Combining configuration and query rewriting for Web service composition

Amin Mesmoudi
Université de Lyon, CNRS
INSA-Lyon, LIRIS, UMR5205, F-69622, France
amin.mesmoudi@liris.cnrs.fr

Michaël Mrissa and Mohand-Saïd Hacid
Université de Lyon, CNRS
Université Lyon 1, LIRIS, UMR5205, F-69622, France
{michael.mrissa,mshacid}@liris.cnrs.fr

***Abstract*—In this paper, we investigate the combination of configuration and query rewriting for semantic Web service composition. Given a user query and a set of service descriptions, we rely on query rewriting to find services that implement the functionalities expressed in the user query (discovery stage). Then, we use configuration to capture dependencies between services, and to generate a set of composed Web services described as a directed acyclic graph, while maintaining validity with respect to business rules (orchestration stage). Finally, we propose a semantic ranking algorithm to rank results according to user preferences (classification stage).**

The techniques used in our approach take into account the semantics of concepts utilized to describe the elements (services, business rules, query and user preferences) involved in the composition process. We provide a formal approach and its implementation, together with experiments on Web services from different application domains.

I. Introduction

Domain and service ontologies enable the annotation of Web service description files with respect to several aspects (functionality, input and output data, quality of service, etc.). Domain ontologies formalize the vocabulary of some application domain (medical, e-business, etc.) whereas service ontologies formalize the vocabulary used for describing operational service aspects (operation, input, output, protocols, etc.). Thanks to these underlying semantic representations and to the reasoning possibilities they offer, semantic annotation allows to automatically handle service-related tasks (e.g. discovery and composition) with the help of reasoning mechanisms (i.e. logical reasoning or "inference") [1].

In a service ontology, we can distinguish between two kinds of Web services: abstract and concrete services. Abstract services play the role of interfaces (as in object-oriented programming), while concrete services implement the abstract services [2]. However, the most known service ontology OWL-S [3] is bound to concrete services, and does not offer the logical constructs to describe abstract services in a flexible way [4].

Composing Web services allows to provide novel and complex functionalities. However, identifying existing services (discovery) and combining them (orchestration), in order to fulfill the requirements for a complex service, is extremely time-consuming for a human operator. There are many approaches devoted to automated synthesis of service composition [5]. Several techniques exist to compose Web services, mainly variants of planning [6]. However, these techniques mainly focus on finding the execution plan, relying on a predefined template [7]. A template represents the dependency between different kinds of abstract Web services. This template should be instantiated, by selecting concrete Web services, to achieve the composition goal. So far, this template is created manually (see, e.g., [8]), created semi-automatically (see, e.g., [9]) or generated using logs (execution traces) (see, e.g. [7]).

Our contribution in this paper is two-fold:

First, we use a language for describing abstract services, while remaining independent from any underlying service ontology that providers could use for describing services at the concrete level (OWL-S or others).

Second, we propose a three-stage approach to automatically generate a Web service composition template. We rely on configuration and query rewriting techniques. The stages can be described as follows: (1) finding the services that implement the functionality of each goal in the query (discovery stage), (2) orchestrating the interactions between selected Web services in order to achieve the composition and to fulfill user's requirements (such as constraints) (orchestration stage) and (3) ranking candidate solutions according to users' preferences (classification stage).

In this approach, we use the semantic annotation of services to automatically generate, without user intervention or logs analysis, composition template from the user requirements (such as preferences and constraints).

This paper is organized as follows: in Section II, we provide some background knowledge on query rewriting and configuration. We highlight the need for a combination of configuration and query rewriting techniques in Section III. We present the formal aspects of our approach in Section IV, and we discuss the implementation in Section V. Section VI discusses the advantages and limitations of our approach, and Section VII gives some directions for future work.

II. Background knowledge and Related Work

In this section, we discuss the main approaches regarding the use of *query rewriting* and *configuration* for Web service composition. We also highlight our contribution in this area.

A. Service ontology

As mentioned in the previous section, we can distinguish between abstract and concrete Web services, as described in [2]. The most known service ontology, OWL-S [3] is not relevant for describing abstract services. More precisely, OWL-S is bound to concrete services, and does not offer the logical constructs to describe abstract services in a flexible way [4]. On the contrary, another well-known service ontology is WSMO [10], which has a capability to describe abstract services as defined above. However, to the best of our knowledge, no decidability results are known for basic reasoning task (such as structural subsumption) for services described in WSMO.

In our work, we propose a language for describing service functionalities at the abstract level, relying on description logic, while remaining independent from underlying service ontologies that providers use for describing services at the concrete level (OWL-S or others).

B. Semantic Web service composition

Raising the Web service composition problem to the semantic level offers new opportunities for the automation of the composition and discovery tasks. Several approaches have investigated the use of AI planning to automate semantic Web service composition. The approaches in [11], [12] are based on a translation of the composition problem to planning as situation calculus problem. In [13], authors exploit planning as Model Checking. In [4], the authors proposed a hybrid approach based on HTN planning and OWL-DL reasoning. HTN enables task decomposition but is not semantic-aware, and OWL-DL allows to reason about Web services. In Lécué et al. [14], [15], the authors use AI planning and CLM (Causal Link Matrix) to tackle the semantic service composition problem. However, these techniques mainly rely on a predefined template to find a correct execution plan. A template represents the dependency between abstract Web services. To use a template, abstract services should be instantiated. So far, this template is created manually (e.g. [8]), semi-automatically (e.g. [9]) or generated using logs (execution traces) (see, e.g., [7].

