Combining configuration and query rewriting for Web service composition Preliminary report

Michaël Mrissa, Mohand-Saïd Hacid

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 new one in order to provide the user with advanced, value-added functionalities (travel planning, online shopping, etc.). A composition involves several steps, which consists in: (1) decomposing the high-level user goal into subtasks, (2) finding Web services that implement the functionalities of each subtask, 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.

Several techniques exist to compose Web services, mainly variants of planning such as model checking or situation calculus [12]. However, these techniques mainly focus on finding a composition without building the execution plan. In our approach, we propose to achieve the composition by resorting to discovery and orchestration. Discovery consists in finding individual web services that implement functionalities required by tabtasks extracted from user's goal. This discovery is mainly based on query rewriting, and (2) orchestration which consists in building an ordering for web services invocation. Orchestration is based on configuration techniques. Thus, in this paper we investigate the combination of configuration and query rewriting techniques in order to facilitate Web service discovery and orchestration.

In Section 2, we provide some background knowledge on query rewriting and configuration. We also summarize the limitations of current works and highlight the need for a combination of configuration and query rewriting. In Section 3, we develop our proposal and show how it facilitates the composition task. Finally, we discuss the approach and give some insights on future works in Section 4

2 Background knowledge and Related Work

The proposal developed in this paper relies on a combination of query rewriting and configuration techniques. In this section we introduce these approaches in order to provide the reader with some background knowledge for a good understanding of this paper. At the same time we highlight the originality of the approach we propose with respect to existing works.

2.1 Query Rewriting

Query rewriting (using views) consists in reformulating a query according to views that are already available from the database, in order to optimize the execution plan of the query. Query rewriting techniques have been widely explored in the database field. A good survey of the main query rewriting algorithms is presented in [5].

With respect to the domain of Web service composition, query rewriting techniques have also been utilized in [8, 13]. In both works, Web services are accessed via datalog queries. Lu et al. [8] provide a framework for answering queries with a conjunctive plan that includes inputs and outputs of participating Web services. In Thakkar et al. [13], 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.

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 [9] or WSMO [2], which both provide a general ontology for service description that support XML syntax. 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 [7, 11, 12].

2.2 Configuration

Configuration has been part of the Artificial Intelligence (AI) field for a long time. Some attempts to formalize configuration have been proposed in [3, 6, 10]. 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]).

In [1], the authors decompose the composition task into two main stages. First, composition is performed at the abstract level, which consists in identifying which sets of Web services can satisfy the composition at the func-

tional level. Second, the possible sets of Web services are processed and a valid workflow is generated. Configuration is utilized in the second step

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 interest of our proposal is to improve the service selection step, and in a second time 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 Running Example

To illustrate the idea, we use a running example that consists of an online travel reservation process. For instance, a user planning to travel to some country for a certain period needs to book a flight, to find an accommodation, and to rent a car in order to visit some interesting sites around. The domain ontology this example relies on is presented in Fig. 1.

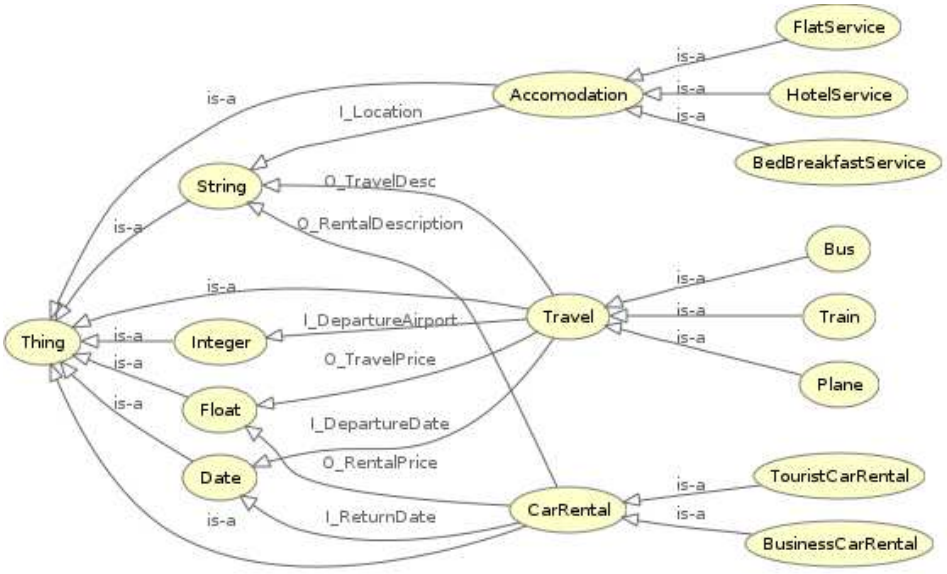

Figure 1: Overview of the travel ontology

We model user's requirements for a composition with a query Q specified as a triple $< I, O, C >$ where I (for input) denotes the input data the user provides, which are treated as constraints that reflect the requirements

