Combining configuration and query rewriting for Web service composition [⋆]

Amin Mesmoudi, Michaël Mrissa, and Mohand-Saïd Hacid

Université de Lyon
Université Lyon 1
LIRIS CNRS UMR 5205
F-69622, France
`amin.mesmoudi@insa-lyon.fr`,
{`michael.mrissa,mohand-said.hacid`}`@liris.cnrs.fr`

Abstract. In this paper, we combine query rewriting and configuration to provide a new semantic-based approach to service composition, featuring a two-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. Given a user query and a set of service descriptions, query rewriting is used to decompose the query into sets of services that implement the required functionalities (discovery phase). At the orchestration phase, configuration is used to capture dependencies between services, and to generate a set of composite Web services ranked according to user preferences, while maintaining validity with respect to business rules organized into different levels (composition, service and user). We provide a formal framework and a complete implementation of the proposed approach, together with experiments by considering services from different domains.
Keywords:semantic Web services, composition, query rewriting, configuration

1 Introduction

Web services provide diverse functionalities that range from online payment to weather forecast, flight reservation, or simply data retrieval. Composition consists in combining several Web services into a composite one in order to provide the user with advanced, value-added functionalities (travel planning, online shopping, etc.). A composition involves several steps, which consist in: (1) decomposing a high-level user goal into sub-goals, (2) finding Web services that implement the functionalities of each sub-goal, and (3) orchestrating the interactions between composed Web services in order to achieve the high-level goal of the composition and to fulfill user's requirements.

[⋆] 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).

Several techniques exist to compose semantic Web services using AI planning, mainly based on planning as model checking, situation calculus and HTN planning (see, e.g., [6,19] for surveys). However, those techniques mainly focus on functional properties of services when building the execution plan, without investigating the complex dependencies that come into play between services, and do not offer a clear separation between discovery and orchestration tasks.

In our approach, we propose to achieve composition as a two-step task by resorting to discovery and orchestration. Discovery consists in finding individual Web services that implement functionalities required by sub-tasks extracted from the user's goal. Service discovery is mainly based on query rewriting. Orchestration consists in building an ordering for Web services invocation. This task falls into two steps: (1) retrieve the dependencies between Web services and (2) generate the execution plan. We perform the orchestration task by means of configuration techniques.

This paper is organized as follows: in Sect. 2, we provide some background knowledge on query rewriting and configuration. We also summarize the limitations of current approaches regarding Web service composition and we highlight the need for a combination of configuration and query rewriting. In Sect. 3 we develop the theoretical part of our proposal, and we detail our implementation in Sect. 4. Sect. 5 discusses the advantages and limitations of our approach, and Sect. 6 gives some insights on future work.

2 Background knowledge and Related Work

The approach described in this paper relies on a combination of *query rewriting* and *configuration* for service **discovery and composition**. In this section, we explore related works using these techniques for Web service composition. We also highlight the originality of the approach we propose with respect to the state of the art.

2.1 Semantic Web service composition

With the advent of the Semantic Web, Web services are annotated with semantic descriptions linked to ontologies, which makes their semantics explicit and machine-understandable and allows advanced reasoning about their capabilities, inputs/outputs, etc. The most known Web service ontologies are OWL-S [15] and WSMO [2], which both provide a general ontology for service description. Then, the Web service composition problem comes to the semantic level, which offers new opportunities for the automation of composition, using advanced techniques such as planning (see, e.g., [7,11,12,13,21,25]).

Some works such as [18] only deals with the discovery aspect of composition, without taking care about control and data dependencies between services. In our approach we consider these two steps of the composition.

In Lecué et al. [13], the authors propose an algorithm to compose semantic Web services with respect to functional constraints (represented as semantic links between services). In our work, in addition to functional constraints, we support other types of constraints (referred to as composition, service, and user constraints).

In the other works, the dependencies between services have to be part of the query and are not handled by the composition approach. For example, Sirin et al. [21] define the relations between services in the goal of the composition, and generate an order of execution of the atomic processes that constitute the new composite service at the end of the planning step. In our case, we propose algorithms to automatically detect (i.e. without user intervention) the dependencies between services by relying on both business rules and functional dependencies (see Sect. 3.4 for details).

Another limitation of the aforementioned works is the tight coupling with a particular description language (i.e., OWL-S or WSMO) to describe and reason about services. In our proposal, we separate the abstract and concrete levels of service description and we remain independent from the underlying semantic Web service (SWS) description formats. Instead, we describe service functionalities in a domain ontology. Many concrete services may implement the same abstract functionality, keeping the reasoning task at the abstract level. A similar work is proposed in Dong et al. [7] who extend the OWL-S profile with abstract descriptions of services, that are hierarchically organized. In the following, we review the main works related to the techniques we rely on (i.e. query rewriting and configuration) in order to discover and orchestrate Web services.

2.2 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 been widely explored in the database community. A survey of the main query rewriting algorithms is provided in [9].

With respect to the domain of Web service composition, query rewriting techniques have also been used in [14,22,4]. In such cases, Web services are accessed via Datalog queries. Lu et al. [14] provide a framework for answering queries with a conjunctive plan that includes inputs and outputs of participating Web services. In Thakkar et al. [22], 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. [4], 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 treat only functional constraints.

