Discovery and Mediation using DIANE Service Descriptions

Ulrich Küster[1], Birgitta König-Ries[1], Michael Klein[2]

[1]Institute of Computer Science, Friedrich-Schiller-Universität Jena,
07743 Jena, Germany, ukuester|koenig@informatik.uni-jena.de
[2]Institute for Program Structures and Data Organization, Universität Karlsruhe,
76131 Karlsruhe, Germany, kleinm@ipd.uni-karlsruhe.de

Abstract. In this paper, we introduce the DIANE Service Description (DSD) and show how it can be used to solve the mediation and discovery problems stated in the scenarios for the second SWS-Challenge workshop June 2006 in Budva, Montenegro.

1 Introduction

The Semantic Web Services Challenge 2006 [1] has presented a set of two problem scenarios to provide a common application to explore the trade-offs among existing approaches that facilitate the automation of mediation, choreography and discovery for services using semantic annotations. One scenario covers the mediation problem to make a legacy order management system interoperable with external systems that use a simplified version of the RosettaNetPIP3A4 specifications. The other scenario deals with the problem to dynamically discover and invoke the most appropriate shipment service provider for a set of given shipment requests.

In this paper we introduce our solution to both scenarios which is based on the Diane Service Description language DSD and the middleware built around it. In the following section we will introduce and explain DSD in general to lay the foundation to show how the discovery scenario has been solved using DSD in Section 3. In Section 4 we will explain how DSD was used to solve the mediation scenario and finally in Section 5 we will evaluate our approach and summarize.

2 What is DSD?

The goal of service-oriented computing is the ability to dynamically discover and invoke services at run-time, thus forming networks of loosely-coupled participants. The most important prerequisite is an appropriate semantic service description language – and with *DIANE Service Description* (DSD) [2] we try to provide such a language together with an efficient matching algorithm.

One main difference between DSD and other semantic service description languages is its own light-weight ontology language that is specialized for the

characteristics of services and can be processed efficiently at the same time. The basis for this ontology language is standard object orientation which is extended by four additional elements:

- Services perform world-altering operations (e.g., after invoking a shipment service, a package will be transported and a bill will be issued) which is captured by *operational elements*. We view this is the most central property of a service, thus, in DSD, services are primarily described by their effects – all other aspects (as flow of information, choreography etc.) are seen as secondary, derived properties. An effect is comprehended as the achievement of a new *state*, which in DSD is an instance from a state ontology.
- Services offer/request more than one effect (e.g. a shipment provider offers shipment to a multitude of possible locations and for various types and sizes of packages) which is captured by *aggregational elements*. Thus, the effect of a service is typically a *set of states*. In DSD, these are declaratively defined which leads to descriptions as acyclic directed graphs (see example in the next section).
- Services allow to choose among the offered effects (e.g. as a matter of course all shipment providers allow to input the package being transported and to select where to pick it up and where to ship it) which is captured by *selecting elements*. In DSD, selecting elements are represented as variables that can be integrated into set definitions, thus leading to configurable sets. Therefore, a service offer in DSD is represented by its effects as configurable sets of states.
- The appropriateness of services and their effects is varying for different requestors (e.g., in the scenario, a more expensive shipment provider will still be accepted, but a less expensive one will be preferred) which is captured by *valuing elements*. In DSD, these elements are represented by *fuzzy sets* capturing all preferences of the requestor – the larger the membership value the higher the preference.

For processing a semantic service description language, an efficient *matchmaking algorithm* is needed. Thus, for a given DSD offer description o and a given DSD request r, a matchmaker has to solve the following problem: What configuration of o's effect sets is necessary to get the best fitting subset of r's fuzzy effect sets. Our implementation answers this by stepping through the graphs of o and r synchronously in order to calculate the matching value in [0,1] as well as the optimal configuration of the variables. As the preferences are completely included in r, in contrast to existing approaches, our matcher does not need to apply any heuristics and thus is able to operate deterministically.

In order to interact with a service, DSD supports a simple choreography. During matchmaking several stateless *estimation steps* may be performed where operations of the service are called, which provide information (like the price of a package given its weight) but do not have real-world effects and do not imply any contract between the requestor and the provider. After the best match is found that service can be invoked by executing a single *execution step* which is supposed to produce the offered effects.

