Dynamic Service Discovery through Meta-Interactions with Service Providers*

Tomas Vitvar, Maciej Zaremba, and Matthew Moran

Digital Enterprise Research Institute (DERI),
National University of Ireland, Galway
{tomas.vitvar, maciej.zaremba, matthew.moran}@deri.org

Abstract Dynamic discovery based on semantic description of services is an essential aspect of the Semantic Web services integration process. Since not all data required for service discovery can always be included in service descriptions, some data needs to be obtained during run-time. In this paper we define a model for service interface allowing required data to be fetched from the service provider during discovery process. We also provide a specification of such interface for WSMO and demonstrate the model on a case scenario from the SWS Challenge implemented using WSMX – a middleware platform built specifically to enact semantic service oriented architectures.

1 Introduction

The Web has a volatile nature where there can only be a limited guarantee of being able to access any specific service at a given time. This leads to a strong motivation for discovering and binding to services at run-time (*late binding*). Existing XML-based WSDL descriptions of data, messages, or interfaces are insufficient or provide limited expressivity for machines to understand. Service discovery operating on semantic descriptions offer the potential of flexible matching that is more adaptive to changes over services' lifetime. In general, discovery matches definitions of user requests (goals) with those of offered services. Different levels of match are possible e.g. subsumption match, plug-in match, exact match etc.[13,4]. Semantic discovery works on the abstract definitions of services and goals (containing no instance data). This needs to be further elaborated to achieve more accurate results. For example, a request to "buy a Harry Potter book" involves first searching for descriptions of services that sell books, but which then determining if the service sells Harry Potter books and if those books are in stock. Taking Amazon as an example, it is clearly unfeasible to include data for the entire catalogue and its availability directly in the service description. Such information has a dynamic character and therefore should only be fetched from the service at discovery-time.

* *This work is supported by the Science Foundation Ireland Grant No. SFI/02/CE1/I131, and the EU projects Knowledge Web (FP6-507482), DIP (FP6-507483) and SUPER (FP6-026850).*

For this purpose, we propose a general mechanism enabling the definition of an interface on the service to allow the *fetching* of required data from the service during the late binding phase (e.g during service discovery, contracting/negotiation, selection etc.). These tasks are performed in a semantic service environment in a (semi) automated fashion by means of the "intelligence" of intermediary (middleware) services. We define a model for the service interface which provides a mechanism to fetch data from the service provider during the discovery process. Choosing the Web Service Modeling Ontology (WSMO) as our conceptual model, we define an extension for this interface and demonstrate this work through a case scenario of the SWS Challenge[1] implemented using WSMX – a middleware platform built specifically to enact semantic service oriented architectures.

In section 2 we introduce the underlying specifications for our work, namely WSMO, WSML and WSMX providing a conceptual framework, ontology language and execution environment for Semantic Web services. In section 3 we define a model for a service interface and algorithm to fetch data for service discovery and further show how this model can be specified using WSMO service model. In section 4 we illustrate the model on the case scenario implemented in the WSMX environment and describe the evaluation for the implementation. In section 5 we describe related work and in section 6 we conclude the paper and indicate our future work.

2 Semantic Web Services and WSMO

A general aim of Semantic Web Services is to define a semantic mark-up for Web services providing the higher expressivity then traditional XML-based descriptions. One of the initiatives in the area is the Web Service Modeling Ontology (WSMO)[11]. WSMO provides a conceptual model describing all relevant aspects of Web services in order to facilitate the automation of service discovery, composition and invocation. The description of WSMO elements is represented using the Web Service Modeling Language (WSML)[11] – a family of ontology languages – which consists of a number of variants based on different logical formalisms and different levels of logical expressiveness. WSMO also defines the conceptual model for WSMX[9], a Semantic Web Services execution environment. Thus, WSMO, WSML and WSMX form a coherent framework for modeling, describing and executing Semantic Web Services. The WSMO top-level conceptual model consists of *Ontologies*, *Web Services*, *Goals*, and *Mediators*.

- **Ontologies** provide the formal definition of the information model for all aspects of WSMO. Two key distinguishing features of ontologies are, the principle of a shared conceptualization and, a formal semantics (defined by WSML in this case). A shared conceptualization is one means of enabling information interoperability across independent Goal and Web service descriptions.

