Natural Language Service Composition with Request Disambiguation

Florin-Claudiu Pop[1], Marcel Cremene[1], Mircea Vaida[1], and Michel Riveill[2]

[1] Technical University of Cluj-Napoca, Romania
{florin.pop,cremene,mircea.vaida}@com.utcluj.ro
[2] University of Nice, Sophia-Antipolis, France
riveill@unice.fr

Abstract. The aim of our research is to create a service composition system that is able to detect and deal with the imperfections of a natural language user request while keeping it as unrestricted as possible. Our solution consists in three steps: first, service prototypes are generated based on grammatical relations; second, semantic matching is used to discover actual services; third, the composed service is generated in the form of an executable plan. Experimental results have shown that inaccurately requested services can be matched in more than 95% of user queries. When missing service inputs are detected, the user is asked to provide more details.

1 Introduction

The major issue for a natural language composition system is to understand the user request, knowing that natural language implies a diversity of expressions, may be inaccurate and incomplete. Some user queries may contain indications about how to achieve a goal (i.e. *book flight from Paris to London*) but other expressions (i.e. *get me to London*) will just indicate a goal without giving an indication about how to achieve it. The user request needs to be interpreted in a specific context; some of the user needs are implicit, etc.

To overcome these problems, current natural language service composition approaches impose restrictions like: a narrow dictionary (a subset of natural language), the use of keywords as boundaries for different sentence blocks, a specific grammatical structure (e.g a verb and a noun).

Service composition based on restricted natural language is not convenient for the end user. In order to keep the format of the request unrestricted, we need a more reliable way to deal with the ambiguous nature of the language. When using the term *ambiguity* in the context of this paper, we refer to inaccurate (inexact) or incomplete user requests.

This paper proposes a service composition system that is able to detect and deal with the imperfections of a natural language user request while keeping it unrestricted, in the context of current Web services technologies. The next section presents the proposed solution, while section 3 shows the evaluation of our implementation. In Section 4, we discuss related approaches. Finally, the conclusions and perspectives are summarized in Section 5.

P.P. Maglio et al. (Eds.): ICSOC 2010, LNCS 6470, pp. 670–677, 2010.

2 Proposed Solution

Our service composition system, called *NLSCd - Natural Language Service Composer with request disambiguation* is the second release of NLSC [1]. Its architecture is depicted in Fig. 1.

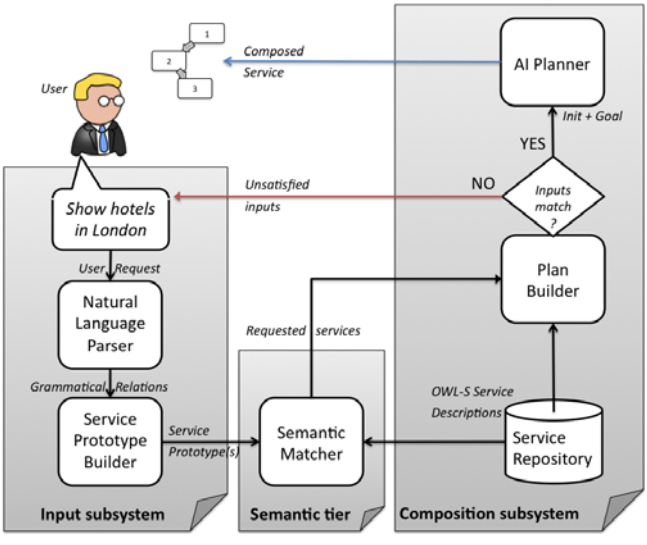

Fig. 1. NLSCd architecture

The **input subsystem** receives a natural language query from the user. A *natural language parser* extracts grammatical relations (GR) between the parts of speech in the request. Based on GRs, the *Service Prototype Builder* transforms the user input into a machine readable, formal request - a list of *service prototypes*.

A **semantic tier** is used as a bridge between natural language and the services ontology. In order to identify actual Web services based on their natural language prototypes, we use the *Semantic Matcher*. Services in the *Service Repository* are annotated using OWL-S [3].