of the user in the query, and O (for output) denotes the unknown part of the request, i.e. the information the user looks for, and C denotes service categories that must be utilized to answer the query. In our example, I includes **departure and return dates and locations**, and O includes **required information** provided by three categories of Web services C (in bold), i.e. **transport** (flight, train or bus ticket number and details), **accomodation** (hotel, flat or b&b information) and **vehicle rental** (type of vehicle and price).

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.

3.2 Defining a WS description language

3.2.1 Context

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. We do not handle the concrete part of services.

Definition 1 *A **semantic Web services database** $\mathcal{O}_\mathcal{T}$ describes the structural part of services. That is, 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), constrained by some contents in $\mathcal{O}_\mathcal{T}$ and identified with the prefix $I\_$ denoting an input parameter and $O\_$ denoting an output parameter.*

In our running example, we assume three categories of Web services in the application domain (e-tourism). These categories are shown in Figure 2.

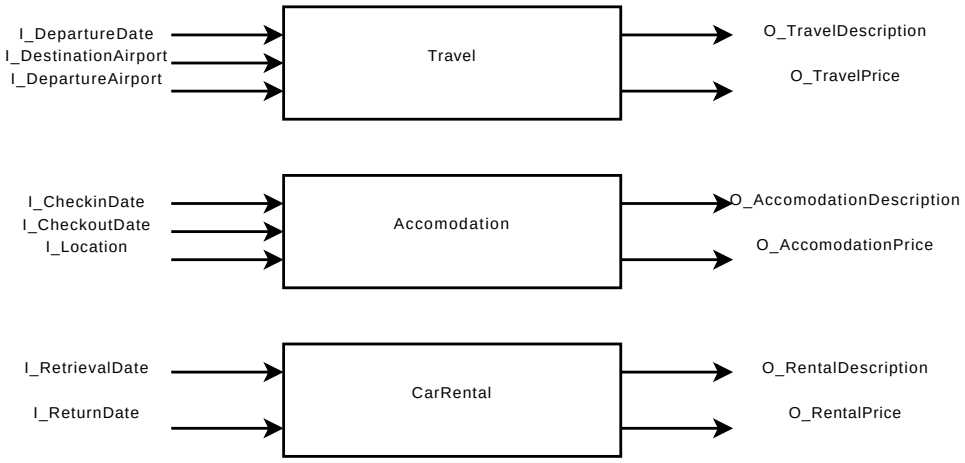

Figure 2: Categories of e-tourism Web services

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 in Figure 1.

3.2.2 The ontology part

Here we specify syntax and semantics of the language for describing the constrained vocabulary that will be used to specify $\mathcal{O}_\mathcal{T}$.

Basically, the atom $A \sqsubseteq D$ is used in the descriptions 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 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:

$D, E \longrightarrow$	$A \mid$	primitive concept
	$D \sqcap E \mid$	conjunction
	$\forall R.D \mid$	universal quantification
	$\exists R.D \mid$	existential quantification
	$P(f_1, ..., f_n) \mid$	predicate restriction

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 figure 3 ($\#S$ denotes the cardinality of the set S).

We say that two concepts C, D are equivalent if $C^\mathcal{I} = D^\mathcal{I}$ 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

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 $(d_1, \ldots, d_n) \in P^{\mathcal{D}}\}$
$(D \sqcap E)^{\mathcal{I}}$	$D^{\mathcal{I}} \cap E^{\mathcal{I}}$

Figure 3: Structural subsystem: semantics of the constructs

$\mathcal{O}_{\mathcal{T}}$-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

A query Q is defined as a conjunction of terms. Each term can be 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 Section 3.2.2.