[1] http://sws-challenge.org

- **Web Services** are defined by the functional capability they offer and one or more interfaces that enable a client of the service to access that capability. The Capability is modeled using preconditions and assumptions, to define the state of the information space and the world outside that space before execution, and postconditions and effects, defining those states after execution. Interfaces are divided into *choreography* and *orchestration*. The choreography defines how to interact with the service while the orchestration defines the decomposition of its capability in terms of other services.
- **Goals** provide the description of objectives a service requester (user) wants to achieve. WSMO goals are described in terms of desired information as well as "state of the world" which must result from the execution of a given service. The WSMO goal is characterized by a requested capability and a requested interface.
- **Mediators** describe elements that aim to overcome structural, semantic or conceptual mismatches that appear between different components within a WSMO environment. Although WSMO Mediators are essential for addressing the requirement of loosely coupled and heterogeneous services, they are out of the scope of our work at this point.

In this paper, we will elaborate on WSMO service definition and in particular on its service interface. The service interface defines choreography as a formal description of a communication pattern the service offers. Two types of such choreography interfaces are defined: (1) *execution choreography* used during the execution phase where the functionality of the service is consumed by a service requester and (2) *meta-choreography* used during late binding to get additional information necessary for communication with the service. Since the WSMO model is open, such definitions may be extended to be used for the particular tasks of the late binding phase. We focus on the definition of one such extension for use for the service discovery phase.

3 Data Fetching for Discovery

A Web service capability can be described in terms of results potentially delivered by the service. A goal describes its capability as the information the user wants to achieve as a result of service provision. We denote the description of the Web service and the goal as $\mathcal{W}$ and $\mathcal{G}$ respectively. For the $\mathcal{W}$ and $\mathcal{G}$ we also introduce the data of these descriptions which we denote $\mathcal{D}_{\mathcal{W}}$ and $\mathcal{D}_{\mathcal{G}}$ respectively and which is provided *directly* as part of their respective descriptions.

For purposes of our work and based on grounds of [4,13], we define the following three basic steps when the matching of a goal $\mathcal{G}$ and a Web service $\mathcal{W}$ needs to be performed: (1) *Abstract-level match*, (2) *Instance-level Match*, (3) *Data Fetching*. Abstract-level Match operates on abstract descriptions of $\mathcal{G}$ and $\mathcal{W}$ without their data being taken into account. The matching is defined by the following set-theoretic relationships between objects of $\mathcal{G}$ and $\mathcal{W}$: (1) exact match, (2) subsumption match, (3) plug-in match, (4) intersection match and

(5) disjointness. If the goal and the Web service match (relationships 1-4), it is further checked if the service can provide a concrete service by consulting the data of the goal and the service (Instance-level Match). If all data is not available for step 2, the data needs to be obtained from the service (Data Fetching). Later in this section we further formalize these steps and define the algorithm. For step 1 and step 2, we define a matching function as follows:

$$s \leftarrow matching(\mathcal{G}, \mathcal{W}, \mathcal{B}_{gw}), \tag{1}$$

where $\mathcal{G}$ and $\mathcal{W}$ is a goal and a service description respectively and $\mathcal{B}_{gw}$ is a common knowledge base for the goal and the service. The knowledge base contains data which must be *directly* (through descriptions $\mathcal{G}$ and $\mathcal{W}$) or *indirectly* (through data fetching) available so that the matching function can be evaluated. The result s of this function can be: (1) *match* when the match was found at both abstract and instance levels (in this case all required data in $\mathcal{B}_{gw}$ is available), (2) *nomatch* when the match was not found either at abstract level or at instance level (in this case all required data in $\mathcal{B}_{gw}$ is available), or (3) *nodata* when some required data in $\mathcal{B}_{gw}$ is not available and thus the matching function cannot be evaluated.

We further assume that all data for the goal is directly available in the description $\mathcal{G}$. The data fetching step is then performed for the service when the matching function cannot be evaluated (the result of this function is *nodata*). We then define the knowledge base as:

$$\mathcal{B}_{gw} = \mathcal{D}_{\mathcal{G}} \cup \mathcal{D}_{\mathcal{W}} \cup \{y_1, y_2, ..., y_m\}, \tag{2}$$

where $\{y_i\}$ is all additional data that needs to be fetched from the service in order to evaluate the matching function.

