Semantic Web Service Offer Discovery with Lightweight Semantic Descriptions[⋆]

Jacek Kopecký and Elena Simperl

Semantic Technology Institute (STI Innsbruck)
Innsbruck, Austria
⟨firstname.lastname⟩@sti2.at

Abstract. Semantic Web Services (SWS) are a research effort aimed at automation of the usage of Web services, a necessary component for the Semantic Web. Offer discovery is an important part of the general discovery process of finding the most suitable services for a user's goal. Nevertheless, the task of offer discovery has been largely ignored by Semantic Web Services frameworks. In this paper, we present a solution for offer discovery that uses WSMO-Lite, the new lightweight semantic Web service annotation framework.

1 Introduction

The Semantic Web is not only an extension of the current Web with semantic descriptions of data; it also needs to integrate services (e.g. e-shops and hotel reservations) that can be used automatically by the computer on behalf of its user. A major technology for publishing services on the Web is the so-called *Web services*. Based on WWW standards HTTP and XML, Web services are gaining significant adoption in areas of application integration, wide-scale distributed computing, and business-to-business cooperation. Still, many tasks commonly performed in service-oriented systems remain manual (performed by a human operator), and services in areas such as business-to-customer (e-commerce) mostly remain available only through a human interface (HTML).

In order to make Web services part of the Semantic Web, the research area of Semantic Web Services (SWS) investigates ways to increase the level of automation around Web services. Typical tasks automated by SWS technologies are discovering available services and composing them to provide more complex functionalities.

SWS automation is supported by machine-processible semantic descriptions that capture the important aspects of the meaning of service operations and messages. SWS descriptions are processed by a semantic execution environment (SEE, for instance WSMX [2]). A user can submit a concrete *goal* to the SEE, which then accomplishes it by finding and using the appropriate available Web services. SWS research focuses mainly on how the SEE "finds the appropriate Web service(s)", as illustrated in Figure 1 with the first four SEE tasks.

In the figure, the user wants to arrange a June vacation in Rome. There are four services with published descriptions: the airline Lufthansa, and hotel reservation services for New York, Rome, and one for the Marriott chain worldwide. The SEE first *discovers*

[⋆] This work is partially funded by the EU research project SOA4All.

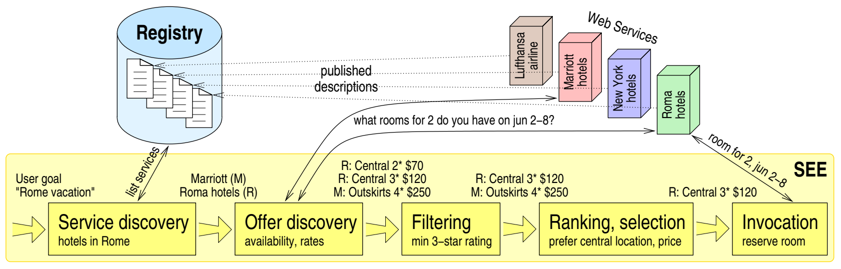

Fig. 1. Semantic Execution Environment (SEE) automation tasks

services that may have hotels in Rome, discarding Lufthansa which does not provide hotels, and the New York service which does not cover Rome. Then the SEE *discovers offers* by interaction with the discovered services. The available offers are a 4* Marriott at the outskirts of Rome, and one 2* and one 3* hotel in the city center. Then the SEE *filters* the offers depending on the user's constraints and requirements (minimum 3-star rating), *ranks* them according to the user's preferences (central location, then price) and *selects* one offer, in the end *invoking* the respective service.

While this example comes from e-commerce, the process of discovery, filtering, ranking and invocation applies in general for any SWS deployment. In some cases, when the semantic description of a service is not sufficient for the SEE to determine whether a service will satisfy the user's goal, discovery is split into service discovery, which finds services that can *potentially* fulfill the goal, and offer discovery, which interacts with the discovered services in order to find out any concrete offers. In [3], we define and motivate offer discovery in more detail.