Composition subsystem. The list of actual services identified by the matcher represents the input for the *Plan Builder*. First, the *Plan Builder* finds other services, on which required services depend on. Second, it matches the input values of the prototypes with the inputs of actual services. Values for the inputs that can't be satisfied is requested from the user. Finally, an initial state and a goal is defined, which the *AI Planner* uses to generate the composed service.

2.1 From a Natural Language Request to a Service Prototype

NLSCd's input subsystem accepts imperative natural language requests. According to Portner [4], an imperative construct can be modeled as function which

assigns a set of imperative denotations (a To-Do List with properties P) to the addressee (A): *Requiring TDL(A) U* $\{P\}$.

In service composition, the addressee is implicit (the computer), and each property P describes a service (process) to be invoked. Current natural language interfaces rely on parts of speech (mainly verbs and nouns) to extract such properties. But in cases of more complex user requests, parts of speech by themselves are not sufficient to correctly identify services. For example, in a request with two or more verbs and several nouns, it would be impossible to find which noun is related to each verb without taking into account the syntax of the sentence.

In a sentence, the action is expressed by a predicate, which must contain a verb. Other parts of speech (direct/indirect/prepositional objects, complements, adverbials, etc.) may complete the predicate of the sentence. To identify the parts of speech and the grammatical relations between them, we use a natural language parser called RASP - Robust Accurate Statistical Parser [2].

First, the parser finds the predicate of the sentence, which becomes the root node in a grammatical relations (GR) tree. Then, parts of speech (PoS) that are related to the predicate are added as leaves to this tree. The process continues for parts of PoS related to the ones added in the previous step, and so on.

For example, the predicate of the sentence *Print directions from Paris to London* contains the verb *print* and its direct object (**dobj**) is the noun *directions*. Prepositions *to* and *from* complete the predicate *print*, while themselves are completed by the nouns *Paris* and *London*, therefore these nouns are indirectly related to the predicate.

The GR tree is used by the *Service Prototype Builder* component to generate *service prototypes*. A service prototype corresponds to a property P in the request function and consists in a *process name* and its *input values*.

To create prototypes, the *Service Prototype Builder* component finds *dobj* (direct object) relations that include the predicate of the sentence. Then, the predicate and its direct object represent the process name. Other PoS that are related to either the predicate or its direct object (except other predicates) represent inputs for the process. For the previous example, the process is *print directions* and its inputs are *Paris* and *London*.

2.2 Semantic Matching

Information about services is published at a service registry. In Service Oriented Computing, the act of locating a machine-processable description of a resource that can be reached through a service that may have been previously unknown and that meets certain functional criteria is called *service discovery*. In essence, the discovery activity is comprised of matching the requirements of a service requester with the properties of a service provider.

Services are annotated using OWL-S [3], which is a standard ontology for describing the semantics of Web services. In order to find concrete services based on the user request, the concepts in the OWL-S service profiles are matched against their natural language service prototypes. Semantic matching evaluates

the semantic similarity between a prototype and a concrete service. The metric we use, called the *conceptual distance*, was presented in paper [1].

The *conceptual distance* is a measure of the relatedness between two concepts. It is based on the WordNet [6] lexical database for the English language. The distance is zero for two synonyms and increases for remotely related terms. The use of related terms compensates for the **inexact** (inaccurate) user requests. Based on the conceptual distance, the distance between a prototype and a concrete service is evaluated as follows:

$$\frac{1}{2}min\{D(Ppn, Sn), D(Ppn, Pn))\} + \frac{1}{4}D(Ppi, Pi) + \frac{1}{4}(1 - \frac{min(Npp, Np)}{max(Npp, Np)}) \quad (1)$$

where *Ppn* is the prototype process name, *Sn* is the OWL-S service name, *Pn* is the OWL-S process name, *Ppi* are the prototype input names, *Pi* are the OWL-S input names, *Npp* is the number of prototype inputs, *Np* is the number of the actual service inputs.