Based on the representation of service interface using abstract state machines[12], we define the *data fetch interface* for service $\mathcal{W}$ as

$$\mathcal{I}_{\mathcal{W}} = (in(\mathcal{W}), out(\mathcal{W}), L), \tag{3}$$

where $in(\mathcal{W})$ and $out(\mathcal{W})$ denotes input and output vocabularies which correspond to input and output data of the interface $in(\mathcal{I}_{\mathcal{W}})$ and $out(\mathcal{I}_{\mathcal{W}})$ respectively, and L is a set of transition rules. The matching function can be then evaluated if

$$\forall y_i \in out(\mathcal{I}_{\mathcal{W}}) : \exists x \in \mathcal{B}_{gw} \wedge x \in in(\mathcal{I}_{\mathcal{W}}) \qquad i = 1, 2, \ldots, m. \tag{4}$$

According to 4, data $\{y_i\}$ can be fetched from the service through the data fetch interface if input data $in(\mathcal{I}_{\mathcal{W}})$ is either initially available in the knowledge base $\mathcal{B}_{gw}$ (data directly available from the goal or web service descriptions) or the input data becomes available during the processing of the interface. For a rule $r \in L$ we denote $r.ant$ and $r.con$ as the rule *antecedent* and the rule *consequent* respectively. The antecedent $r.ant$ defines an expression which if holds during

run-time in the *memory*[2], the memory is modified according to the definition of an action in *r.con* (i.e. specified data is *added*, *updated* or *removed* from the memory) (see the algorithm in section 3.1). Please note that each concept of the vocabulary $in(\mathcal{W})$ and $out(\mathcal{W})$ has defined *grounding* to respective message in WSDL. Through this grounding definition it is possible to invoke WSDL operation when instance data of the concept is to be added or updated in the memory (and thus the data is fetched from the service). This definition of the grounding is described in [6].

3.1 Algorithm

In algorithm 1, the matching function is integrated with data fetching. The algorithm operates on inputs, produces outputs and uses internal structures as follows:

Input:

- Repository $Q = \{W_1, W_2, ..., W_n\}$, where $W \in Q$ is the web service description. For each web service W we denote D_W as data of the web service and I_W as data fetching interface of the web service with rule base L. This interface is defined according to definition 3 and its description is optional for the web service. In addition, for each rule $r \in L$ we denote action of the rule consequence as *r.con.action* and its corresponding data as *r.con.data*.
- Goal description G for which we denote D_G as data of the goal.

Output:

- List $E = \{W_1, W_2, ..., W_m\}$, where $W_i \in Q$ and W_i matches G (the result of the matching function for W_i and G is *match*).

Uses:

- Processing memory M containing data fetched during processing of the rules of the data fetching interface.
- Knowledge base B_{gw} which contains data for processing of the matching function.
- Boolean variable $modified$ indicating whether the knowledge base has been modified or not during the processing.

The algorithm performs the matching of the goal with each Web service in the repository using the matching function (line 7). If the matching cannot be evaluated (the result is *nodata*), the algorithm tries to fetch data from the service by processing the service's data fetch interface. Whenever the new data is available from the service, the algorithm updates the knowledge base and process the matching again. This cycle ends when no data can be fetched from the interface or matching can be evaluated (the result is *match* or *nomatch*).

[2] We use the term memory to denote a processing memory through which states of an abstract state machine are maintained during its processing.

Algorithm 1 Data Fetching for Discovery

```
1: E ← ∅
2: for all W in Q do
3:    B_gw ← D_G ∪ D_W
4:    M ← B_gw
5:    repeat
6:       modified ← false
7:       s ← matching(G, W, B_gw)
8:       if s = nodata and exists(I_w) then
9:          while get r from L: holds(r.ant, M) and not modified do
10:            if r.con.action = add then
11:               add(r.con.data, M)
12:               add(r.con.data, B_gw)
13:               modified ← true
14:            end if
15:            if r.con.action = remove then
16:               remove(r.con.data, M)
17:            end if
18:            if r.con.action = update then
19:               update(r.con.data, M)
20:               update(r.con.data, B_gw)
21:               modified ← true
22:            end if
23:         end while
24:      end if
25:   until s ≠ nodata or not modified
26:   if s = match then
27:      E ← E ∪ W
28:   end if
29: end for
```

In case the matching is evaluated as *match*, the web service is added to the list of matched services and the cycle is performed for the next service in the repository.