A typical case where offer discovery is necessary is e-commerce: it would be impractical (or worse) to attempt to describe an e-commerce service completely, as it would require the description to contain an up-to-date catalogue together with availability and delivery information. Instead, the service is described as e-commerce, selling products in certain categories (e.g. hotel or flight reservations in our case above) and perhaps with other information (such as location scope for the New York hotel service); the product catalogue and availability is reachable at runtime through an invocation interface provided by the service.

There are a number of SWS frameworks that support SWS automation; the major two are WSMO [6] and OWL-S [7]. Both are built from the top down, providing semantic descriptions tailored for the expected automation tasks. However, neither of these frameworks supports the particular task of offer discovery; they have to be extended with further constructs, such as a "data-fetching interface" introduced in [9].

Recently, we have proposed WSMO-Lite [8] as a lightweight SWS framework that adds semantic annotations as a layer on top of established Web service technologies. It is built from the bottom up and it provides semantic annotations for WSDL[1], based in a clean model of service semantics, instead of tailoring the semantic constructs towards any particular tasks. As a consequence, WSMO-Lite annotations are more reusable and

[1] Web Service Description Language, `http://w3.org/TR/wsdl20`

they make it easier to realize various SWS automation functionalities. In this paper, we discuss **a concrete realization of offer discovery** which uses WSMO-Lite semantic descriptions.

This paper is structured as follows: in Section 2, we define offer discovery and in Section 3, we discuss the necessary semantic annotations. Section 4 provides details of our implemented offer discovery algorithm. In Section 5, we discuss some related work, and Section 6 concludes the paper.

2 Web service offer discovery

Within the big picture of SWS automation, offer discovery follows the task of service discovery, and its results go into filtering, ranking and selection. Service discovery returns a set of services that can potentially fulfill the user's goal. Offer discovery interacts with these services (or the service providers) and finds out any concrete offers that are relevant to the goal; the result of offer discovery is the set of available offers. This set is then subject to filtering and ranking according to the user's constraints and preferences. In the end, the best offer is selected and *consumed*, i.e. the client invokes the service that gave this offer and, after successful invocation, it will get the offered product or functionality.

In order to be able to talk about offer discovery, we need to specify what we mean by the term "offer". An offer O is a tuple that contains two sets of parameters (data values or types): execution parameters P_x that are required for invoking the service and consuming this offer, and extra parameters P_e that help the client to filter and to rank the offers:

$$O = \langle P_x, P_e \rangle$$

All the parameters (both P_x and P_e) can be used for filtering and ranking of offers; the extra parameters are distinguished from the execution parameters because they are not necessary for consuming the offer. The distinction affects the creation and handling of offers, discussed further on in this paper.

In our hotel scenario, the execution parameters are the start and end dates of the stay, the number and personal data of the guests; and of course the selected hotel reservation service and the concrete hotel. All this data is necessary for making a reservation. The extra parameters for filtering and ranking would be the locations and ratings of the hotels and the room prices. Strictly speaking, the client does not need to know the price when reserving a room, and it certainly does not need to send the price to the service.

Some of the execution parameter values come from the user goal, in our case the data of the guests and the dates of the stay. Further, the particular service that supplies any given offer is itself an execution parameter of the offer. This ensures that the offer is a self-contained construct for the further SWS automation steps: the invocation component knows what service to invoke; and in filtering and ranking the client may express constraints or preferences directly on the services, for instance by building trust with particular providers that delivered good value in the past. Any other offer parameter values come from the offer-discovery interaction with the services.

The split of offer parameters into execution parameters and extra ones allows us to establish identity for offers — two offers are equivalent iff their execution parameter sets are the same:

$$O_1 = \langle P_x^1, P_e^1 \rangle,\ \ O_2 = \langle P_x^2, P_e^2 \rangle : \quad O_1 \equiv O_2 \ \ \Leftrightarrow \ \ P_x^1 = P_x^2 \tag{1}$$

This equivalence relation comes from the fact that only the execution parameters are used in the invocation phase, when an offer is consumed. If two offers vary only in the extra parameters, the service cannot know which of them the client intends to consume. Hence, the offer discovery process must assure that it does not produce different but equivalent offers.