For example, let's consider that the service prototype is *reserve flight (John Doe)*, where *reserve flight* is the prototype process name and *John Doe* and is the prototype process input. Also, the service repository contains 3 services:

1. Book Flight Service: Book Flight Atomic Process (Customer, Account Data, Flight), where the OWL-S service name is *Book Flight Service*, the OWL-S process name is *Book Flight Atomic Process* and the process input names are: *Customer, Account Data, Flight*

2. Book Flight Service 2: Book Flight Atomic Process (Flight, Account Data)

3. Book Medical Flight Service: Book Medical Flight Atomic Process (Attendant, Patient, Flight)

The distances for the prototype and named services are: D1 = 0.516, D2 = 0.75 and D3 = 0.75. The first service is the match for the given prototype. Even though the request is inexact, the matcher identifies the correct service as the verb *to book* is a related term to the verb *to reserve*. Also, *John Doe* is a *Person*, which is a related term to *Customer* and *Attendant*, therefore they match better than with other inputs.

2.3 Service Composition and Dealing with Request Incompleteness

Having identified the services requested by the user, a composed service can be created. Since generating workflows would be practically impossible without control constructs, we chose to use AI planning to keep the format of the user request unrestricted. But, in order for a planner to function, it requires complete knowledge of the world - a description of the initial state and goal, the set of services.

To specify the initial state and the user goal, first we determine the list of all services required for the composition. Semantic matching only retrieves services that are indicated by the user. But these services may require data provided by other services. For example, the *flight booking* service may require a valid account, therefore the *create account service* (which was not mentioned in the user request) needs to be invoked first.

To solve this, we use XPlan's [7] OWL reasoner. In AI planning, the reasoner is used to store the world state, answer the planner's queries regarding the evaluation of preconditions, and update the state when the planner simulates the effects of services. [8] When the complete list of required services is available, their inputs are matched against the inputs from the service prototypes. An input can have one of two sources: a value from the user, or the result of invoking another service. The value from the user can be part of the user request or acquired based on the user context / profile. For simplification, we assume that the user request is the only source of values for user provided inputs.

First, the inputs that result from invoking other services are resolved. What remains after this step is the list of inputs that need to be satisfied by the user. These are compared to the types and values of the service prototype inputs. If all the inputs are satisfied, the initial state and the user goal are created and provided to the planner for composition. Unsatisfied inputs, if any, are caused by **incomplete requests**. In a case when such a request is detected, the user is prompted for the value of unsatisfied inputs. This way, he is engaged in a dialog with the system, which keeps the service composition process intuitive and unrestricted.

3 Evaluation and Results

We approached two aspects of ambiguity - incompleteness and inexactness - for which we used two separate methods. Therefore, a separate evaluation is required for each of these methods. Instead of creating a new ontology from scratch, we extended the Health-SCALLOPS (H-S) ontology, that was proposed for evaluating XPlan. We chose this ontology, because it already provides support for various transportation services.

For the evaluation of semantic matching, we used 23 most common user queries, some that request a transportation service, others completely out of the context of the H-S ontology. Prototypes based on five of these queries and their similarity with services in the repository are summarized in Table 1.

The smallest semantic distance corresponds to a match. Matches for 95% of tested prototypes were correctly identified. One particular case is that of the prototype *create account (John)*, when the semantic matcher finds two candidate services. Situations like this are solved by picking the service that has the minimum *D(Ppn, Sn)* from equation (1).

For the evaluation of service composition we used the same set of user queries. Following, we present a scenario for the where the user request is *Book medical flight from Paris to London*. Based on this request, the service prototype is *book medical flight (Paris, London)*. The matching service in this case is *Book Medical Flight Service*.

The OWL reasoner selects 3 extra services for composition: *Find Nearest Airport Service*, *Propose Medical Flight Service* and *Create Medical Flight Account Service*. Following inputs need to be satisfied: *1.* Provided Flight (isBookedFor : Person, hasDepartureLocation : Location, hasDestinationLocation : Location) and *2.* Person (hasAddress : Address).