During the processing of the interface, the algorithm allows to hook in a matching function which is called whenever the new data is available from the service. The algorithm uses independent memory (memory M) from the knowledge base (B_{gw}) for processing of the data fetching interface. This allows that already-obtained data cannot be removed from the knowledge base while, at the same time, correct processing of the interface is ensured. The memory M is used not only for data but also for control of interface processing (in general, the content of the memory does not need to always reflect the content of the knowledge base). According to the particular interface definition, the data can be fetched step-wise allowing minimizing of the interactions with the service during discovery. This also follows the strong decoupling principle when services are described semantically and independently from users' requests. For example, during service creation phase a service provider (creator) does not know which particular

data will be required for particular data fetching (in general, matching with a goal could require some or all defined data which depends on the definition of the request). The interface defined using rules allows to get only data which is needed for the matching (for example in some cases only price is needed, in some cases price and a location of selling company could be needed if offered by the service).

3.2 WSMO Service Interface for Data Fetching

As stated in [11], Web Service interface defines *choreography* and *orchestration* allowing the modeling of external and internal behavior of the service respectively. In this respect, the interface for data fetching follows the WSMO service interface describing a meta-choreography which allows additional data to be obtained from the service for the discovery process. WSMO service will thus have additional interface defined (WSMO service allows multiple interface definitions). This interface will however only use the choreography describing a meta-choreography for obtaining additional data for the discovery process. We do not specify *orchestration* for this interface as the logic of how data fetch is performed by the service (how data is obtained out of aggregation of other services) is not of interest for discovery and we do not use it in our algorithm. In order to distinguish between the interface used for data fetching and the interface used for execution (defining how actual service is consumed by the service requester), we use non-functional property. For purposes of our work we further use non-functional property *interfacePurpose* with values *execution* and *discovery*. Another possibilities for distinguishing both interfaces would be to define data-fetch interface as specialization of WSMO service interface. The decision on whichever approach will be used will be done in the context of the WSMO WG.

4 Implementation and Evaluation

The model introduced in this paper has been implemented and evaluated through the SWS Challenge discovery scenario. The scenario introduces five different service providers offering various purchasing and shipment options. They provide different availability and pricing for their services with constraints on package destination, weight, dimension and shipment date. Service requesters with different requirements search for the best offers with packages of different weight and shape. Our model for data fetching for discovery fits well into this scenario since not all information can be provided in service descriptions meaning they must be dynamically obtained at discovery-time. In this section we base examples on the Mueller service whose price information is not available in the service description and needs to be fetched during the service discovery via data fetching interface. In section 4.3 we further describe the evaluation of our implementation from the broader context of the SWS Challenge requirements.

4.1 Scenario and Assumptions

In the scenario depicted in figure 1, a user accesses the e-marketplace called Moon where a number of companies such as Mueller or Racer have registered their services. The Moon runs a (1) Web portal through which it provides services to users and (2) the WSMX middleware system through which it facilitates the integration process between users and service providers.

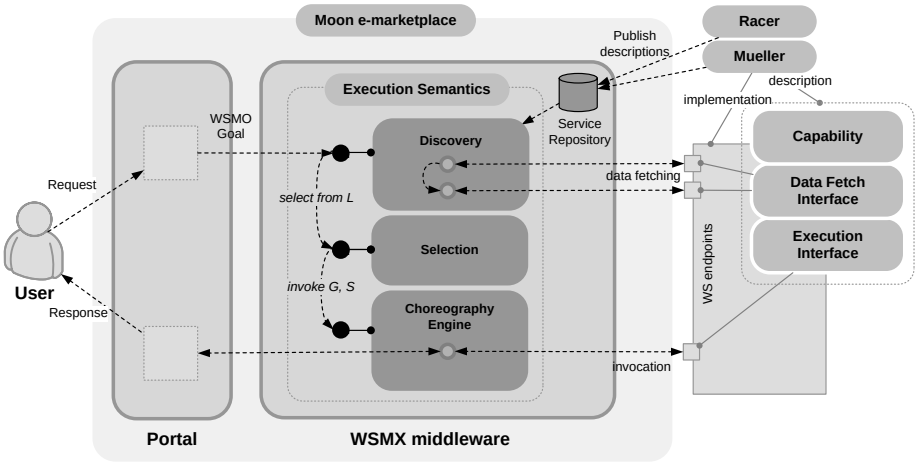

Figure 1. Architecture for the Scenario

For this scenario and the aims of this paper we make the following assumptions.