2.3 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., [5,8,10,16,17]). Configuration consists in finding sets of concrete objects that satisfy the properties of a given model.

With respect to Web service composition, several techniques based on configuration have been proposed (see, e.g., [1,20]). Sheng et al. [20] use configuration techniques to solve the composition problem. They rely on process templates and use ECA (Event Condition Action) rules together with tuple spaces in order to select services that match the process template. No template is required in our work, the process is dynamically generated from service dependencies.

In Albert et al. [1], the authors use configuration to calculate the composition goal, and from this goal they generate a valid workflow. So, configuration is used in the discovery task. On the contrary, we use configuration techniques to find relationships between services after the discovery task and we generate an execution plan with such relationships. The use of configuration for discovery is expensive in terms of complexity compared to query rewriting, as it requires reasoning with objects, whereas query rewriting simply consists in decomposing a query using propagation rules.

Additionally, in our case the size of the search space is bounded by the number of leaf concepts in the ontology. In the work of Albert et al. [1], the search space is the number of all valid services in the repository. Another limitation of this work comes from the integration of different kinds of constraints in the goal of composition i.e. there is no distinction between the different types of constraints. In our work, we classify constraints into different levels (classified by types of constraints). The user has not to specify such constraints in a query, they are used during the composition process. Doing this way, the user concentrates on the required functionalities rather than the way services should be combined. Hence, the query language displays the declarativity propery.

Combining query rewriting and configuration allows separating several concerns that come into play in the composition process. First, query rewriting allows identifying inputs, outputs and service functionalities required in the composition. Second, configuration enables the formalization of constraints at different levels (domain level, composition level and service level). The goal of our proposal is to improve the service discovery step with query rewriting and to handle service orchestration with the help of configuration constraints. In the following, we show the main advantages of our approach and illustrate it on a typical scenario.

3 Contribution

3.1 Case study

To illustrate our proposal, we use a case study that consists of an online travel reservation process. For instance, a user planning to travel to some country for a determined amount of time needs to book a flight, 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 part 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 functionalities that must be utilized to answer the query.

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 Sect. 3.5.

3.2 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. We will reason on the abstract descriptions of services, and do not handle the concrete part of services (binding).

Definition 1. *A* ***semantic Web services database*** $\mathcal{O}_{\mathcal{T}}$ *describes the structural part of services, i.e. the categories of services used in the database.*

Definition 2. *A* ***service*** S *is composed of a set of input parameters (I_S) and a set of output parameters (O_S).*

In our case study, we assume the existence of three categories of Web services in the application domain (e-tourism). Several instances of these Web service categories belong to the semantic Web service database and could implement these categories in different ways. For instance, hotel, flat, B&B and youth hostel reservation services are subcategories of the Accommodation category (see ontology at `http://liris.cnrs.fr/~soc/doku.php?id=transverse`).

In the following, we specify syntax and semantics of the language for describing the constrained vocabulary that will be used to specify $\mathcal{O}_{\mathcal{T}}$. Basically, axioms of the form $A \sqsubseteq D$ are contained in $\mathcal{O}_{\mathcal{T}}$. The elementary building blocks are primitive concepts (ranged over by the letter A) and primitive roles (ranged over by R). Intuitively, concepts describe sets of objects and thus correspond to unary predicates while attributes describe relations and thus correspond to binary predicates. Concepts (ranged over by D, E) are formed according to the following syntax rule:

$$\begin{array}{lrl} D, E \longrightarrow & A \mid & \text{primitive concept} \\ & D \sqcap E \mid & \text{conjunction} \\ & \forall R.D \mid & \text{universal quantification} \\ & \exists R.D \mid & \text{existential quantification} \\ & P(f_1, ..., f_n) \mid & \text{predicate restriction} \end{array}$$

Axioms come in the form $A \sqsubseteq D$. This axiom states that all instances of A are instances of D. An ontology part of services $\mathcal{O}_{\mathcal{T}}$ consists of a set of axioms.

Given a fixed interpretation, each formula denotes a binary or unary relation over the domain. Thus, we can immediately formulate the semantics of attributes and concepts in terms of relations and sets without the detour through predicate logic notation. An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a set $\Delta^{\mathcal{I}}$ (*the domain* of $\mathcal{I}$) and a function $\cdot^{\mathcal{I}}$ (the *extension function* of $\mathcal{I}$) that maps every concept to a subset of $\Delta^{\mathcal{I}}$, every constant to an element of $\Delta^{\mathcal{I}}$ and every attribute to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. Moreover, we assume that distinct constants have distinct images (Unique Name Assumption). The interpretation function can then be extended to arbitrary concepts as shown in Tab. 1.