In our work, we propose a novel approach to automatically generate a Web service composition template, without user intervention or logs analysis. Dong et al. [16] as well as Sirin et al. [4] proposed extensions to OWL-S to represent abstract services of the composition template. Our templates are described using a DAG (Directed acyclic graph), this description is free from concrete service description language. We consider preferences described by using terms from the service ontology, e.g. functionalities, inputs/outputs, etc., which have not been considered in previous works (see, e.g., [17], [18]) on preferences integration in the composition task. In the following, we review the main works related to query rewriting and configuration in order to compose Web services.

C. Query Rewriting

Query rewriting (using views) consists in reformulating a query according to views that are already available from the database. Query rewriting techniques have also been used for Web service composition in [19], [20], [21] where services are accessed via Datalog queries. Lu et al. [19] provide a framework for answering queries with a conjunctive plan that includes inputs and outputs of participating Web services. In Thakkar et al. [20], a combination of inverse rules algorithm and tuple-level filtering allows building the composition. However, in those works, Web services are matched without taking into account the semantic information contained in their descriptions. In Bao et al. [21], a new algorithm is proposed to construct a composite Web service i.e. generating the dependencies between already selected services, but the discovery phase is missing and the authors only deal with functional constraints. Also, Barhamgi et al. [22] use query rewriting for querying services described as RDF views with SPARQL queries, but this work is focused on data-providing services.

D. Configuration

Configuration has been part of the Artificial Intelligence (AI) field for a long time. Some attempts to formalize configuration have been proposed in several works (see, e.g., [23], [24]). Configuration consists in finding sets of concrete objects that satisfy the properties of a given model. With respect to Web service composition, some approaches based on configuration have been proposed (see, e.g., [25], [8]). These approaches rely on predefined process templates that allow selecting services that match the process template. In the following, we discuss our contribution and illustrate it on a typical scenario.

III. Description of the proposed approach

In this section, we develop the novel aspects of our approach for the generation of composition templates, and we motivate the choice of combining query rewriting and configuration. Our approach follows the well-known "separation of concerns" principle [26] and identifies independent, consecutive sub-problems (called composition stages) in the composition problem: discovery and orchestration. We rely on query rewriting for performing service discovery and on configuration for solving the orchestration problem. Furthermore, a classification stage enhances our proposal with ranking capabilities. Now, let us detail our contribution according to these three stages:

Discovery: In order to discover sets of services that answer a user query, we adapt the bucket algorithm used in query rewriting [27]. We propose two algorithms, the first one for identifying services according to the concepts in the query, and the second one for filtering irrelevant rewritings (missing outputs).

Orchestration: Orchestration consists in finding the correct order of invocation of services, while taking into account the business rules related to the application domain and the dependencies that exist between services. We categorize dependencies as follows: functional dependencies, i.e. dependencies between input parameters and output parameters, and non-functional dependencies, which depend on the business rules presented as a predefined set of constraints related to the domain of composition.

We rely on configuration to deal with the orchestration stage. We propose a formalization of the configuration problem where business rules are represented as constraints in the configuration task. Also, we propose a calculus based on inference rules.

Classification: We propose a ranking model that uses semantic information available in the service ontology to classify the results obtained after the composition, according to the user preferences. For each result we calculate a score that represents the semantic proximity between this result and user preferences. The ranking will be established using this score.

IV. Our Composition framework

A. Running Example

To illustrate our proposal, we use the traditional travel reservation process case study (see, e.g., [28]). A user who plans to travel to some country for a determined amount of time needs to book a transport, to find an accommodation, and to rent a car in order to visit some interesting places around. This example relies on a domain ontology available at http://liris.cnrs.fr/~soc/doku.php?id=transverse. We model user requirements for a composition with a query Q specified as a couple $< M, P >$, where M represents the *Mandatory* part of the query and P represents the *Preference* part of the query. Each part is specified as a triple $< I, O, C >$. With regards to M:

- I (for input) denotes the input data the user provides, which are handled as constraints in the query,
- O (for output) denotes required information to be provided as a result of the query,
- C denotes service categories, representing required functionalities.

In our example, I includes departure and return dates and locations, and O includes details about flight, train or bus tickets, hotel, flat or B&B information and type of vehicle and price and C includes transport, accommodation and vehicle rental. According to our query representation and given some user input I, the objective is to provide all the information required in O, by finding an appropriate combination of Web services that only make use of the input I specified in the query. P represents preferences on inputs(I), outputs(O) and category of services(C). Details and an example about preferences are given in Section IV-E.