- Service providers and Moon use the WSMO formalism for Web service description. We assume that service requesters maintain their own adapters to their back-end systems while at the same time providing lifting/lowering functionality between their existing technology (e.g. WSDL, XML Schema) and WSML.
- All service providers adopt a common ontology maintained by the Moon e-marketplace. Handling data interoperability issues, where service providers and Moon use different ontologies, is out of the scope of this paper.
- During execution, interactions between the user and the Moon are simplified to a single request-response exchange. Either the user submits a goal (our scenario) or a pre-selected service for invocation. Meta-interactions between users and the middleware system are not of our interest in this work.

In our scenario, a user defines her requests through the Web portal's user interface. The Web portal generates a WSMO goal corresponding to the request, sends it to WSMX, receives the response and presents the result back to the user. The execution process, run in WSMX after the receipt of the goal, includes discovery (with data-fetching from services), selection of the best service and

invocation. Although the whole process of this scenario is implemented, the contribution of this paper lies in the integration of data fetching with discovery. Other parts, i.e. selection and invocation are not described in detail here.

4.2 Modeling Ontologies, Goals and Services

In order to implement the scenario, we first need to create semantic models for ontologies, goals and services. We describe these models in the following subsections. We present examples of ontologies, services and goals definitions in WSML using the following prefixes to denote their respective namespaces: *mo* – common ontology, *gl* – goal ontology.

Ontologies describe concepts used for the definition of goals and services. In our scenario we use a common scenario ontology with additional ontologies to define specific axioms or concepts used by the descriptions of services and/or goals.

The common ontology defines shared concepts used in the description of the goal and services, such as *Address*, *ShipmentOrderReq*, *Package*, etc. In addition, we use the common ontology to specify named relations for services and goals. Specific ontologies for goals and services declare axioms that define the relations to represent their conditions. An analogy for this approach are interfaces in programming labguages like Java. The interface declares some functionality but does not say how this should be implemented. Using this approach, we define a set of relations in the common ontology which represent the axioms that a service may need to define. Listing 1.1 shows the simple definition for the *isShipped* relation from the common ontology and its implementation in the Mueller's ontology.

```
1  /* isShipped relation in the common ontology */
2  relation isShipped(ofType mo#ShipmentOrderReq)
3
4  /* implementation of the isShipped relation in the Mueller's ontology */
5  axiom isShippedDef
6      definedBy
7        ?shipmentOrderReq[mo#to hasValue ?to, mo#package hasValue ?package] memberOf mo#
               ShipmentOrderReq and
8        ?to[mo#city hasValue ?city] and
9        isShippedContinents(?city, mo#Europe, mo#Asia, mo#Africa) and
10       ((?package [mo#weightKg hasValue ?weightKg] memberOf mo#Package) and (?weightKg < 50))
11     implies
12       mo#isShipped(?shipmentOrderReq).
```

Listing 1.1. *isShipped* relation

The relation *isShipped* is true if the service provider can ship products according to the shipment order request (*ShipmentOrderReq*). In the second part of the listing 1.1, *isShipped* is defined such that the destination city for the shipment must be in Europe, Asia or Africa and the weight of the package is less then 50kg. This forms part of the Mueller service description.

Services. We focus on the description of the data-fetching interface of the Mueller service showing how and which data can be fetched during discovery.

```
1  interface WSMullerDataFetchInterface
2    nfp
3      _"interfacePurpose" hasValue "discovery"
4      ...
5    endnfp
6
7    choreography WSMullerDataFetchChoreography
8      ...
9      transitionRules WSMullerDataFetchTransitionRules
10       /* Rule 1: Request for product quote */
11       forall {?purchaseQuoteReq} with (
12         ?purchaseRequest memberOf mo#PurchaseQuoteReq
13       ) do
14         add(_# memberOf mo#PurchaseQuoteResp)
15       endForall
16
17       /* Rule 2: Request for shipment quote */
18       forall {?shimpmentQuoteReq} with (
19         ?purchaseQuoteResp[mo#package hasValue ?package] memberOf mo#PurchaseQuoteResp and
20         ?shipmentQuoteReq[mo#to hasValue ?to] memberOf mo#ShipmentOuoteReq and
21         mo#isAvailable(?purchaseQuoteResp) and mo#isShipped(?to, ?package)
22       ) do
23         add(_# memberOf mo#ShipmentQuoteResp)
24       endForall
```