Finally, offer discovery (function *DiscO*) maps a set of discovered Web services $\{S_i\}$ into a set of non-equivalent offers $\{O_j\}$ from these services. Below, we formalize that no pair of offers produced by offer discovery must be equivalent (Eq. 2), and that every offer contains a parameter that ties it to the service that provides it (Eq. 3):

$$\begin{aligned} DiscO(\{S_i\}) \ &\rightarrow \ \{O_j\} \\ \forall o_1, o_2 \in \{O_j\} :\ &\ o_1 \equiv o_2 \Leftrightarrow o_1 = o_2 \qquad (2) \\ \forall o \in \{O_j\},\ o = \langle P_x, P_e \rangle :\ &\ \exists! s \in \{S_i\} : s \in P_x \qquad (3) \end{aligned}$$

Initially, we simplify offer discovery to deal with a single discovered service at a time, but it is possible that a more sophisticated offer discovery implementation can negotiate with multiple services in parallel, and pitch them one against another in order to get better deals; for example, a retailer can promise to match any competitor's price, so the offer discovery process would need to get the competitors' offers first and then use them to get matching counteroffers from the retailer.

Similarly, it is possible that the input to offer discovery for some service comes from offers of another service. For example, a retailer can offer products of varying dimensions, and a delivery service can offer different prices for different sizes. Therefore, offer discovery may also be investigated in the future in context of service composition.

3 WSMO-Lite semantic annotations for offer discovery

Semantic offer discovery should optimally be able to communicate with any Web service that provides operations for finding information about its offers. To achieve this, the offer discovery engine needs a description of the service interface, to see what operations it contains that can be used to gather offer information; and a description of the exchanged data, to understand the offers and to be able to compare them against the goal. The WSMO-Lite SWS description framework [8], together with the underlying SAWSDL standard[2], provide all the necessary semantic descriptions.

Any Web service, described in WSDL, has an *interface* which consists of a number of *operations*. Web service interfaces often intermix operations for offer inquiry with operations that actually provide the resulting product or service, for instance a hotel reservation service would provide the availability inquiry operations along with

[2] Semantic Annotations for WSDL and XML Schema, `http://w3.org/TR/sawsdl`

the operations for making reservations. For the purposes of automated offer discovery, we will use operations that only provide information and do not have any significant side-effects. In other words, we need what the Web architecture [1] calls "safe interactions[3]". WSMO-Lite allows operation annotations with functionality categories, and the WSDL 2.0 defines an operation safety flag[4], which we treat as a WSMO-Lite category for safe operations.

It remains to be seen whether all safe operations can be treated as "offer-inquiry" operations, but due to their safety, there is no harm in invoking such operations even if they do not actually help get information about the service offers.

Further, WSMO-Lite annotates operation inputs and outputs with pointers to ontology entities, such as RDFS classes. These annotations allow the semantic client to match its goal data against the inputs of the offer-inquiry operations, and to match the goal and offer data against the inputs of the execution operations[5], to distinguish between execution and extra parameters.

While WSMO-Lite describes services, the modeling of user goals is outside its scope. The representation of goals depends on the concrete SEE that implements SWS automation; in our work, we use WSMX as the SEE, so goals are described in WSMO.

In summary, WSMO-Lite provides semantic service descriptions and WSMO provides goal descriptions, which are together sufficient for us to realize an automated offer discovery process, described in the following section. Alas, due to space constraints we cannot provide an example annotated WSDL document or an example user goal which would clarify what the annotations look like in practice.