Table 1. Similarity matching results

Service URI \ Service Prototype	*book flight alternative (John)*	*book flight (Paris, London)*	*create account (John)*	*find airport (Paris)*	*transport person (John)*
../BookFlight.owl	0.458333333	**0.516666667**	0.683333333	0.783333333	0.495833333
../BookFlight2.owl	0.458333333	0.75	0.833333333	0.789583333	0.633333333
../BookFlightAlternative.owl	**0.404761905**	0.75	0.833333333	0.789583333	0.633333333
../BookMedicalFlight.owl	0.526785714	0.708333333	0.8125	0.7875	0.625
../BookMedicalFlight2.owl	0.609375	0.732142857	0.8125	0.8125	0.651785714
../CreateFlightAccount.owl	0.583333333	0.791666667	**0.666666667**	0.808333333	0.6875
../CreateFlightAccount2.owl	6.61E-01	0.80952381	0.69047619	0.833333333	0.672619048
../CreateMedicalFlightAccount.owl	0.583333333	0.791666667	**0.666666667**	0.808333333	0.697916667
../CreateMedicalFlightAccount2.owl	0.680555556	0.80952381	0.69047619	0.833333333	0.681547619
../CreateMedicalTransportAccount.owl	0.722222222	0.863095238	0.69047619	0.833333333	0.645833333
../CreateVehicleTransportAccount.owl	0.722222222	0.863095238	0.69047619	0.833333333	0.645833333
../CreateVehicleTransportAccount2.owl	0.722222222	0.916666667	0.833333333	0.833333333	0.731770833
../FindNearestAirport.owl	0.895833333	0.7	0.95	**0.75**	0.6
../ProposeFlight.owl	0.5625	0.591666667	0.758333333	0.802083333	0.558333333
../ProposeFlight2.owl	0.601190476	0.8125	0.833333333	0.808333333	0.645833333
../ProposeMedicalFlight.owl	0.601190476	0.8125	0.833333333	0.808333333	0.6875
../ProposeMedicalFlight2.owl	0.708333333	0.827380952	0.833333333	0.833333333	0.672619048
../RegisterPersonWithMedicalTransport.owl	0.736111111	0.880952381	0.833333333	0.833333333	0.606547619
../RegisterPersonWithTransport.owl	0.625	0.666666667	0.708333333	0.786458333	0.420833333
../RequestMedicalTransport.owl	0.654761905	0.875	0.833333333	0.795833333	0.614583333
../RequestTransport.owl	0.583333333	0.666666667	0.708333333	0.786458333	**0.470833333**

The *isBookedFor* field of the *Provided Flight* input is satisfied by the *Person* input. The remaining fields are compared to the service prototype input values: *hasDepartureLocation, hasDestinationLocation, hasAddress*. But the service prototype has only two inputs: *Paris, London*. Since input matching also uses the conceptual distance, and *location* and *address* are synonyms and *Paris* and *London* can be the value of both an address and a location, the system assigns *Paris* to the *hasDepartureLocation* field and *London* to the *hasDestinationLocation* of the *Provided Flight* input, then it asks the user to provide the value of the *hasAddress* field for the *Person* input. The composition plan the AI planner generates is shown in figure 2.

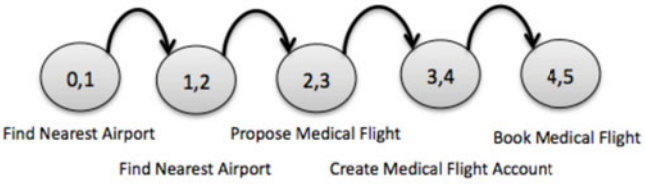

Fig. 2. The composition plan

4 Related Work

Several natural language service composition systems have been proposed in literature. The most relevant solutions related to our work are discussed below.

A solution based on restricted natural language and sentence templates. The system described in [5] assumes that the user request is expressed using a controlled vocabulary and is according to patterns like: *if ... then ... else*, *when ... do* and others. Based on verbs and patterns, the user request is transformed into a flow model. The major limitation of this solution derives from the restrictions imposed on how the user request is formulated. Another limitation is that, according to this approach, it is not clear how to create a new service if the user request specifies only a goal, without indications about how to achieve it.

Constructing workflows from natural language requests. Lim & Lee [9] propose a method for extracting a workflow from a natural language request. First, the user sentence is split into blocks based on control constructs (*if, then, else, etc.*) and verbs. Workflow templates are extracted by applying basic workflow patterns to control constructs. Second, for each sentence block, the service type and a list of candidate services are determined. Third, a service is selected from the candidate service list by calculating a similarity with the sentence block. Finally, an abstract workflow is generated by combining the workflow templates and the selected services. Control constructs and the lack of a semantic similarity measure (services need to be specified using words exactly as in their ontology) are the main disadvantages of this solution.