Listing 1.2. Mueller data fetching interface

In listing 1.2, the first rule (line 6) describes how to get the price and the product availability information (the quote request data is part of the goal description). The second rule (line 13) describes how to get a quote for shipment. This rule will be only used if requested product is available (determined through relation *isAvailable*) and Mueller can ship to specified address (determined through relation *isShipped*). Here, shipment address (*to* variable) is taken from the shipment quote request and packaging information (*package* variable) is taken from purchase order response. According to this definition, the first rule is used independently (and could be the only rule used where the user does not request shipment) while for the second rule, the first rule need to be executed first (the rule can be executed if the product is available and shippable which is determined through results of the first rule). Concepts *PurchaseQuoteReq*, *ShipmentOuoteReq* and *PurchaseQuoteResp*, *ShipmentQuoteResp* are defined as input and output vocabularies respectively (including grounding mechanism) (for brevity, this is not shown in the listing).

Goals The goal for the scenario describes the user's aim to buy certain products and ship them to a specific location. In addition, the goal specifies a preference that price be used for selection of the best service where multiple matching services are discovered. The goal as in listing 1.3 is defined for our scenario with respect to the implementation of the matching function from section 3 (we discuss this implementation in section 4.3). The goal defines the capability postcondition specifying to get a quote for the product while at the same time the product must be available and shippable to location specified by the shipment order request.

```
Goal GoalPurchaseShip
  nfp
    _"preference" hasValue "?price"
    ...
  endnfp
  ...
  capability GoalPurchaseShipCapability
    postcondition
      definedBy
        ( ?x[mo#price hasValue ?price] memberOf mo#PriceQuoteResp and
          mo#isAvailable(go#purchaseOrderReq) and
          mo#isShipped(go#shipmentOrderReq) ).
  ...
```

Listing 1.3. User Goal in WSMO

4.3 Implementation

The scenario is implemented as follows: when the goal is generated out of the request specified by the user, it is sent to the WSMX system. The WSMX starts a new operational thread (execution semantics) which first invokes the discovery component which in turn returns a list of services matching the goal. This list is passed to the selection component to select the service that best fits the user request. Control passes to the choreography engine which uses the choreography descriptions of the goal and service respectively, to drive the message exchange with the discovered service. This section describes the implementation of the algorithm from section 3 within the discovery component of WSMX. The details about other parts of the execution process can be found in our previous work in [2].

Section 3 describes three steps for discovery. A prototype for the *abstract-level* matching is under development in the WSMO working group. The implementation, described here, focuses on the steps of *instance-level* matching and *data-fetching*. A match between the goal and Web services is determined on the knowledge base created out of their descriptions, including instance data (both available from the descriptions and fetched). The goal capability defines a query (listing 1.3) which is used to query the knowledge base.

According to the algorithm 1 in section 3, the knowledge base B_{gw} is created for every goal and web service from the repository as shown in figure 2. Initially, the knowledge base imports all concepts from the common ontology and data from both goal and web service descriptions. In order to evaluate the matching function, we simply query the knowledge base using the goal postcondition. If the result of the evaluation is *true*, we add the web service to the list E of web services to the position determined by the preference. If the result of the evaluation is $false$, we first try to fetch new data by processing the fetching interface. If new data is available we evaluate the matching function again. If new data is available, the matching function is evaluated again. Otherwise, the cycle ends and the next service from the repository is processed. We briefly discuss this implementation in the next section 4.4.