Construct	Set Semantics
$(\forall R.D)^{\mathcal{I}}$	$\{d_1 \in \Delta^{\mathcal{I}} \mid \forall d_2.(d_1, d_2) \in R^{\mathcal{I}} \Rightarrow d_2 \in D^{\mathcal{I}}\}$
$(\exists R.D)^{\mathcal{I}}$	$\{d_1 \in \Delta^{\mathcal{I}} \mid \exists d_2.(d_1, d_2) \in R^{\mathcal{I}} \wedge d_2 \in D^{\mathcal{I}}\}$
$P(f_1, \ldots, f_n)^{\mathcal{I}}$	$\{d \in \Delta^{\mathcal{I}} \mid \exists d_1, \ldots, d_n \in \Delta^{\mathcal{I}} : f_1^{\mathcal{I}}(d) = d_1, \ldots, f_n^{\mathcal{I}}(d) = d_n$ and $(d_1, \ldots, d_n) \in P^{\mathcal{D}}\}$
$(D \sqcap E)^{\mathcal{I}}$	$D^{\mathcal{I}} \cap E^{\mathcal{I}}$

Table 1. Structural subsystem: semantics of the constructs

We say that two concepts C, D are equivalent iff $C^{\mathcal{I}} = D^{\mathcal{I}}$ for every interpretation $\mathcal{I}$, i.e., equivalent concepts always describe the same sets.
We say that an interpretation $\mathcal{I}$ *satisfies* the axiom $A \sqsubseteq D$ if $A^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. If $\mathcal{O}_{\mathcal{T}}$ is a set of axioms, an interpretation $\mathcal{I}$ that satisfies all axioms in $\mathcal{O}_{\mathcal{T}}$ is called a $\mathcal{O}_{\mathcal{T}}$-interpretation. A concept D is $\mathcal{O}_{\mathcal{T}}$*-satisfiable* if there is an $\mathcal{O}_{\mathcal{T}}$-interpretation $\mathcal{I}$ such that $D^{\mathcal{I}} \neq \emptyset$. We say that D is $\mathcal{O}_{\mathcal{T}}$-subsumed by E (written $D \sqsubseteq_{\mathcal{O}_{\mathcal{T}}} E$) if $D^{\mathcal{I}} \subseteq E^{\mathcal{I}}$ for every $\mathcal{O}_{\mathcal{T}}$-interpretation $\mathcal{I}$.

3.3 Query Rewriting

Each part (M and P) of the query Q is defined as a conjunction of terms. Each term is a concept expressed in the **query language** $\mathcal{L}$ 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 Sect. 3.2. In this section we focus on the Mandatory part of the query. The Preference part is considered in Sect. 3.5.

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. In this section we use Q to refer to the Mandatory part of 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 data, 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 expressing the needs for a travel:

$$Q = Transport \sqcap \exists HasInput.departurePlace \sqcap \exists HasInput.destinationPlace \sqcap \\ \exists HasInput.departureDate \sqcap \exists HasOutput.transportPrice \sqcap Accommodation \sqcap \\ \exists HasInput.checkoutDate \sqcap \exists HasOutput.accommodationPrice \sqcap \\ \exists HasOutput.accommodationDescription \sqcap CarRental \sqcap \exists HasOutput.rentalPrice \sqcap \\ \exists HasInput.retrievalDate \sqcap \exists HasInput.returnDate \sqcap \exists HasOutput.rentalDescription$$

The inputs specified in query Q are $departurePlace$, $destinationPlace$, $departureDate$, $checkoutDate$, $retrievalDate$ and $returnDate$. In our context, we are at design time, and thus we are looking for Web services that once composed will provide the required functionality. Hence, we do not specify the actual values to be sent to the resulting business process afterwards. According to the query Q, a query could have values such as ($Lyon, Paris$, 12/06/2010, 18/06/2010) as input data. Accordingly, the outputs expected as a result to the query are $transportPrice$, $accommodationPrice$, $accommodationDescription$, $rentalPrice$, and $rentalDescription$. As shown in Tab. reftab:tableofsubsumption, in our case study, we have the following information described in the ontology $\mathcal{O}_\mathcal{T}$ (available at `http://liris.cnrs.fr/~soc/doku.php?id=transverse`).

$Hotel \sqsubseteq Accommodation$	$Plane \sqsubseteq Transport$	$TouristCar \sqsubseteq CarRental$
$BedBreakfast \sqsubseteq Accommodation$	$Train \sqsubseteq Transport$	$BusinessCar \sqsubseteq CarRental$
$Flat \sqsubseteq Accommodation$	$Bus \sqsubseteq Transport$	

Table 2. Hierarchical relations between service categories of the domain

In order to rewrite the query we rely on a modified version of the bucket algorithm presented in [9]. The bucket algorithm allows to rewrite a user query according to existing views that relate to available data sources. *"Both the query and the sources are described by select-project-join queries that may include atoms of* ***predicates****... the main idea underlying the bucket algorithm is that* *the number of query rewritings that need to be considered can be drastically reduced if we first consider each* ***subgoal*** *in the query in isolation, and determine which* ***views*** *may be relevant to each subgoal"* [9].

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 [9] 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 *service categories*, *predicates* to *constraints* and *subgoals* to *concepts*. Views in the original bucket algorithm correspond to service categories in our context, and they are associated with constraints related to the service. The constraints can be expressed either directly in the query or taken from the ontology and added to the query. For example, when a user specifies an Accommodation request, the query rewriting consists in selecting the service categories subsumed by the concept *accommodation*, and identifies in the bucket those satisfying the constraints of the query.