4 Offer discovery algorithm and implementation

Algorithm 1 shows how we have realized offer discovery on top of the semantic annotations described above. It is an initial working algorithm that treats the discovered services independently; this is not a serious limitation, though, as the currently available public Web services do not even allow more complex multi-party negotiations.

In short, the algorithm starts by creating an initial offer from the goal data, then it keeps invoking the offer-inquiry operations for every offer that does not provide all the required execution parameters (so-called *incomplete offers*).

To get from the goal data to all the required execution parameters for each offer, we use *planning* (cf. [5]). The currently known data (from the goal and from the current offer) becomes the initial state of the planning problem, and the presence of all execution parameters is the goal state. In our current implementation, we represent the presence of a value of a certain type with a predicate EXISTS(*type*), which is also the only predicate in the planning problem. The offer-inquiry operations are the possible transitions: the precondition of an operation is that values exist for all the input types, and the effect is

[3] Information retrieval is the canonical example of a safe interaction: the client may query a service about the availability of hotel rooms, yet by issuing the query the client makes no commitment to book the room.

[4] `http://www.w3.org/TR/wsdl20-adjuncts/#safety`

[5] Execution operations are those operations that are used when invoking a Web service in order to consume an offer.

that values exist for all the output types. Note that the offer inquiry operations do not have negative effects (or *delete lists*) because we currently assume monotonic behavior where new information cannot invalidate what was known earlier.

When a plan is found for a given incomplete offer, the first operation in this plan is invoked (its inputs must be available because otherwise it could not be the first operation of the plan). The outputs of the operation are added to the offer parameters. A single offer inquiry operation may return a list of offers; for instance, `findAvailableHotels(dates,guests)` can return multiple hotels at a time. This is detected if multiple instances in the output data are of the same type as some needed execution parameter. We assume that a single parameter in one offer can only have a single value (which may be structured and complex), therefore on line 16, the initial incomplete offer is cloned as many times as necessary, each of the clones getting a single value from the received outputs of the same type.

In the hotel scenario, booking a room requires the dates and guest data (from the user goal) and a selected hotel, so the algorithm would invoke an operation such as `findAvailableHotels(dates,guests)`, and the resulting list of hotels would complete the list of offers.

In the end (line 17), the algorithm tries to gather further extra parameters for all the known offers. This is necessary because the main algorithm only satisfies the execution parameters, so it would never invoke an operation such as `getPrice(hotel,dates)`, because the resulting *price* value is not an execution parameter. We are working on heuristic approaches for selecting the operations to invoke for the extra parameters.

Algorithm 1: Offer discovery for a single service

```
   Input: Service S whose offers should be discovered, user goal G.
   Result: The set of offers provided by S.
1  set up initialOffer with execution parameter values from G, S
2  set up incompleteOffers = {initialOffer} and completeOffers = {}
3  identify offerInquiryOperations from S
4  while incompleteOffers is not empty do
5      offer = first(incompleteOffers)
6      if offer is complete (all execution parameters are present) then
7          move offer to completeOffers; continue while
8      set up knowledgeBase from offer and G
9      select neededOfferExecutionParams — execution parameters not satisfied by offer
10     plan = planOperations(
11                 initial state: knowledgeBase,
12                 goal state: neededOfferExecutionParams,
13                 operations: offerInquiryOperations)
14     if empty(plan) then eliminate offer; continue while
15     receivedValues = invoke(S, plan[1], knowledgeBase)
16     add receivedValues to offer (possibly splitting it into multiple offers)
17 gatherExtraParameters(completeOffers)
18 return completeOffers
```

The execution parameters mentioned on line 1 are the input parameters of the execution operations. Currently, we treat all non-safe operations as the execution operations, and we attempt to satisfy all their inputs. However, it is necessary to select only those execution operations that are relevant for the user goal: for instance, a hotel reservation service may have an operation `cancelReservation(reservationCode)` which is not going to be invoked when booking a room, and whose input parameter cannot be satisfied by discovering offers. Our algorithm would fail to find any offers here.