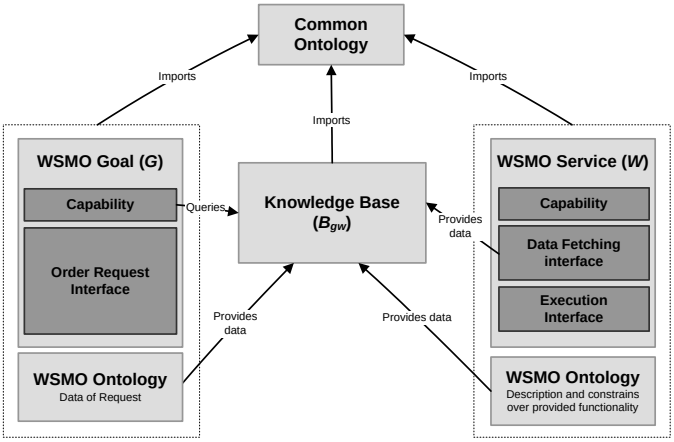

Figure 2. Knowledge Base B_{gw}

4.4 Evaluation

Our implementation has been evaluated according to the methodology defined by the SWS Challenge. The SWS Challenge is an initiative led by a Semantic Web Services community providing a standard set of increasingly difficult problems, based on industrial specifications and requirements. Entrants to the SWS Challenge are peer-evaluated to determine if semantically-enabled integration approaches reduce costs of establishing and maintaining the integration between independent systems. In each SWS challenge workshop, the entrants first address introduced initial scenario of particular problem (e.g. mediation, discovery) in a testing environment prepared by the SWS Challenge organizers. The organizers then introduce some changes to back-end systems of the testing environment when the adaptivity of solutions is evaluated – solutions should handle introduced changes by modification of declarative descriptions rather than code changes. This evaluation is done by a methodology, developed by the SWS Challenge organizers and participants, which identifies following so called *success levels*. *Success level 0* indicates a minimal satisfiability level, where messages between middleware and backend systems are properly exchanged in the initial scenario. *Success level 1* is assigned when changes introduced in the scenario require code modifications and recompilation. *Success level 2* indicates that introduced changes did not entail any code modifications but only declarative parts had to be modified. *Success level 3* is assigned when changes did not require either modifications to code or the declarative parts, and the system was able to automatically adapt to the new conditions. Our implementation was evaluated to successfully address the scenario based on the location, weight, dimensional weight and price requirements, scoring success level 2. The implementation proved to be generic where only modifications of the WSMO Goals were necessary in order to correctly handle introduced changes. Discovery based on

the location was successfully resolved using the common *isShipped* relation (see listing 1.1). Additional criteria imposed on the service such as weight and price have also been evaluated to level 2. No changes in WSMX code or in the descriptions of the services were required – only the Goal requests had to be changed. With respect to the fully-fledged discovery, there are however some limitations. It does not distinguish between the result *nodata* and *nomatch* (as defined in the algorithm) while it treats both results as *nodata*. This means that the whole fetching interface needs to be always processed until new data can be fetched or unless the match is found. This is a forced limitation of our implementation while at the same time it is a temporary solution for our environment before the fully-fledged discovery component will be available. The algorithm presented here however allows to use various implementations of matching functions which adhere to its defined interface. As a consequence, our solution currently offers a limited scalability. It might generate a significant network overhead in large-scale discovery scenarios when detailed interactions with every potential web service needs to be performed. We plan to address the optimality of our algorithm with respect to scalability in our future work. Our current solution also does not directly address security. It is important to ensure that information retrieved from service provider can be accessed after authorization and that data is fetched in a secure way. Such security aspects should be however implemented between the e-marketplace and service providers transparently to data fetching.

5 Related Work

There is no directly comparable work in the extension of the interface description in Web services to allow the fetching of additional data to aid discovery at run-time. However, there are two topics that are closely related. The first is service discovery based on semantic matchmaking which is the research area in which this paper is set and the second is service contracting and negotiation. Research into Goal-based discovery for WSMO and WSMX takes a stepwise approach with both theoretical and implementation concerns addressed at each stage. Three strategies have been investigated in this manner. The first is keyword-based discovery [4], which uses an algorithm that matches keywords, identified from the Goal description, with words occurring in various locations within the Web service description. The second strategy is for a lightweight Semantic Web Services discovery component for the WSMX platform and is described in [1]. This approach models a service in terms of the objects it can deliver. The term object, in this sense, means something of value the service delivers in its domain of interest. A third strategy is based around the use of quality-of-service attributes as described in [3] and implemented by the authors as a WSMX component. Upper level ontologies describing various domains for quality-of-service attributes are provided and non-functional properties are introduced to the service descriptions whose meanings are defined in these QoS ontologies. The approach in this paper is compatible with each of the matching strategies as it extends the matching power by requesting data from the service

that is not directly available in its description. In [7], contracting is identified as an activity that may take place between the requester and provider once the initial discovery has identified candidate services. The discovery mechanisms in OWL-S rely on subsumption reasoning to match a service profile of service requesters with candidate service profiles published by service providers ad described in [10]. As with the WSMO efforts, they acknowledge that a *negotiation* phase may be necessary after discovery to allow requesters and providers agree on quality of service issues. Automated negotiation of service provision is related to the topic of this paper as, for negotiation to take place, it must be possible to determine during discovery exactly the terms that are being offered by the service which may be open to negotiation. A substantial body of work is devoted to the development of negotiation systems ranging from the application of intelligent agents for eCommerce in [8] through negotiation using Bayesian Learning [14] to using Web services and BPEL for automated negotiations [5].