The proposed concepts are implemented in the DSD middleware. The architecture of the system is depicted in Figure 1. On the left hand side, the client is shown. It runs an application that at some point in time requests an external service to provide some functionality. The service request is formulated using DSD and sent to the middleware. There, the matcher module compares it to known service offers. When a matching result is found, it is configured appropriately and passed on to the execution module. This module then invokes the service using its WSDL grounding and finally returns the execution results to the client application.

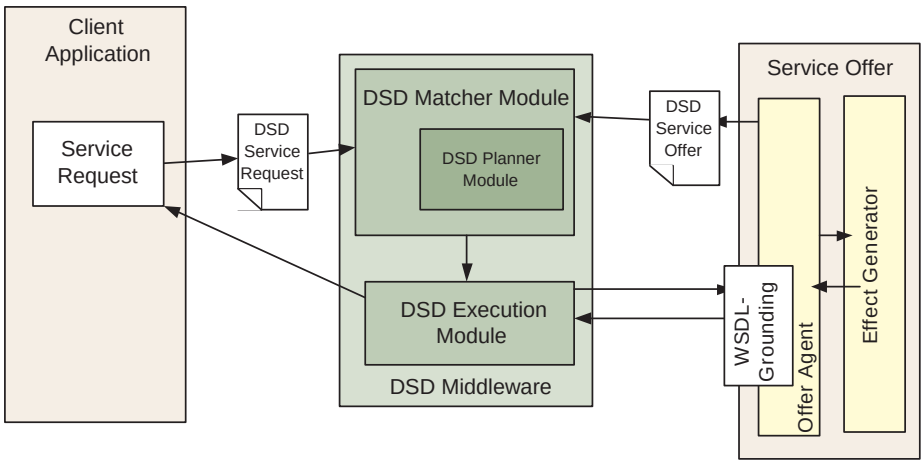

Fig. 1. DSD Middleware

3 Solving the discovery problem with DSD

The discovery problem posed by the SWS Challenge consisted of dynamically finding and invoking the most suitable shipment service for a number of shipment requests. In this section, we explain how offers and requests are described using DSD. These descriptions are then fed to the matchmaking algorithm introduced in Section 2 and finally, the best matching service is being invoked. Details on the invocation can also be found in this section.

3.1 Offer descriptions

Figure 2 shows the offer description of the Muller shipment service. The service collect a *cargo* at a certain *pickup* time and ships it from *fromAddress* to *toAddress* for a certain *price*. Thereby the following information has been encoded in the description:

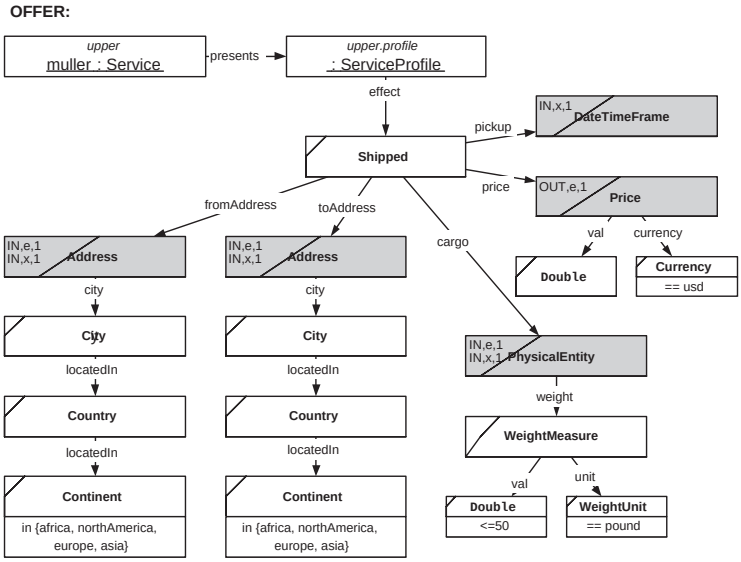

Fig. 2. Excerpt from the DSD description for the Muller shipment service.