Several services can satisfy the required functionalities, for instance the transport functionality can be satisfied by three services: Flight booking, Train booking or Bus booking. Therefore, an agent must consider all the possible combinations, then check their validity with respect to user constraints, such as “select service that can provide hotel description“, and service constraint, for example “a hotel booking service requires a reservation date to confirm booking for a client”. Even with valid combinations, we need to establish the order of invocation of services with respect to business rules governing the application, for example we cannot book a hotel without booking the flight, or we can use one service if we pay by credit card. One can show that it is very difficult for a human agent to do all these operations in the case of a large number of services provided and a complex user query. We consider this problems in the rest of paper.

B. Defining a WS description language

In this section, we define the kind of semantic Web services we consider. We also give an informal introduction to the knowledge representation language we use.

- A **semantic Web services database** $\mathcal{O}_\mathcal{T}$ describes the structural part of services, i.e. abstract services.
- A **service** S is composed of a set of input parameters (I_S), a set of output parameters (O_S) and represents an abstract function.

In our case study, we assume the existence of eleven categories of Web services in the application domain (e-tourism). For instance, hotel, flat, B&B and youth hostel reservation services are subcategories of the Accommodation category.

1) The ontology part: To describe the constrained vocabulary that will be used to specify $\mathcal{O}_\mathcal{T}$, we resort to description logic. The constructs of this description logic include Conjunction, Existential and Universal quantification. Axioms in $\mathcal{O}_\mathcal{T}$ are of the form $A \sqsubseteq D$ where A is a primitive concept and D is an arbitrary concept. Roughly speaking the language corresponds to $\mathcal{FLE}$ description logic. The complete syntax, semantics together with decidability of concept entailment are described in [29].

For example, in our case study, we represent the subcategory *Plane* of the category *Transport* with $Plane \sqsubseteq Transport$, the input *departurePlace* of the service *Transport* with $Transport \sqsubseteq \exists\ HasInput.departurePlace$ and the output *transportPrice* of the service *Transport* with $Transport \sqsubseteq \exists HasOutput.transportPrice.$

C. Query Rewriting

We assume that we have a **query language** $\mathcal{L}$ to specify queries. The construct of $\mathcal{L}$ are given in the examples below. Each part (M and P) of the query Q is defined as a conjunction of terms. Each term is a concept expressed in $\mathcal{L}$ over the ontology $\mathcal{O}_\mathcal{T}$. We assume that $\mathcal{L}$ is a subset of the language used to describe $\mathcal{O}_\mathcal{T}$ and presented in Section IV-B1. In this section we focus on the mandatory part of the query. The preference part is considered in Section IV-E.

We identify three types of concepts in the Mandatory part of a query: **inputs**, **outputs** and **service categories**. Inputs have their values provided by the user as query parameters. Outputs must be provided as an answer to the query execution, and service categories represent functionalities to be selected. We use Q to refer to the mandatory part of a user query.

To make things simple, we define Q_{cat} as the service category part of the query and we will use Q_{Cons} to denote the constraint part. Hence, in this context query rewriting consists in finding Web services belonging to the relevant categories (i.e. resolve the Q_{cat} part of the query), and that satisfy the query by: 1) providing the required output, and 2) requiring overall no more data than those provided as inputs (i.e. resolve the Q_{Cons} part of the query). Let us consider the following query:

$$\begin{aligned} Q = \quad & Transport \sqcap \exists HasInput.departurePlace \\ & \sqcap \exists HasInput.destinationPlace \sqcap \exists HasInput. \\ & departureDate \sqcap \exists HasOutput.transportPrice \sqcap \\ & Accommodation \sqcap \exists HasInput.checkoutDate \sqcap \\ & \exists HasOutput.accommodationPrice \sqcap \\ & \exists HasOutput.accommodationDescription \\ & \sqcap CarRental \sqcap \exists HasOutput.rentalPrice \\ & \sqcap \exists HasInput.retrievalDate \sqcap \exists HasInput.returnDate \\ & \sqcap \exists HasOutput.rentalDescription \end{aligned}$$

The inputs specified in query Q are $departurePlace$, $destinationPlace$, $departureDate$, $checkoutDate$, $retrievalDate$ and $returnDate$. Accordingly, the outputs expected as a result to the query are $transportPrice$, $accommodationPrice$, $accommodationDescription$, $rentalPrice$, and $rentalDescription$. We consider a particular class of abstract services, namely primitive services, that represents services that can have a concrete service as instance and does not have abstract services as subcategories. For our example: Plane, Train, TouristCarRental constitute a primitive services.

In order to rewrite the query we use a modified version of the bucket algorithm presented in [27]. The bucket algorithm allows to rewrite a user query according to existing views.

In order to rewrite a query Q, the bucket algorithm starts by creating a bucket for each subgoal containing the views that are relevant. Then, it considers the conjunction of the different views in each bucket, and finally applies filtering mechanisms in order to build the rewriting. The reader may refer to [27] for more details.

We build our proposal on an analogy between the bucket algorithm and the Web service composition problem. In our proposal, **views** correspond to *primitive services*, *predicates* to *concepts* and *subgoals* to *service categories*. Views in the original bucket algorithm correspond to primitive services in our context, and they are associated with constraints related to the service.