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
```

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 (concrete services) that belong to the category C. For example, for the category *Transport*, $L_C = \{Plane, Train, Bus\}$. We suppose 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:

- Line 2: we initialize every L_C with one category element C.
- Lines 3-10: we try to fill all L_C's sets with concrete 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 $L = \bigcup_C L_C$, let BC be the set of all possible rewritings of Q_{cat}, then $BC = \prod_{l \in L} l$

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 of Tab. 3 is a possible rewriting of the query Q. Each cell of the first row denotes the service category mentioned in the query. Cells of the next rows describe concrete services (together with their inputs and outputs) that are subsumed by service categories of the query. Hence, each row of table 3 contains a combination of Web services that fulfill the Q_{cat} part of the query. In the sequel, we use Q_{cat}^c to denote the fact that the service category c is an element of Q_{cat}.

Transport	Accommodation	CarRental
Plane $\sqcap \exists HasInput.departurePlace$ $\sqcap \exists HasInput.destinationPlace$ ...	Hotel $\sqcap \exists HasInput.checkinDate$ $\sqcap \exists HasInput.checkoutDate$ ...	TouristCarRental $\sqcap \exists HasOutput.rentalDescription$ $\sqcap \exists HasOutput.rentalPrice$ ...
...	...	...

Table 3. Contents of the buckets

To each row of the table, we apply Algorithm 2. Its primary goal is to filter invalid combinations of services that do not provide all the required output parameters specified in Q. Its secondary goal is to identify inputs that services require and that are not provided in Q (MI). We denote by D the concrete services utilized to rewrite Q. For each service D, we define its inputs as D_{cons}^i and its outputs as D_{cons}^o. We assume that all the outputs in the request are missing and every time we find a service that provides an output which 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 delete 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.

3.4 Configuration

The configuration task consists in validating Web service composition with the 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. Configuration constraints may prove some rewriting to be inefficient for the needs of the composition, in such a case it is possible to select another set of services that could satisfy the composition. We distinguish between two types of constraints: composition-level and service-level constraints.

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
```

Composition level constraints are relevant to the application domain and involve several services. For example, "if both Flight and CarRental services are called in the composition, then CarRental can only be validated if the flight is successfully booked". Accordingly, service-level constraints are relevant to a specific service, for example "using the JapaneseHotel Web service implies using credit card payment".

Our use of configuration at a distinct stage from service discovery allows 1) decoupling business rules from generic facts in the domain knowledge representation, thus facilitating reuse of the domain ontology and its business exploitation in diverse ways, and 2) identifying constraints related to the composition and homogeneously incorporating these constraints into the composition in order to detect any inconsistencies.

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 s1 must precede s2 in the execution order.
- 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 service s1 in a composition involving s2.

A calculus for configuration

The configuration task is defined as a deductive reasoning task using business rules represented as constraints between services to provide a dependency graph G that represents the dependencies between services that form the execution plan of the composition. 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 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. The following example represents the configuration goal specification corresponding to the first row of our table of buckets shown in table 3:

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

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

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

Here, we have three business rules for the e-tourism domain. The first one expresses that if *Transport* and *CarRental* services are involved in the same composition thus *CarRental* depends on *Transport* (i.e. *Transport* precedes *CarRental*). The second, if *Transport* and *Accommodation* services are involved in the same composition thus *Accommodation* depends on *Transport* (i.e. *Transport* precedes *Accommodation*), and finally, if *Accommodation* and *CarRental* services are involved in the same composition thus *CarRental* depends on *Accommodation* (i.e. *Accommodation* precedes *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 can be applied. In order to simplify the definition of rules we use the following notations (with $(s, d) \in S \times S$):

- A **simple 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 order to solve our problem, in the following $\acute{R}$ denotes the changed/new set of dependencies:

R1 $(S \cup \{s\}, MI).(V, R) \to (S, MI).(V \cup \{s\}, \acute{R})$ with $\acute{R} = R \cup D^D_{S \cup V}(s)$ and $D^D_{S \cup V}(s)$ is the set of induced dependencies related to the service s.
R2 $(S, MI \cup \{i\}).(V.R) \to (S, MI).(V, \acute{R})$ 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 $\acute{R} = R \cup (s_1, s_2)$
R3 $(S, MI \cup \{i\}).(V, R) \to (S, MI).(V, R)$ if $\exists \acute{i} \in Q^i_{cons}$ with $\acute{i} \sqsubseteq i$

R4 $(\emptyset, MI).(V \cup \{x, y\}, R) \rightarrow (\emptyset, MI).(\acute{V}, R)$
$\acute{V} = V \cup \{x, y\}$ 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
R5 $(\emptyset, MI).(V \cup \{x, y\}, R) \rightarrow (\emptyset, MI).(\acute{V}, R)$
$\acute{V} = V \cup \{x\}$ 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
R6 $(\emptyset, MI).(V, R \cup Path(x, y) \cup (x, y)) \rightarrow (\emptyset, MI).(V, R \cup Path(x, y))$
R7 $(\emptyset, MI).(V, R \cup Path(x, x)) \rightarrow (\emptyset, MI).(V \cup \{\perp\}, R)$

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

- R1 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.
- R2 reflects the creation of dependency relationships between services.
- R3 removes missing inputs provided from another part of the query.
- R4 detects incompatibilities between services.
- R5 controls the requirements of services.
- R6 detects deadlocks between services (cycles in the solution graph).
- R7 optimizes dependencies between services (removes useless dependencies).

