. Similarity search for web services. In *Proceedings of the 30th International Conference on Very Large Data Bases*. 372–383.

A. M. Ferreira, K. Kritikos, and B. Pernici. 2009. Energy-aware design of service-based applications. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC'09)*. Springer.

J. M. C. Fonseca (ed.). 2010. W3C *Model-Based UI XG Final Report, May 2010*. Retrieved from http://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui-20100504/.

J. Gracia and E. Mena. 2008. Web-based measure of semantic relatedness. In *Proceedings of the 9th International Conference on Web Information Systems Engineering*. Springer-Verlag, 136–150.

M. C. Jaeger, G. Mühl, and S. Golze. 2005. QoS-aware composition of web services: A look at selection algorithms. In *Proceedings of the International Conference on Web Services (ICWS'05)*. IEEE Computer Society.

J. Janeiro, A. Preubner, T. Springer, A. Schill, and M. Wauer. 2009. Improving the development of service-based applications through service annotations. In *Proceedings of the IADIS WWW/Internet Conference*.

D. Khushraj and O. Lassila. 2005. Ontological approach to generating personalized user interfaces for web services. In *Proceedings of the International Semantic Web Conference (ISWC'05)*. Springer, 916–927.

M. Klusch, B. Fries, and K. Sycara. 2009. 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.

K. Kritikos and S. Kubicki. 2011. A goal-based business service selection approach. In *Proceedings of the IEEE Conference on Commerce and Enterprise Computing*. IEEE Computer Society.

K. Kritikos and F. Paternò. 2010a. Service discovery supported by task models. In *Proceedings of the ACM-SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'10)*.

K. Kritikos and F. Paternò. 2010b. Task-driven service discovery and selection. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*.

K. Kritikos and D. Plexousakis. 2009. Mixed-integer programming for QoS-based web service matchmaking. *IEEE Transactions on Services Computing* 2, 2, 122–139.

D. Lau and J. Mylopoulos. 2004. Designing web services with Tropos. In *Proceedings of the International Conference on Web Services (ICWS'04)*.

F. Lécué, A. Delteil, A. Léger, and O. Boissier. 2009. Web service composition as a composition of valid and robust semantic links. *International Journal of Cooperative Information Systems* 18, 1, 1–62.

I. Manolescu, M. Brambilla, S. Ceri, S. Comai, and P. Fraternali. 2005. Model-driven design and deployment of service-enabled web applications. *ACM Transactions on Internet Technology* 5, 3, 439–479.

G. Mori, F. Paternò, and C. Santoro. 2002. CTTE: Support for developing and analyzing task models for interactive system design. *IEEE Transactions on Software Engineering* 28, 8.

F. Paternò. 1999. Model-based design and evaluation of interactive applications. Springer-Verlag.

F. Paternò, C. Santoro, and L. D. Spano. 2011. Engineering the authoring of usable service front ends. *Journal of Systems and Software* 84, 10, 1806–1822.

P. F. Pires, M. R. F. Benevides, and M. Mattoso. 2003. Building reliable web services compositions. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*. Springer-Verlag, 59–72.

M. Pistore, P. Traverso, and P. Bertoli. 2005. Automated composition of web services by planning in asynchronous domains. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS'05)*. 2–11.

P. Plebani and B. Pernici. 2009. URBE: Web service retrieval based on similarity evaluation. *IEEE Transactions on Knowledge and Data Engineering* 21, 11, 1629–1642.

F. Rossi, P. V. Beek, and T. Walsh. 2006. *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Elsevier Science Inc., New York, NY.

M. Ruiz, V. Pelechano, and O. Pastor. 2006. Designing web services for supporting user tasks: A model driven approach. In *Advances in Conceptual Modeling–Theory and Practice*. LNCS, vol. 4231. Springer, 193–202.

M. Sasajima, Y. Kitamura, T. Naganuma, K. Fujii, S. Kurakake, and R. Mizoguchi. 2008. Obstacles reveal the needs of mobile Internet services—OOPS: Ontology-based obstacle, prevention and solution modeling framework. *Journal of Web Engineering* 7, 2, 133–157.

E. Stroulia and Y. Wang. 2005. Structural and semantic matching for assessing web-service similarity. *International Journal of Cooperative Information Systems* 14, 4, 407–438.

K. Sycara, S. Wido, M. Klusch, and J. Lu. 2002. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous Agents and Multi-Agent Systems* 5, 173–203.

J. Vanderdonckt. 2005. A MDA-compliant environment for developing user interfaces of information systems. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE'05)*. LNCS, vol. 3520. Springer, 16–31.

J. Vermeulen, Y. Vandriessche, T. Clerckx, K. Luyten, and K. Coninx. 2007. Service-interaction descriptions: Augmenting services with user interface models. In *Engineering Interactive Systems (EIS)*. Springer-Verlag, 447–464.

D. Wu, B. Parsia, E. Sirin, J. A. Hendler, and D. S. Nau. 2003. Automating DAML-S web services composition using SHOP2. In *Proceedings of the International Semantic Web Conference (ISWC'03)*. LNCS, vol. 2870. Springer, 195–210.

K. P. Yoon and C.-L. Hwang. 1995. Multiple attribute decision making: An introduction. Sage Publications Inc.

A. M. Zaremski and J. M. Wing. 1997. Specification matching of software components. *ACM Transactions on Software Engineering and Methodologies* 6, 4, 333–369.

L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. 2004. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering* 30, 5, 311–327.

Received April 2012; revised October 2012; accepted May 2013