The propagation rule given in Algorithm 1, where C,D and S are concepts in the ontology such that $D \sqsubset S$ is an element of the ontology, is first applied. We denote by L_C the set of all the leaves (primitive services) that belong to the category C. For example, for the category *Transport*, $L_C = \{Plane, Train, Bus\}$. We assume that Q_{cat} is not empty, which means that at least there is one C in the query Q. Here are some explanation of Algorithm 1:

Algorithm 1 Propagation rule

```
1: for all $C$ in $Q$ do
2:     $L_C = \{C\}$
3:     for all $S$ in $L_C$ do
4:         for all $D \sqsubset S$ in $\mathcal{O}_\mathcal{T}$ and $D \notin L_C$ do
5:             $L_C = L_C \cup \{D\}$
6:         end for
7:         if $\exists D \sqsubset S$ in $\mathcal{O}_\mathcal{T}$ then
8:             $L_C = L_C \setminus \{S\}$
9:         end if
10:    end for
11: end for
```

- Line 2: we initialize every L_C with one category element C.
- Lines 3-10: we collect in each set L_C primitive services.
- Lines 3-6: we add the subconcept D of a service S to L_C if it is not yet in L_C,
- Lines 7-9: we remove the service S from L_C if this concept has at least one sub-concept.

Then, we generate the bucket table as the Cartesian product of all L_C generated from Algorithm 1. Let $L = \bigcup_C L_C$ and let BC be the set of all possible rewritings of Q_{cat}, then $BC = \prod_{l \in L} l$. To verify the subsumption, we make use of a DL reasoner based on tableau calculus (see, e.g., [30]). In the implementation we make use of Pellet [31].

At the end of the process, several combinations of services will satisfy the Q_{cat} part of the query, which means that the selected services satisfy the query in terms of functionality. Each Row of BC is a possible rewriting of the query Q. Cells describe primitive services. Hence, each row contains a combination of Web services that fulfil the Q_{cat} part of the query. In the sequel, we use Q^c_{cat} to denote the fact that the service category c is an element of Q_{cat}.

To each row of the table, we apply Algorithm 2. Its primary goal is to filter irrelevant combinations of services that do not provide all the required output parameters specified in Q. In addition, it identifies inputs that services require and that are not provided in Q (MI).

We denote by D the primitive services used to rewrite Q. For each service D, we define its inputs as D^i_{cons} and its outputs as D^o_{cons}. We assume that all the outputs in the request are missing and every time we find a service that provides an output that is in the request, we remove it from the set of missing outputs denoted by MO. At the end of processing of each line, if there are missing outputs, we remove this line from the table because this set of services does not provide the required outputs. Finally, we add one column to the BC table to represent MI.

Algorithm 2 I/O algorithm

```
1:  for all row L in BC do
2:      $MI = \emptyset, MO = Q^o_{cons}$
3:      for all service D in $L_s$ do
4:          $MI = MI \cup \{D^i_{cons}\} \backslash Q^i_{cons}$
5:          $MO = MO \backslash \{MO \cap D^o_{cons}\}$
6:      end for
7:      if $MO \neq \emptyset$ then
8:          some output is missing: invalid combination
9:          remove the line from the table
10:     else
11:         record MI
12:     end if
13: end for
```

D. *Configuration*

The configuration task consists in establishing the order of invocation of services with respect to business constraints that usually govern application domains. Constraints include dependency relationships between Web service invocations, data and control flow constraints such as "CarRental can only be validated if the flight is booked", etc.

1) Formal Representation of Composition Constraints: To represent the constraints involved in the configuration, we use three sets D, I, E such that:

- D represents a set of dependency relationships related to the domain of composition, $D \subseteq S \times S$, $(s1, s2) \in D$ denotes that the execution of s1 must precede the execution of s2.
- I represents a set of incompatibility relationships, $I \subseteq S \times S$, $(s1, s2) \in I$ denotes that it is strictly forbidden to put s1 and s2 in the same composition.
- E represents a set of requirement relationships, $E \subseteq S \times S$, $(s1, s2) \in E$ denotes that it is mandatory to find the service s1 in a composition involving s2.

2) A Calculus for Configuration: The configuration task is defined as a deductive reasoning task that uses business rules represented as constraints between services to provide a dependency graph G that represents the dependencies between services that form the composition template. We define a dependency graph G as a tuple $G = (V, R)$, where V is a set of services involved in the composition, $R \subseteq V \times V$ represents a set of dependencies between services, $(s1, s2) \in R$ denotes the fact that a service $s2$ cannot be invoked before the end of the execution of the service $s1$.