We identify three types of concepts in a query: **inputs**, **outputs** and **service categories**. Inputs have their values provided by the user on query submission. Outputs must be provided as an answer to the query execution, and service categories represent the categories of services (in terms of functionality) that can be utilized in order to answer the 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 first in finding Web services belonging to the relevant categories (i.e. resolve the Q_{cat} part of the query), and second 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:

$$Q = Travel \sqcap \exists I\_departurePlace \sqcap \exists I\_destinationPlace \sqcap \exists I\_departureDate \sqcap \exists O\_TravelPrice \sqcap Accom$$

The inputs specified in query Q are

$$I\_departurePlace, I\_destinationPlace, I\_departureDate, I\_retrievalDate, \text{ and } I\_returnDate.$$

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

$$O\_TravelPrice,\ O\_AccommodationPrice\ ,O\_AccommodationDescription,\ \text{and}\ O\_CarRentalPrice$$

In our running example, we have the following information described in the ontology $\mathcal{O}_{\mathcal{T}}$ (see figure 1):

- $Hotel \sqsubseteq Accomodation$ and $BedBreakfast \sqsubseteq Accomodation$ and $Flat \sqsubseteq Accomodation$ for the Accomodation class,
- $TravelbyPlane \sqsubseteq Travel$ and $TravelbyTrain \sqsubseteq Travel$ for the Travel class,
- $TourismCar \sqsubseteq CarRental$ and $BusinessCar \sqsubseteq CarRental$ for the CarRental class.

In order to rewrite our query we rely on a modified version of the bucket algorithm presented in [5]. 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* [5].

In order to rewrite a query Q, the bucket algorithm starts by creating a bucket for each subgoal containing the views that are relevant to the response. Then it 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 [5] 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 request or taken from the ontology and appended to the query.

When a user specifies an Accommodation request for example, the query rewriting consists in selecting the service categories subsumed by the accommodation class, and identifies in the bucket those that satisfy the constraints of the query.

We recursively apply the following propagation rule, where C and D are concepts in the ontology such that $D \sqsubseteq C$ is an element of the ontology:

Algorithm 1 Propagation rule

```
for all C in Q do
  if D ⊑ C
  and D is not in Q then
    Q → Q ∪ {D} \ {C}
  end if
end for
```

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.

The first step of our algorithm consists in creating a bucket for each service category in the query, as shown in Table 1. Each cell of the first row denotes the service category mentioned in 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}. 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 1 contains a combination of Web services that fulfills the Q_{cat} part of the query.

Travel	Accommodation	CarRental
TravelbyPlane $\sqcap\exists I\_departureAirportCode \sqcap$ $\exists I\_destinationAirportCode \sqcap$ $\exists I\_departureDate \sqcap$ $\exists O\_TravelPrice \sqcap$ $\exists O\_TravelDescription$	Hotel $\sqcap\exists O\_Accom$	
TravelbyTrain $\sqcap\exists I\_departureTrainStation\sqcap$ $\exists I\_destinationTrainStation\sqcap$ $\exists I\_departureDate \sqcap$ $\exists O\_TravelPrice \sqcap$ $\exists O\_TravelDescription$	Flat $\sqcap\exists O\_AccommodationPrice \sqcap$ $\exists O\_AccommodationDescription \sqcap$ $\exists O\_Grade$	TouristCarRental $\sqcap\exists O\_CarRentalPrice \sqcap$ $\exists O\_CarRentalDescription \sqcap$ $\exists O\_HasGPS \sqcap$ $\exists I\_returnDate$
TravelbyTrain $\sqcap\exists I\_departureTrainStation\sqcap$ $\exists I\_destinationTrainStation\sqcap$ $\exists I\_departureDate \sqcap$ $\exists O\_TravelPrice \sqcap$ $\exists O\_TravelDescription$	Flat $\sqcap\exists O\_AccommodationPrice \sqcap$ $\exists O\_AccommodationDescription \sqcap$ $\exists O\_Grade$	TouristCarRental $\sqcap\exists O\_CarRentalPrice \sqcap$ $\exists O\_CarRentalDescription \sqcap$ $\exists I\_returnDate$

Table 1: 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.

In this algorithm, MO represents the missing outputs and must be empty at the end of the computation, for the combination to be valid. MI represents missing inputs, and a non-empty value indicates that at least one service in the line requires input data that are not provided in Q. Such information is useful, as missing inputs could be provided by other services involved in the composition. We will get back to this issue later on.