A solution based on AI Planning. The solution called SPICE ACE (Automatic Composition Engine) [10] is proposed as a development and a deployment platform for value-added services. The user request is transformed into a formal service request, specific to the ACE platform, a process which is not detailed in the paper. A causal link matrix models all possible interactions between all the known Web services as semantic connections. A semantic connection or a causal link consists of a triple: the output, the input and the semantic meaning of the link. An ontology is used to infer concept matching. The backward chaining technique is used for service composition: a plan of Web services is generated for finding an optimal plan among various compositions. The main drawbacks of this approach derive from the proposal's initial assumptions: a) the set of Web services is closed; situations like non-determinism, implicit goal, fuzzy service description are not considered.

5 Conclusion and Perspectives

In this paper, we propose an efficient method to compose a service, based on natural language requests. We focus on dealing with inexact and incomplete requests, while keeping them unrestricted. Inexactly named services are discovered using a semantic matcher that is able to measure the similarity between a natural language term and a concept from an ontology. Taking into account preconditions and effects, a reasoner discovers services related to those pointed by the user. When invoking these services, if missing inputs are detected, as in an incomplete request, the user is prompted to provide the required values.

The original aspect of our work is that by dealing with inexact and incomplete user requests, we manage to keep the query unrestricted in the context of current

Web service standards. Also, the user is actively involved into the composition process, through a dialog with the machine.

A few assumptions have been made: the user request is a form of imperative, a service prototype only matches a service from the repository and a contextual source of inputs is out of scope. The first two assumptions are justified, as most English requests are a form of imperative, and additional input sources can be implemented with ease.

The condition that a service prototype only matches a service from the repository is the major limitation of our system. Our semantic matching algorithm is intended to compensate for inexactness, which requires a compromise between a relaxed similarity metric and a small number of matches. The number of matches is linked to the number of candidate services. Instead of using a single candidate, the system could provide the user with a list of potential candidates to select from. But this is a challenge we intend to approach in the future.

Acknowledgments. This work was supported by CNCSIS-UEFISCSU, project code 1062/2007 and 1083/2007 and Brancusi PHC 301 project.

References

1. Cremene, M., Tigli, J.Y., Lavirotte, S., Pop, F.C., Riveill, M., Rey, G.: Service composition based on natural language requests. In: IEEE International Conference on Services Computing, pp. 486–489 (2009)
2. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., et al.: OWL-S: Semantic Markup for Web Services (2004)
3. Portner, P.: The semantics of imperatives within a theory of clause types. In: Young, R.B. (ed.) Proceedings of SALT XIV, pp. 235–252. CLC Publications, Ithaca (2004)
4. Briscoe, T., Carroll, J., Watson, R.: The second release of the rasp system. In: Proceedings of the COLING/ACL on Interactive presentation sessions, Morristown, NJ, USA. Association for Computational Linguistics, pp. 77–80 (2006)
5. Christiane Fellbaum, E.: WordNet An Electronic Lexical Database. The MIT Press, Cambridge (1998), `http://wordnet.princeton.edu/`
6. Klusch, M., Gerber, A.: Evaluation of service composition planning with owls-xplan. In: WI-IATW 2006: Proceedings of the 2006 IEEE/WIC/ACM international conference on Web Intelligence and Intelligent Agent Technology, Washington, DC, USA, pp. 117–120. IEEE Computer Society, Los Alamitos (2006)
7. Sirin, E., Sirin, E., Parsia, B.: Planning for semantic web services. In: Semantic Web Services Workshop at 3rd International Semantic Web Conference (2004)
8. Bosca, A., Corno, F., Valetto, G., Maglione, R.: On-the-fly construction of web services compositions from natural language requests. JSW 1(1), 40–50 (2006)
9. Lim, J., Lee, K.H.: Constructing composite web services from natural language requests. Web Semant 8(1), 1–13 (2010)
10. Lécué, F., da Silva, E.G., Pires, L.F.: A framework for dynamic web services composition. In: WEWST (2007)