The goal specification in our configuration task is represented as a tuple (S, MI), where S is a set of services to be composed and MI is a set of missing inputs represented as a set of functional constraints. One rewriting of the query example is represented as follows:

$(S, MI) = (\{Plane, Hotel, TouristCarRental\}, \{location, chekinDate\})$

Additionally, the dependency relationships for the e-tourism domain are :

$D = \{(Transport, CarRental), (Transport, Accommodation), (Accommodation, CarRental)\}$

Our inference rules work on a pair of sets C.G where C is the specification of the goal and G is the solution of the configuration represented as a graph. We start with the initial goal and the empty solution (C.($\emptyset$,$\emptyset$)), and the algorithm **ends** when no more inference rule applies. In order to simplify the definition of rules we use the following notations (with $(s, d) \in S \times S$):

- A **functional dependency** is denoted by $(s, d)_i$ where $i \in (s^i_{cons} \cap MI)$ and $\exists o \in d^o_{cons}$ s.t. $o \sqsubseteq i$.
  This dependency reflects the relationships between service inputs and outputs.
- An **induced dependency** is denoted by $D(s) = \{(s, d)/ \ \exists d \in S \cup V$ and $\exists (x, y) \in D$, s.t. $s \sqsubseteq x$ and $d \sqsubseteq y\}$. This set reflects the dependencies of a service s deduced from the set D of dependencies of a domain.
- A **path of indirect dependency** is referred to by $Path(x, y) = \{(x, x_1), (x_1, x_2), ..., (x_{i-1}, x_i), (x_i, y)\}$ with $x \neq x_1$ and $x_i \neq y$
  This set reflects an indirect dependency relationship between two services.

Now we present the rules used in the configuration phase. In the following $\acute{R}$ denotes the changed/new set of dependencies:

1) $(S \cup \{s\}, MI).(V, R) \rightarrow (S, MI).(V \cup \{s\}, \acute{R})$ with $\acute{R} = R \cup D(s)$ and $D(s)$ is the set of induced dependencies related to the service s.
2) $(S, MI \cup \{i\}).(V.R) \rightarrow (S, MI).(V, \acute{R})$ iff $\exists \{s_1, s_2\} \subseteq V$ with $(s_1, s_2)_i$
   $\acute{R} = R$ if $(s_1, s_2) \in R$ else $\acute{R} = R \cup (s_1, s_2)$
3) $(S, MI \cup \{i\}).(V, R) \rightarrow (S, MI).(V, R)$ if $\exists \acute{i} \in Q^i_{cons}$ with $\acute{i} \sqsubseteq i$
4) $(\emptyset, MI).(V \cup \{x, y\}, R) \rightarrow (\emptyset, MI).(\acute{V}, R)$
   $\acute{V} = V \cup \{x, y\}$ iff $\nexists (s_1, s_2) \in I$ with ($x \sqsubseteq s_1$ and $y \sqsubseteq s_2$) or ($x \sqsubseteq s_2$ and $y \sqsubseteq s_1$)
   $\acute{V} = V \cup \{\perp\}$ otherwise
5) $(\emptyset, MI).(V \cup \{x, y\}, R) \rightarrow (\emptyset, MI).(\acute{V}, R)$
   $\acute{V} = V \cup \{x\}$ iff $\nexists (s_1, s_2) \in E, \exists y \in V$ with ($x \sqsubseteq s_1$ and $y \sqsubseteq s_2$)
   $\acute{V} = V \cup \{\perp\}$ otherwise
6) $(S, MI).(V, R) \rightarrow (\acute{S}, \acute{MI}).(\acute{V}, \acute{R})$
7) $(\emptyset, MI).(V, R \cup Path(x, y) \cup (x, y)) \rightarrow (\emptyset, MI).(V, R \cup Path(x, y))$
8) $(\emptyset, MI).(V, R \cup Path(x, x)) \rightarrow (\emptyset, MI).(V \cup \{\perp\}, R)$

Let us give some intuition regarding the interpretation of these rules:

- Rule 1 simply expresses that each time we add a service to the graph of dependencies, we must add all dependencies related to this service, these dependencies must be related to another service in S or V.

- Rule 2 reflects the creation of dependency relationships between services.
- Rule 3 removes missing inputs provided by another part of the query.
- Rule 4 detects incompatibilities between services.
- Rule 5 controls the requirements of services.
- Rule 6 detects deadlocks between services (cycles in the solution graph).
- Rule 7 optimizes dependencies between services (removes useless dependencies).

The configuration task starts with $V = \emptyset$, $R = \emptyset$ and $S = L_s$, where L_s is the set of services induced by the specification at the row L of the bucket table. If the inference task ends with $MI = \emptyset$, and $\perp \notin$ V we conclude that the composition is **correct**. Otherwise, we delete the row L from the bucket table. If $MI \neq \emptyset$, there are missing inputs that cannot be provided in the composition, and if $\perp \in$ V, there is an inconsistency between services. Let us illustrate our approach with an example that is the previous example of the goal specification and business rules of the e-tourism domain:

- Initial state:
  $(S, MI) = (\{Plane, Hotel, TouristCarRental\}, \{location, chekinDate\})$
- Application of the Rule 1 to Plane, Hotel and TouristCarRental results in the creation of new dependencies
- Application of the Rule 2 to checkinDate means that this missing input can be provided from another service of the same composition
- Application of the Rule 3 to location means that this missing input can be provided by the user query
- Application of the Rule 4 remove useless dependencies
- After triggering these rules we obtain the following solution:
  $(S, MI) = (\emptyset, \emptyset)$
  $(V, R) = (\{$ $Plane$, $Hotel$, $TouristCarRental\}$,$\{$ $(Plane$, $Hotel)$, $(Hotel$, $TouristCarRental)$ $\})$

The inference mechanism performs the selection and the application of rules while keeping satisfied a predefined triggering order between rules. This order is specified as follows: $R7 \preccurlyeq R6 \preccurlyeq R5 \preccurlyeq R4 \preccurlyeq R2 \preccurlyeq R3 \preccurlyeq R1$. Here R1 has the higher priority, then R3, and so on. The result of the configuration phase is a DAG (Directed Acyclic Graph).