We denote as D the concrete services utilized to rewrite Q. For each service D, we define its inputs as D^i_{cons} and its outputs as D^o_{cons}.

Algorithm 2 I/O algorithm

for all row L in BC **do**
  $MI = \emptyset$, $MO = \emptyset$
  for all service D **do**
    $MI = MI \cup \{D^i_{cons}\} \backslash Q^i_{cons}$
    $MO = MO \cup Q^o_{cons} \backslash \{D^o_{cons}\}$
  end for
  if $MO \neq \emptyset$ **then**
    some output is missing: invalid combination
    remove the line from the table
  else
    record MI and keep the set of D as a possible solution
  end if
end for

We add two columns in the BC table, in order to represent MI and MO.

3.4 Selecting instances of Web services

Once the query has been rewritten, 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 OWL-S and they refer to terms of $\mathcal{O}_\mathcal{T}$ in order to explicitly describe in a machine-interpretable way the functionality the corresponding Web service provides.

Since several rewritings could be proposed, it is possible to select one of the rewritings, to look into the repository for Web services that correspond to the functionality, and in case none can be found, select another rewriting until a valid combination is found.

This step ends up with several sets of Web services that, together, fulfill the user's requirements in terms of functionality and input/output data. The next step, called configuration, builds on these sets of Web services, and verifies the validity of these sets with respect to business rules of the

domain, constraints of the services, and user constraints.

3.5 Configuration

The configuration task consists in validating Web service composition with the business constraints that need to be applied in each application domain. Constraints include causality relationships between Web service invocations, 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 case it is possible to return to the instance selection step and to select another set of service instances that satisfies the composition.

We distinguish between two types of constraints: composition level "business" constraints, and service level constraints.

Composition level constraints are generic and relevant to the application domain, 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".

Configuration 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 Web services and homogeneously incorporating these constraints into the composition in order to detect any inconstancies.

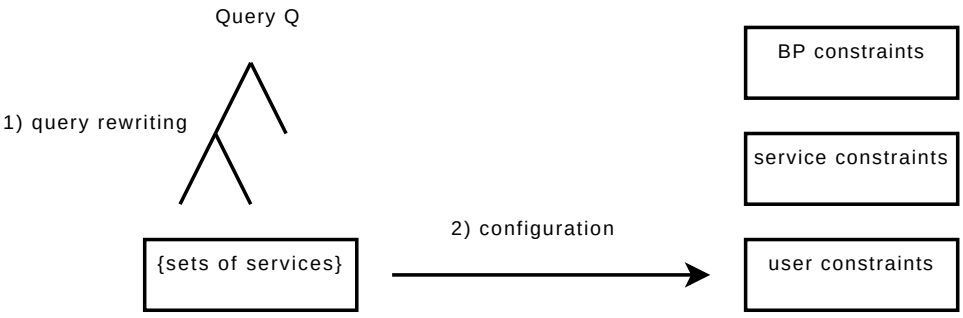

Figure 4: Overview of the composition method

4 Conclusion

This preliminary research report shows how to adapt the bucket rewriting algorithm to the requirements of Web service composition with the help of additional algorithms. After providing some background on related works, we define an ontology language for describing domain knowledge with respect to Web services and we provide algorithms in order to facilitate service selection.

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] H.-J. Bürckert, W. Nutt, and C. Seel. The role of formal knowledge representation in configuration.

[4] J. Cardoso and A. P. Sheth, editors. *Semantic Web Services and Web Process Composition, First International Workshop, SWSWPC 2004, San Diego, CA, USA, July 6, 2004, Revised Selected Papers*, volume 3387 of *Lecture Notes in Computer Science*. Springer, 2004.

[5] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[6] R. Klein, M. Buchheit, and W. Nutt. Configuration as model construction: The constructive problem solving approach. *Scientific Commons*, 1994.

[7] 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.

[8] J. Lu, Y. Yu, and J. Mylopoulos. A lightweight approach to semantic web service synthesis. In *WIRI*, pages 240–247. IEEE Computer Society, 2005.

[9] 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 Cardoso and Sheth [4], pages 26–42.

[10] 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.

[11] 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.

[12] J. Rao and X. Su. A survey of automated web service composition methods. In Cardoso and Sheth [4], pages 43–54.

[13] 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.