6 Conclusion and Future Work

Service discovery which operates on abstract descriptions of services needs to be further elaborated in order to return results of concrete services satisfying concrete goals. For this purpose, instance data needs to be used. Since all data can not be included in service descriptions (usually for practical reasons) it needs to be fetched from the service provider at discovery-time. In this paper we presented an approach to model the service interface allowing such data to be fetched from the service provider. We use the abstract state machine formalism to model the interface allowing scalable interactions with a service provider for specific discovery sessions. This approach allows the use of only the rules and data required, by the service requester at discovery-time (and thus limit data transmission or other costs) while at the same time it is possible to adapt the interface for various purposes of the late binding phase, i.e. discovery, selection, contracting/negotiation, etc. We also showed how, by extending WSMO service interface, the WSMO service choreography definition can be used to implement this interface. In a case scenario, we described the necessary semantic models and presented the algorithm (including creation of the knowledge base, processing the interface, and querying the knowledge base).

In our future work we plan to address the optimality for data fetching to decide on preferences for those interactions which might lead to results without processing all data fetching interface. In addition, we want to extend the data fetching interface to support other parts of the late binding phase. For example, negotiation building on data fetching might use interactions with specific meaning, such as for bidding etc. Layering of specific late-binding interfaces on the top of data fetching allows a modular approach to the definition of such interactions. We also plan to improve the implementation of the matching function for fully-fledged service discovery. In addition, we plan to incorporate run-time data mediation aspects into the discovery process where service requester and service providers use different ontologies.

References

1. Andreas Friesen and Stephan Grimm. DIP WP4 Service Usage, D4.8 Discovery Specification, available at http://dip.semanticweb.org/documents/D4.8Final.pdf. Technical report, 2005.
2. Thomas Haselwanter, Paavo Kotinurmi, Matthew Moran, Tomáš Vitvar, and Maciej Zaremba. Wsmx: A semantic service oriented middleware for b2b integration. In *ICSOC*, pages 477–483, 2006.
3. Manfred Hauswirth, Fabio Porto, and Le-Hung Vu. P2P and QoS-enabled service discovery specification available at http:/dip.semanticweb.org/documents/D4.17-Revised.pdf. Technical report, 2006.
4. Uwe Keller, Ruben Lara, Holger Lausen, Axel Polleres, Livia Predoiu, and Ioan Toma. WSMO D10.2 Sematic Web Service Discovery available at http://www.wsmo.org/TR/d10/v0.2/d10.pdf. Technical report, 2005.
5. J.B. Kim, A.Segev, A.Patankar, and M.G.Cho. Web services and bpel4ws for dynamic ebusiness negotiation processes,. In *International Conference on Web Services*, Las Vegas, Nevada, USA, 2003.
6. Jacek Kopecký, Dumitru Roman, Matthew Moran, and Dieter Fensel. Semantic web services grounding. In *AICT/ICIW*, page 127, 2006.
7. Ruben Lara and Daniel Olmedilla. Discovery and Contracting of Semantic Web Services. Technical report, 2005.
8. L.C. Lee. Progressive multi-agent negotiation. In *International Conference on Multi–Agent Systems*. MIT Press, 1995.
9. Adrian Mocan, Matthew Moran, Emilia Cimpian, and Michal Zaremba. Filling the gap - extending service oriented architectures with semantics. In *ICEBE*, pages 594–601. IEEE Computer Society, 2006.
10. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *1st International Semantic Web Conference (ISWC)*, page 333347, 2002.
11. Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubn Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter Fensel. Web Service Modeling Ontology. *Applied Ontologies*, 1(1):77 – 106, 2005.
12. Dumitru Roman, James Scicluna, Dieter Fensel, Axel Polleres, and Jos de Bruijn. D14v0.3. Ontology-based Choreography of WSMO Services, available from http://www.wsmo.org/TR/d14/v0.4/. Technical report, 2006.
13. A. Moormann Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.
14. D. Zeng and K. Sycara. Bayesian learning in negotiation. In *Working Notes for the AAAI Symposium on Adaptation, Co-evolution and Learning in Multiagent Systems*, pages 99 – 104, 1996.