E. User preferences and Ranking

Integrating user preferences in composition allows us to obtain ranked results. In our work, we are interested in user preferences at the process level (such as "FlightService is preferred to BusService") and at the service input/output level (such as "we prefer a service that accepts credit card payment" for input preference, "we prefer a service that shows if a car has GPS" for output preference). Once the configuration task achieved, we obtain several results that can satisfy the M part of the query. Here, we propose to rank these results according to preferences expressed in part P of the query.

First, we define a "concept score" that is calculated for each concept in P and represents the degree of relevance between a composition and one concept in P. Next, a global score for the composition is calculated from individual concept scores. Our technique is inspired from the computation of geographical scores in [32]. We adapted this technique as it provides an interesting similarity with our work. Indeed, the computation of geographical scores relies on two measures: closeness and specificity. We noticed that the more specific a solution is, the better ; and accordingly, we noticed that it is interesting to calculate the closeness between concepts of the query and concepts of the composition in order to evaluate its relevance. The experiments are in favor of this assumption.

1) Concept score: A concept score represents the degree of relevance between a concept in P and the composition result. It is characterized by the following two elements:

Definition (Closeness) Semantic distance between the part P of the query and the concept.

Definition (Specificity) A weight discounting term based on the semantic extent of the concept.

The relevance $S(c, R)$ between a composition R and a preference concept c is calculated as follows:

$$S(c, R) = Closeness(c, R).Specificity(c) \tag{1}$$

In the next equation, V^i_{cons} and V^o_{cons} represent sets of inputs and outputs respectively related to the services in V.

$$Closeness(c, R) = \begin{cases} 1 & iff\ c \in V \cup V^i_{cons} \cup V^o_{cons} \\ & \quad with\ R = (V, \acute{R}) \\ \alpha & iff\ \exists y, y \sqsubset c\ such\ that\ y \in V \\ & \quad \cup V^i_{cons} \cup V^o_{cons} with R = (V, \acute{R}) \\ \beta & iff\ \exists y, c \sqsubset y\ such\ that\ y \in V \\ & \quad \cup V^i_{cons} \cup V^o_{cons} with R = (V, \acute{R}) \\ 0 & otherwise \end{cases} \tag{2}$$

α and β indicate the minimum score between two subsumed concepts in the ontology. In our evaluation ontology, we fixed α=0.7 and β=0.4. These values are chosen for the purpose of the tests we conducted. They can differ from a context to another.

$$Specificity(c) = 1/extent(c) \tag{3}$$

The function $extent(c)$ is the semantic extent implied by the concept c. It is related to the hierarchical position of the concept in the ontology, and intuitively measures the granularity of a domain concept. For example, let us take $X \sqsubseteq Y$ and $Y \sqsubseteq Z$, if X has no sub-concepts, the extent value of X is 1, for Y the extent value is 2 and for Z the extent value is 3. In the case of several extents for one concept, we take the greatest value.

Definition (Composition score)

We define the composition score $S(R, P)$ for a given composition result R using P as follows:

$$S(R, P) = \frac{\sum_{c \in P} S(c, R)}{|P|} \qquad (4)$$

After computing all the composition scores of candidate compositions we apply a descending sort and obtain the final (ranked) results.

V. Implementation

In this section, we describe our prototype implementation to generate composition template. We implemented three components: discovery, configuration and ranking.

With respect to the discovery component, we implemented the algorithms presented in Section IV-C. We relied on ANTLR[1] to implement a query parser. We used the Jena library[2] to access to service ontology.

With respect to the configuration component, we implemented the calculus defined in Section IV-D2, the goal is to generate a graph of dependencies between services, starting from a set of rewritings, the service ontology and a set of business rules formally described in Section IV-D1. As an implementation of this component we utilized the Drools[3] rules engine together with the Jena 2 library for ontology access.

With respect to the ranking component we implemented our ranking model defined in Sect IV-E with the Jena library to access to service ontology.

Our framework is an open source and available for download on the project website[4] under the GNU LGPL license. We provide a sample package that demonstrates how to use our framework with some scenarios, one of those scenarios is a GUI application that loads the ontology, executes user queries and displays results as directed graphs. We use the standard graphical Java components and JUNG (java universal network/graph)[5] framework to display graphs.