- Both addresses are input variables for the first (and only) estimation step as well as the execution step (specified by the markers "IN, e, 1" and "IN, x, 1"). Valid addresses are addresses located in the enumerated continents (Africa, North America, Europe and Asia).
- The cargo is an input variable for the first estimation step as well as the execution step and its weight is restricted to not exceed 50 pounds.
- The pickup time may be provided as input for the execution step, but is not restricted. This differs from the given scenario. Unfortunately DSD currently neither supports to express concepts like *today* or *now* nor to formulate conditions over multiple input elements. Constraints like *"at most two days in advance"* (which would be equivalent to *"smaller than now plus two days"*) or *"pickup time at least 90 minutes"* (which would be equivalent to *"90 minutes smaller than end of pickup minus begin of pickup"*) cannot be expressed and had to be ignored.
- The shipping price of a package can be retrieved by calling the invokePrice operation of Muller's webservice. Thus the price is declared as an output variable of the first estimation step. The price will be given in US Dollars. The details how to execute the first estimation step, i.e. how to interact with Muller in order to retrieve the shipping price for a certain package can be found in the service offer's grounding. This has been omitted in Figure 2 but will be explained later.

The other service's offer descriptions have been encoded in a similar fashion without problems except for one. For the other service providers directions how to compute the price of a package depending on the delivery address and the

weight and dimension of the package had been given. DSD does not support to declaratively express such complex expressions directly. Thus, for each service provider a small webservice has been deployed that offers to compute the price of a package given the address and the package's size. These webservices are then used in the same way Muller's invokePrice operation is used.

3.2 Request descriptions

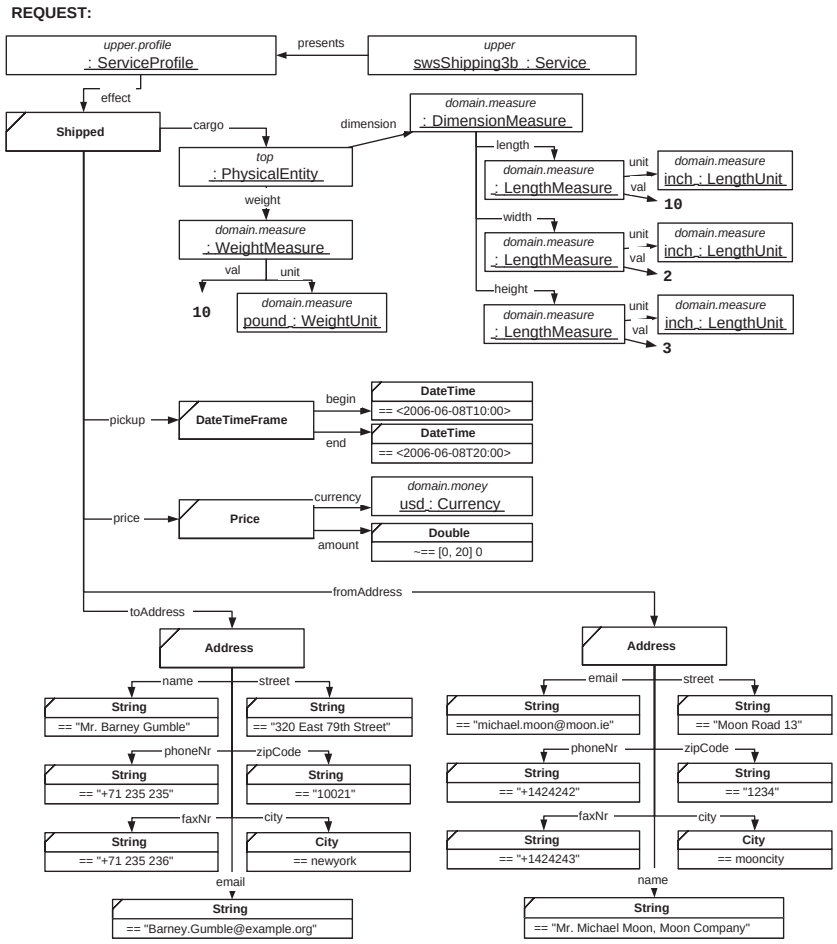

Fig. 3. Excerpt from the DSD request description of Goal 3b

Figure 3 shows the request corresponding to Goal 3b of the discovery scenario. The structure resembles the one of the Muller offer. Addresses, pickup time

and the cargo to be shipped are provided. Note that the city restrictions ("== newyork" and "== mooncity") in the addresses refer to ontological instances, because City – unlike street, email or zipCode which are of the primitive String type – is a complex entity type with publicly known instances stored in the ontology. Thus the state and country where the two cities are located in will be read from the ontology and do not have to be encoded in the request.