We have implemented the algorithm within the WSMX system [2] with positive initial results, but a larger evaluation experiment remains as future work. While we use WSMO goal descriptions, we do not support the full complexity of WSMO goals yet. And finally, the planning algorithm used in our implementation is not semantic (for instance, it cannot take into account any subclass relationships between the parameters and the available values); this limits the expressivity available for the ontologies used to describe the input and output data of the service operations. We are looking for suitable ways of extending the planning part with semantic reasoning.

5 Related work

While service discovery, filtering and ranking have been extensively researched, offer discovery has been seriously investigated only by two research teams other than ours, as far as we know.

Zaremba *et al.* [9, 10] talk about a so-called "contracting interface" with an explicit choreography that guides the execution of this interface. In their case, the WSMX client follows the predefined choreography to find out the concrete offers provided by a discovered Web service. The contracting interface can be likened to a prescribed protocol for offer discovery. In our work, we choose to lower the semantic description burden (we do not require a choreography description) by employing planning over available safe operations. Our current solution could not use unsafe operations, while they could be included in Zaremba's contracting interface. If this proves to be a real-world limitation, we will attempt to add other simple annotations that would mark operations as suitable for offer discovery.

Küster *et al.* [4] models service requests and advertisements as data graphs, which can be mapped structurally, and which can contain input and output variables. The interaction with services is broken down into two phases: *estimation* and *execution*; with input and output variables assigned for either of them. To guide the invocation of the estimation operations, the variables are marked with a simple numeric order in which they are to be used. From the published works, it is unclear how exactly their work ties to concrete Web services and operations, but Küster's approach seems in effect similar to Zaremba's: it requires more complex semantic descriptions than our approach, in exchange for potentially greater expressivity. However, Küster's simple numerical ordering of estimation invocations may be limiting when compared to Zaremba's more generic choreographies.

6 Conclusions

Web services are a necessary part of the Semantic Web, and research on Semantic Web Services aims to automate their use. Among the tasks that can be automated using semantic technologies is offer discovery, especially important for e-commerce applications. In this paper, we have presented a formalization of offer discovery and we have shown an offer discovery algorithm and its implementation using WSMO-Lite semantic descriptions.

While our prototype gives positive results, there are some open points in need of solutions, and we still need to perform a proper evaluation experiment for our approach.

References

1. Architecture of the World Wide Web. Recommendation, W3C, December 2004. Available at `http://www.w3.org/TR/webarch/`.
2. A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler. WSMX – A Semantic Service-Oriented Architecture. *International Conference on Web Services (ICWS 2005)*, July 2005.
3. J. Kopecký, E. Simperl, and D. Fensel. Semantic Web Service Offer Discovery. In *Proceedings of Service Matchmaking and Resource Retrieval in the Semantic Web Workshop, colocated with* 6^{th} *ISWC*, 2007.
4. U. Küster and B. König-Ries. Supporting dynamics in service descriptions — the key to automatic service usage. In *Proceedings of the Fifth International Conference on Service Oriented Computing (ICSOC07)*, Vienna, Austria, September 2007.
5. D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
6. D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.
7. The OWL Services Coalition. OWL-S 1.1 Release. Available at `http://www.daml.org/services/owl-s/1.1/`, November 2004.
8. T. Vitvar, J. Kopecký, J. Viskova, and D. Fensel. WSMO-Lite Annotations for Web Services. In *Proceedings of the 5th European Semantic Web Conference (ESWC)*, Tenerife, Spain, 2008.
9. T. Vitvar, M. Zaremba, and M. Moran. Dynamic service discovery through meta-interactions with service providers. In E. Franconi, M. Kifer, and W. May, editors, *ESWC*, volume 4519 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2007.
10. M. Zaremba, T. Vitvar, M. Moran, and T. Hasselwanter. WSMX Discovery for SWS Challenge. SWS Challenge Workshop, Athens, Georgia, USA, November 2006.