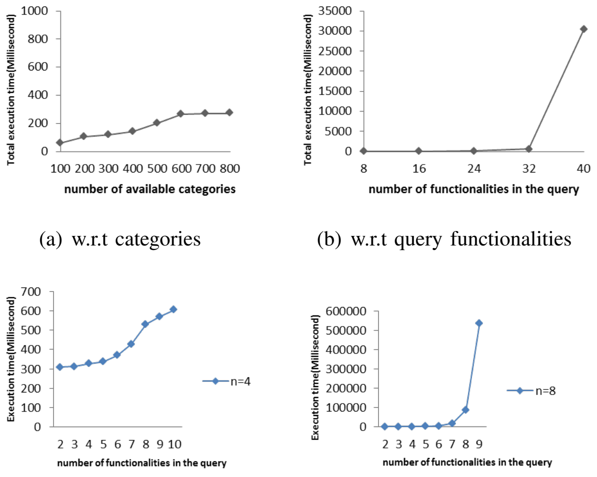
VI. Preliminary results

In this section, we describe our evaluation environment, show the different tests and discuss the interpretation of the obtained results.

A. Experimentation Setup

We conducted experiments to evaluate the approach. In the experiments, the computer used to run the prototype system has the following features: Intel® Core™ 2 CPU, 2.1 GHz with 1.5GB RAM, under Windows 7. In our testing phase, we used OWL-S TC[6], a collection of OWL-S descriptions. We generated an ontology that provides abstract descriptions for these services. We obtained an ontology with 2590 services, available on the project website.

[1]http://www.antlr.org/
[2]http://jena.sourceforge.net
[3]http://www.jboss.org/drools
[4]http://liris.cnrs.fr/~soc/doku.php?id=transverse
[5]http://jung.sourceforge.net
[6]http://projects.semwebcentral.org/projects/owls-tc/



(a) w.r.t categories (b) w.r.t query functionalities

(c) w.r.t query functionalities(n=4) (d) w.r.t query functionalities(n=8)

Figure 1. Evolution of the total execution time

B. Results

The objective is to investigate the influence of some parameters on the execution time of the composition process. Figure 1(a) shows the execution time of the composition task with respect to the number of categories available. In this experiment, we executed the same query and changed the number of available categories. Our test query contains five requested functionalities. One can see that the execution time does not depend directly on the number of categories available. The important change in this experiment is in the transition from 400 to 500 services and 500 to 600 services. This change is a result of the addition of services that have a direct relationship with the requested functionalities.

In a second experiment, we studied the impact of the number of functionalities in the query. Figure 1(b) shows this variation. We executed various queries on the same ontology. For each application we chose random functionalities. One can see that there is a minor change in execution time at the beginning and suddenly the composition time becomes drastic.

In a third experiment we studied the impact of the number of functionalities in the query and the number of primitive services related to a requested functionality in the composition time. Figure 1(c) and 1(d) show this variation. We did two experiments for different values of "n" the average number of primitive services related to a requested functionality. For each experiment we were interested in the variation of execution time versus the number of requested functionalities. In the first curve (n=4), one can see that the change in execution time is stable despite the regular increase of the number of functionalities. In the second curve (n=8), we can see that there is a stable change in execution time at the beginning and at one point the composition time explodes. This result is due to the use of Cartesian product in the rewriting algorithm. Other parameters have some impact on the execution time, for example the number of missing input for a rewriting, the size of business rules associated with a domain, etc.

VII. Conclusion

In this paper, we provided a framework that relies on the combination of query rewriting and configuration, together with a formal definition of the underlying languages and an implementation with a use case, in order to facilitate the composition of semantic Web services.

The main feature of the proposed approach is its construction as a three-stage process that relies on 1) a simple formalization of semantic Web services that supports query rewriting, and 2) a clear separation between constraints and service/domain knowledge description. Also, the proposed approach accommodates user preferences as part of the composition process.

There are many research directions to pursue. For example it could be interesting to generalize the use of business rules so that it can accommodate exceptions. Another important issue is to support approximate queries. In this domain, providing semantic similarity between composition remains challenging. These issues are currently being investigated.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

[1] M.-S. Hacid, F. Lécué, A. Léger, C. Rey, and F. Toumani, "Les web services sémantiques, automate et intégration i. introduction, éléments et scénarios, découverte de services web," *Technique et Science Informatiques*, vol. 28, no. 2, pp. 229–262, 2009.

[2] J. Yang, "Web service componentization," *Communications of the ACM*, vol. 46, no. 10, pp. 35–40, 2003.

[3] D. L. Martin, M. Paolucci, S. A. McIlraith, M. H. Burstein, D. V. McDermott, D. L. McGuinness, B. Parsia, T. R. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. P. Sycara, "Bringing Semantics to Web Services: The OWL-S Approach," in *SWSWPC*, ser. Lecture Notes in Computer Science, J. Cardoso and A. P. Sheth, Eds., vol. 3387. Springer, 2004, pp. 26–42.

[4] E. Sirin, B. Parsia, and J. Hendler, "Template-based composition of semantic web services," in *Aaai fall symposium on agents and the semantic web*, 2005, pp. 85–92.

[5] A. Marconi and M. Pistore, "Synthesis and Composition of Web Services," *Formal Methods for Web Services*, pp. 89–157, 2009.

[6] J. Rao and X. Su, "A survey of automated web service composition methods," *Semantic Web Services and Web Process Composition*, pp. 43–54, 2005.