The configuration task starts with $V = \emptyset$, $R = \emptyset$ and $S = L_s$, where L_s is a set of services obtained from 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 the example of the first row of Tab. 3, 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 rule R1 to Plane, Hotel and TouristCarRental results in the creation of new dependencies
- Application of R2 to chekinDate means that this missing input can be provided from another service of the same composition
- Application of R3 to location means that this missing input can be provided from user query
- Application of R4 remove useless dependencies
- After firing these rules we obtain the following solution:
  $(S, MI) = (\emptyset, \emptyset)$
  $(V, R) = (\{$ *Plane*, *Hotel*, *TouristCarRental*$\}$,$\{$ (*Plane*, *Hotel*), (*Hotel*, *TouristCarRental*) $\}$)

The inference engine performs the selection and the application of rules while keeping satisfied a complete priority order between rules using the transitive function ($\preccurlyeq$) which means that the right side has a higher priority than the left side, as follows : $R7 \preccurlyeq R6 \preccurlyeq R5 \preccurlyeq R4 \preccurlyeq R2 \preccurlyeq R3 \preccurlyeq R1$

Execution Plan We can now use the result of the configuration to build the execution plan P, where P is a triple $\langle S_0, O, S_G \rangle$, the initial state S_0 represents a state where no service is invoked yet, the S_G is the goal state where all services have been invoked, O is the order of service execution, described as follows: first, we invoke services that have no dependency. We can identify these services with $X \in V$ such that $\exists Y \in V$ and $\nexists (Y, X) \in R$ with (V, R) the configuration result. Then, we invoke each service that has its precedence constraints fulfilled (invocation of precedent services is over). The execution is completed when there are no services to execute, in this case we are in state S_G (goal).

3.5 User preferences and Ranking

Integrating user preferences in composition allows us to obtain ranked results. Few works investigated the integration of user preferences in the composition task [3,23]. In [3], the authors proposed the integration of non-functional preferences (QoS) in the service selection task. They resort to Sohrabi et al. [23] proposal to integrate user preferences in the composition of services with Golog at the instance level.

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 part M 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" which 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 [24]. 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 is 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.

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

Definition 4. *(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 set of inputs and outputs respectively related to the services in V.

$$Closeness(c, R) = \begin{cases} 1 & \iff c \in V \cup V^i_{cons} \cup V^o_{cons} \ with \ R = (V, \acute{R}) \\ 0.8 & \iff \exists y, \ y \sqsubset c \ such \ that \ y \in V \cup V^i_{cons} \cup V^o_{cons} \ with \ R = (V, \acute{R}) \\ 0.2 & \iff \exists y, \ c \sqsubset y \ such \ that \ y \in V \cup V^i_{cons} \cup V^o_{cons} \ with \ R = (V, \acute{R}) \\ 0 & \quad otherwise \end{cases} \tag{2}$$

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

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.

Definition 5. *Composition score*
We define the composition score $S(R, P)$ for a given composition result R using P as follows:

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

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

3.6 Selecting instances of Web services

Once the services have been configured, there is a need to select concrete instances of Web services, which are identified with their description files in the Web service repository. These description files are written in some SWS description language like OWL-S or WSMO, but more importantly they refer to terms of $\mathcal{O}_\mathcal{T}$ in order to explicitly describe in a machine-interpretable way the functionality the corresponding Web services provide. Since several compositions could be proposed, we select one of them, we look into the repository for Web services that match the functionalities, and in case no services can be found, we select another composition until a valid combination is found or until there are no more available combinations, in which case no solution can be found.

4 Implementation and experiments

In this section we describe the implementation of a prototype that automates the composition of semantic Web services according to the design process described in the paper. We also provide experiments.

In this section, we show the underlying architecture and how we implemented it. We discuss the tools used and some scenarios on the use of our framework. As shown in Fig. 1, our framework is divided into two major components (A) and (B). (A) represents the repository indexing tools and (B) represents the composition tools.