[7] F. Lécué, Y. Gorronogoitia, R. Gonzalez, M. Radzimski, and M. Villa, "SOA4All: An Innovative Integrated Approach to Services Composition," in *Web Services (ICWS), 2010 IEEE International Conference on*. IEEE, 2010, pp. 58–67.

[8] Q. Sheng, B. Benatallah, Z. Maamar, and A. Ngu, "Configurable Composition and Adaptive Provisioning of Web Services," *IEEE Transactions on Services Computing*, vol. 2, no. 1, pp. 34–49, 2009.

[9] E. Sirin, J. Hendler, and B. Parsia, "Semi-automatic composition of web services using semantic descriptions," in *Web Services: Modeling, Architecture and Infrastructure workshop in conjunction with ICEIS2003*. Citeseer, 2003.

[10] S. Arroyo and M. Stollberg, "WSMO Primer. WSMO Deliverable D3.1, DERI Working Draft," WSMO, Tech. Rep., 2004, http://www.wsmo.org/2004/d3/d3.1/.

[11] S. Narayanan and S. McIlraith, "Simulation, verification and automated composition of web services," in *Proceedings of the 11th international conference on World Wide Web*. ACM, 2002, pp. 77–88.

[12] S. Sohrabi, N. Prokoshyna, and S. McIlraith, "Web service composition via the customization of Golog programs with user preferences," *Conceptual Modeling: Foundations and Applications*, pp. 319–334, 2009.

[13] M. Pistore, P. Traverso, and P. Bertoli, "Automated composition of web services by planning in asynchronous domains," *Proc. ICAPS'05*, 2005.

[14] F. Lécué and A. Léger, "A formal model for semantic web service composition," *The Semantic Web-ISWC 2006*, pp. 385–398, 2006.

[15] F. Lécué, A. Delteil, A. Léger, and O. Boissier, "Web service composition as a composition of valid and robust semantic links," *International Journal of Cooperative Information Systems*, vol. 18, no. 1, pp. 1–62, 2009.

[16] J. Dong, Y. Sun, S. Yang, and K. Zhang, "Dynamic web service composition based on OWL-S," *Science in China Series F: Information Sciences*, vol. 49, no. 6, pp. 843–863, 2006.

[17] N. Lin, U. Kuter, and E. Sirin, "Web service composition with user preferences," *The Semantic Web: Research and Applications*, pp. 629–643, 2008.

[18] S. Sohrabi and S. McIlraith, "Preference-based Web service composition: A middle ground between execution and search," *The Semantic Web–ISWC 2010*, pp. 713–729, 2010.

[19] J. Lu, Y. Yu, and J. Mylopoulos, "A lightweight approach to semantic web service synthesis," in *WIRI*. IEEE Computer Society, 2005, pp. 240–247.

[20] J. L. A. Snehal Thakkar and C. A. Knoblock, "A data integration approach to automatically composing and optimizing web services," in *2004 ICAPS Workshop on Planning and Scheduling for Web and Grid Services*, June 2004.

[21] S. Bao, L. Zhang, C. Lin, and Y. Yu, "A Semantic Rewriting Approach to Automatic Information Providing Web Service Composition," *The Semantic Web–ASWC 2006*, pp. 488–500, 2006.

[22] M. Barhamgi, D. Benslimane, and B. Medjahed, "A query rewriting approach for web service composition," *IEEE T. Services Computing*, vol. 3, no. 3, pp. 206–222, 2010.

[23] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, and M. Zanker, "Configuration knowledge representations for semantic web applications," *AI EDAM*, vol. 17, no. 1, pp. 31–50, 2003.

[24] R. Klein, M. Buchheit, and W. Nutt, "Configuration as model construction: The constructive problem solving approach," in *Artificial Intelligence in Design*, vol. 94. Citeseer, 1994, pp. 201–218.

[25] P. Albert, L. Henocque, and M. Kleiner, "An end-to-end configuration-based framework for automatic sws composition," in *ICTAI (1)*. IEEE Computer Society, 2008, pp. 351–358.

[26] E. W. Dijkstra, "On the role of scientific thought," *Selected Writings on Computing: A Personal Perspective*, pp. 60–66, 1982.

[27] A. Y. Halevy, "Answering queries using views: A survey," *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.

[28] B. Srivastava and J. Koehler, "Web service composition-current solutions and open problems," in *ICAPS 2003 Workshop on Planning for Web Services*, vol. 35. Citeseer, 2003.

[29] S. Rudolph, "Exploring Relational Structures Via FLE F\! LE," *Conceptual Structures at Work*, pp. 233–233, 2004.

[30] I. Horrocks, U. Sattler, and S. Tobies, "Practical reasoning for very expressive description logics," *Logic Journal of IGPL*, vol. 8, no. 3, p. 239, 2000.

[31] E. Sirin and B. Parsia, "Pellet: An owl dl reasoner," in *2004 International Workshop on Description Logics*. Citeseer, 2004, p. 212.

[32] H. Toda, N. Yasuda, Y. Matsuura, and R. Kataoka, "Geographic information retrieval to suit immediate surroundings," in *GIS*, 2009, pp. 452–455.