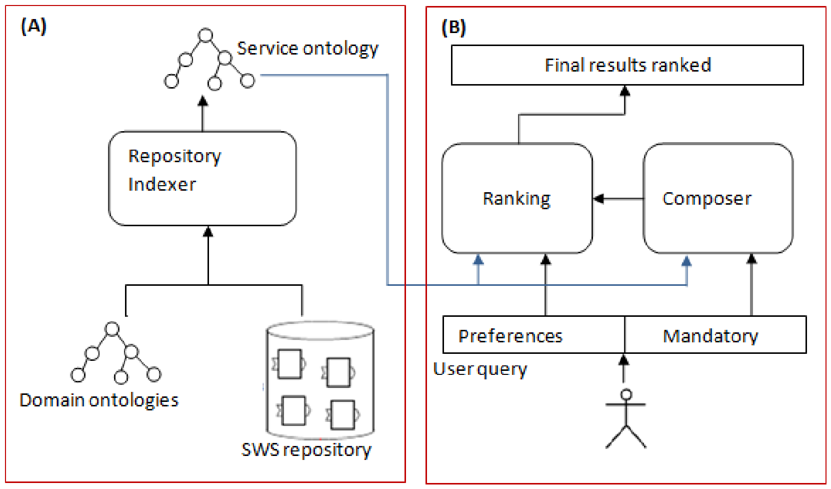

Fig. 1. General architecture of our framework

(A) Repository indexing tool : This component indexes and categorizes a set of SWS from a given application domain into a single service ontology that describes the functionalities the services offer. It takes as input a set of SWS descriptions and the corresponding domain ontology and generates an abstract representation of their functionalities organized in a hierarchical structure that respects the relations between service functionalities according to the domain ontology. Our service ontology is defined with the language presented in Sect. 3.2. As shown in Fig. 2 this component is split into two parts: a **repository parser** that takes as input a set of services of a given domain and stores services in a single data structure. In our prototype, the services are defined with OWL-S, and their descriptions are parsed with the OWL-S API [1]. An **ontology builder** that reads the generated data structure and the domain ontology to generate

[1] `http://on.cs.unibas.ch/owls-api/`

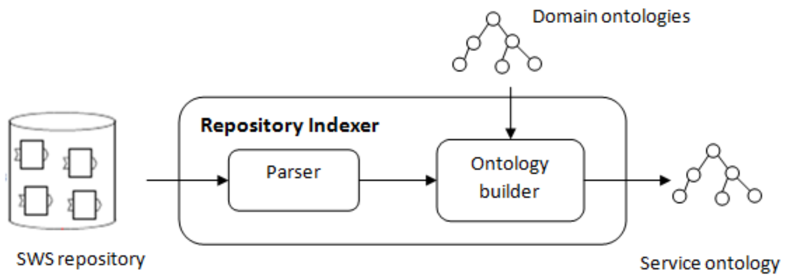

Fig. 2. Detailed overview of the indexer tools

a service ontology. The domain ontology is used to generate the relationships between the concepts of the new service ontology. In the implementation of this component we relied on the Jena library[2] to access to domain ontology and create the new service ontology.

(B) Composition tool : This component generates the execution plan from services involved in the composition and sorts execution plans according to user preferences. It takes as input the user query and the service ontology. Fig. 3 shows details of the two parts of this component (composer and ranking system).

The **Composer** part takes the mandatory part of the user query and a service ontology and generates a set of dependency graphs. We can distinguish the following parts in the composer:

1. **Query parser**: it checks the syntactic conformity of the user query and generates from this query a set of objects that represent inputs, outputs and service categories. Its implementation relies on ANTLR[3]. We have written a grammar by means of ANTLRWorks[4] and generated the lexer and parser code that were integrated into an Eclipse project. Antlr uses the LL analyzer.
2. **Correctness**: it checks the logical form of the user query. For example, it eliminates duplicate objects.
3. **Service Extractor**: it extracts services, which correspond to views mentioned in user query. In the implementation, we use Jena to access to service ontology and extract the final subclasses of categories mentioned in the mandatory part of the query.
4. **Rewritings generator**: it implements the bucket algorithm, it generates all the combinations of services.
5. **Rewritings checker**: it implements algorithm 2 that eliminates rewritings that cannot be potential solutions.

[2] `http://jena.sourceforge.net`
[3] `http://www.antlr.org/`
[4] `http://www.antlr.org/works`

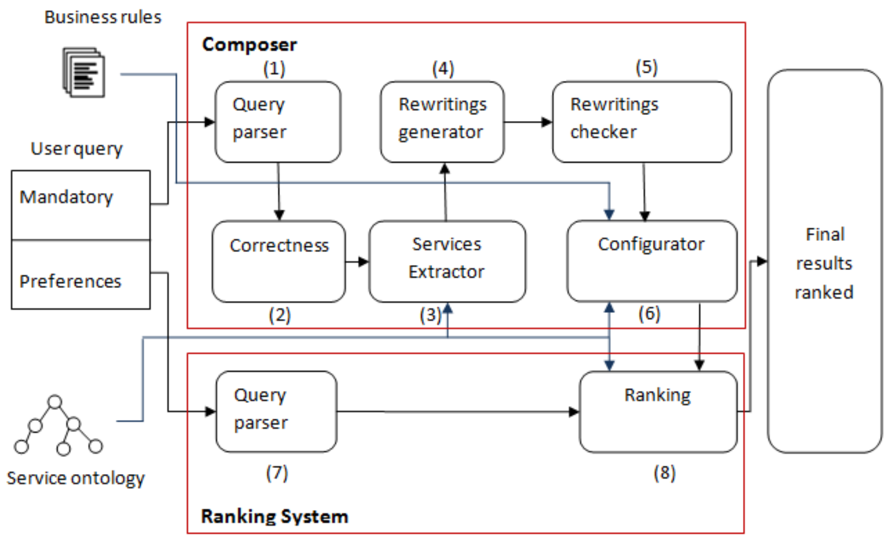

Fig. 3. Detailed overview of the composition tools

6. **Configurator**: it implements the calculus defined in Sect. 3.4, the goal is to generate a graph of dependencies between services, starting from a set of rewritings, the service ontology and a set of the business rules. We use three binary predicates namely `incompatible`, `depends` and `required`, to represent the business rules of a domain (described in Sect. 3.4). In the implementation of this component we used Drools[5] which is a forward chaining inference-based rules engine, together with the Jena 2 library for ontology access and a simple predicate parser to access business rules. Our rules are defined in Sect. 3.4, and described in a separate file from the rules engine implementation.

The **Ranking System** ranks the results of the configuration according to the user preferences, it contains a parser and ranking component. In the ranking component we implemented our model of Ranking defined in Sect 3.5. We used the Jena library to access the service ontology.

Our framework is open source and available for download on the project website[6] 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)[7] framework to display graphs.

[5] `http://www.jboss.org/drools`
[6] `http://liris.cnrs.fr/~soc/doku.php?id=transverse`
[7] `http://jung.sourceforge.net`

4.1 Results and preliminary evaluation

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

Experimentation setup : we conducted experiments using the prototype system implemented to evaluate the approach. In the experiments, the computer used to run the prototype system has the following features: Intel (R) Core (TM) 2 CPU, 2.1 GHz with 3GB RAM. The PC runs Windows Seven and Java SE v 1.6.017. In our testing phase, we used OWL-S TC [8], which is a collection of services for semantic Web services. This collection contains over **1,000 concrete services**, organized into **7 domains** as shown in Tab. 4

Domain	education	medical care	food	travel	communication	economy	weapon
Services	286	286	34	195	59	395	40

Table 4. overview of OWL-S TC services

OWL-S TC uses 23 domain ontology to describe the concepts used in the definitions of Web services in OWL-S. We used domain ontology to index the SWS repository i.e. create abstract definitions of Web services. More details can be found in the documentation of OWL-S TC.

Results The objective is to investigate the influence of some parameters on the execution time of the processes (composition, discovery and orchestration). We generated for each domain a query, to properly measure the variation of execution time of each phase. Tab. 5 shows the number of services and the size of the query for each service ontology in the 7 domains. These 7 ontologies have been extracted from a service descriptions in OWL-S TC using our repository indexer (see Sect. 4).

Domain	education	medical care	food	travel	communication	economy	weapon
Services	658	138	68	348	113	868	102
Serv. in Q	16	22	8	12	4	9	3

Table 5. Overview of service ontologies

Fig. 4 shows the variation of execution time for each phase compared to the number of services in the query.

[8] `http://projects.semwebcentral.org/projects/owls-tc/`

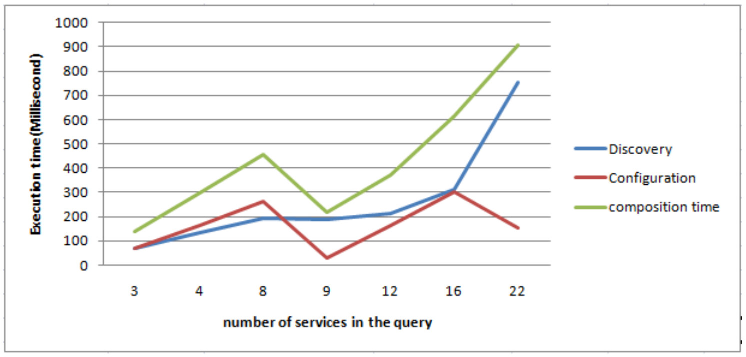

Fig. 4. Evolution of the execution time

One can see that the execution time for the discovery phase (blue curve) is increasing i.e. when the number of services in the query increases the execution time increases. There is also another parameter that impacts the variation of execution time: the number of leaves (in the ontology) for the categories of services. So every time this number increases the number of combinations increases and consequently the execution time increases.

For the second curve (in red) that represents the change in execution time compared to the number of services used in the query for the configuration phase, one can see that the execution time is not increasing because this phase takes as input the buckets table, so if the size of this table is small, the time will decrease, as the configuration of the two queries (services =9 and services = 22) which was the size of buckets table after rewriting, 4 and 18 respectively. Other parameters impact on the execution time, for example the number of missing input for a rewrite, size of business rules associated with a domain, etc. The queries used for testing and service ontologies are available for download on the project Web site.

5 Discussion

Our approach is characterized by a clear separation between query rewriting (for discovery) and configuration (for orchestration). First, the use of query rewriting reduces the complexity of service discovery. Indeed, this technique requires a very simple formalism to hierarchically organize the functionalities of a domain, as opposed to OWL-S or similar languages that require complex reasoning for discovery, due to their strong expressivity. We facilitate reasoning at this stage by relying exclusively on subsumption relations between concepts. Hence, the discovery step provides sets of services that are potential solutions to a query. Orchestration issues are left to the configuration part because of their complexity.

Second, configuration allows integrating business rules as constraints related to the domain of composition and to services of this domain without mixing with the domain ontology, keeping a clear separation between fast-changing business rules and general, more static, domain knowledge. The use of Event-Condition-Action (ECA) rules is more straightforward (and probably less time-consuming) than advanced reasoning on constraints expressed as predicates. Then, it is easier to automate the finding of dependency relations between composed services while satisfying business rules. Another advantage is that the dependencies between composed services do not have to be explicitly part of the query (as it is the case with most existing works like SHOP2-based works). In our approach, we handle them. Finally, our simple, two-step approach to composition allows to support user preferences in the composition process. We proposed a ranking algorithm that classifies valid solutions, if any, depending on a set of user preferences.

6 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, in order to facilitate the composition of semantic Web services.

The main feature of the proposed approach is its construction as a two-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 accommodate user preferences as part of the composition process.

There are many research directions to be pursued. First, we plan to investigate how to accommodate arbitrary business rules during the configuration steps. Other interesting issues are related to the support of (i.e. optimization w.r.t.) QoS aspects of the composition such as cost, response time or reliability, or approximate queries. That is, when an exact composition is not feasible, we should look for alternative solutions that better reflect user preferences. Third, the scalability problem should be tackled on real world services. Another issue is the optimization of the rewriting algorithm.

These issues are currently being investigated.

References

1. P. Albert, L. Henocque, and M. Kleiner. An end-to-end configuration-based framework for automatic sws composition. In *ICTAI (1)*, pages 351–358. IEEE Computer Society, 2008.
2. S. Arroyo and M. Stollberg. WSMO Primer. WSMO Deliverable D3.1, DERI Working Draft. Technical report, WSMO, 2004. http://www.wsmo.org/2004/d3/d3.1/.
3. Y. Badr, A. Abraham, F. Biennier, and C. Grosan. Enhancing Web Service Selection by User Preferences of Non-functional Features. In *Proceedings of the 2008 4th International Conference on Next Generation Web Services Practices*, pages 60–65. IEEE Computer Society, 2008.
4. 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*, pages 488–500.

5. H. Burckert, W. Nutt, and C. Seel. The Role of Formal Knowledge Representation in Configuration. *WRKP'96: Knowledge Representation and Configuration Problems*, 1996.
6. K. Chan, J. Bishop, and L. Baresi. Survey and comparison of planning techniques for web services composition. *University of Pretoria2007.© ISMED*, 209, 2007.
7. J. Dong, Y. Sun, S. Yang, and K. Zhang. Dynamic web service composition based on OWL-S. *Science in China Series F: Information Sciences*, 49(6):843–863, 2006.
8. A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, and M. Zanker. Configuration knowledge representations for semantic web applications. *AI EDAM*, 17(1):31–50, 2003.
9. A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.
10. R. Klein, M. Buchheit, and W. Nutt. Configuration as model construction: The constructive problem solving approach. In *Artificial Intelligence in Design*, volume 94, pages 201–218. Citeseer, 1994.
11. M. Klusch, A. Gerber, and M. Schmidt. Semantic web service composition planning with owls-xplan. In *Proceedings of the AAAI Fall Symposium on Semantic Web and Agents, Arlington VA, USA, AAAI Press*, 2005.
12. A. Kumar, B. Srivastava, and S. Mittal. Information modeling for end to end composition of semantic web services. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 476–490. Springer, 2005.
13. F. Lecue, A. Delteil, A. Leger, and O. Boissier. Web service composition as a composition of valid and robust semantic links. *International Journal of Cooperative Information Systems*, 18(1):1–62, 2009.
14. J. Lu, Y. Yu, and J. Mylopoulos. A lightweight approach to semantic web service synthesis. In *WIRI*, pages 240–247. IEEE Computer Society, 2005.
15. 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 J. Cardoso and A. P. Sheth, editors, *SWSWPC*, volume 3387 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2004.
16. D. L. McGuinness and J. R. Wright. Conceptual modelling for configuration: A description logic-based approach. *AI EDAM*, 12(4):333–344, 1998.
17. O. Najmann and B. Stein. A theoretical framework for configuration. In F. Belli and F. J. Radermacher, editors, *IEA/AIE*, volume 604 of *Lecture Notes in Computer Science*, pages 441–450. Springer, 1992.
18. M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In I. Horrocks and J. A. Hendler, editors, *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2002.
19. J. Rao and X. Su. A survey of automated web service composition methods. *Semantic Web Services and Web Process Composition*, pages 43–54, 2005.
20. Q. Sheng, B. Benatallah, Z. Maamar, and A. Ngu. Configurable Composition and Adaptive Provisioning of Web Services. *IEEE Transactions on Services Computing*, 2(1):34–49, 2009.
21. E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for web service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, 2004.

22. 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.
23. S. Sohrabi, N. Prokoshyna, and S. McIlraith. Web service composition via the customization of Golog programs with user preferences. *Conceptual Modeling: Foundations and Applications*, pages 319–334, 2009.
24. H. Toda, N. Yasuda, Y. Matsuura, and R. Kataoka. Geographic information retrieval to suit immediate surroundings. In D. Agrawal, W. G. Aref, C.-T. Lu, M. F. Mokbel, P. Scheuermann, C. Shahabi, and O. Wolfson, editors, *GIS*, pages 452–455. ACM, 2009.
25. M. Vukovic and P. Robinson. SHOP2 and TLPlan for proactive service composition. In *UK-Russia Workshop on Proactive Computing*. Citeseer, 2005.