The amount of the price is restricted in a fuzzy manner by the expression "~==[0, 20] 0" which reads as: The requested price is 0 but any value from the interval [0, 20] will also be accepted with linearly decreasing preference. When matching Goal 3b against the available offers this will result in Muller, Walker, Weasel and Racer being rejected as their price exceeds 20 Dollars. Runner will be accepted with the smallest possible positive matching value as the price requested by Runner is exactly 20 Dollars.

The request descriptions corresponding to the other goals look pretty much like the one shown in figure 3. Only the addresses and properties of the cargo differ. Goals 1a through 2b do not specify any restrictions on the price, although an expression like "~==[0, 999999] 0" could have been used to express higher preference for less expensive shipment services.

Beside this, all requests could have specified some of the request elements, e.g. the price, as out variables, thereby denoting that – regardless of other restrictions on the price – the price of the package will be given as an output of the execution of the request. The matcher will assure that this constraint can be met by all matching offers, i.e. that the exact price will be known after service execution. The providers either would have to allow to input the price (which doesn't make a lot of sense in this case), have explicitly specified the price in their description or specified to provide the price as an out variable of the service execution or an estimation step themselves.

One more thing to mention is that not all capabilities of DSD had to be used to encode the given goals. If for instance competing preferences on price and pickup time are specified, DSD allows to specify a custom function (like a weighted sum) which will be used to combine the matching values of the elements pickup time and price.

3.3 Service invocations

After the best matching offer has been determined the corresponding service has to be invoked. The necessary information how to do this in an automated fashion is declared in the grounding part of an offer description. The grounding provides information like the endpoint to call and the SOAP action to perform, but also needs to specify how to create the proper message to be sent to the service implementation. In order to do this an empty template XML message has to be deployed together with the service description and mappings have to be specified in the grounding that define how to fill this template with the values from the properly configured offer description. These mappings recursively define which part of the XML template (selected by an XPath expression) will be filled with the attribute values of which element from the offer description.

```
mappingIN += anonymous XmlDsdMapping at upper.grounding [
       variable = $toAddress,
       dataNodePath = "OrderOperationRequest/to",
       attributeMappings += anonymous XmlDsdAttributeMapping at upper.grounding [
              attributePath = "name",
              subNodePath = "LastName"
       ],
       attributeMappings += anonymous XmlDsdAttributeMapping at upper.grounding [
              attributePath = "city/locatedIn",
              subNodePath = "Country",
              // name of united kingdom differs from ontology names
              converterClassName = "diane.converter.RacerCountryConverter",
              converterMethodName = "convert"
       ],
       ...
],
```

Fig. 4. Excerpt from the mapping definitions in the grounding of Racer's offer description

Standard marshalling can be used for primitive types but custom marshalling can be plugged in easily.

Figure 4 shows an excerpt from the mapping definitions of Racer's offer description. The XML element identified by the XPath expression "OrderOperationRequest/to" will be filled using data from the delivery address (variable "$toAddress"). The child node "LastName" will be filled taken the value from the "name" attribute of the delivery address. The "Country" child node will be filled using the "locatedIn" attribute of the "city" attribute of the delivery address. For most service offers the country was marshalled by simply using the value of the name attribute which provides English country names read from the ontology. This didn't work for the Racer service since Racer uses a different name for the United Kingdom ("United Kingdom(Great Britain)" instead of simply "United Kingdom"). Thus a custom marshaller had to be used for Racer to retrieve a string representation of country instances. This custom converter simply uses the name read from the ontology for all countries but the United Kingdom where the proper differing name is returned. It is plugged in into the mapping mechanism by simply specifying the class name of the converter ("RacerCountryConverter") and the name of the method to invoke ("convert").

Using these mappings the given XML template is properly filled and the correct message will be automatically send to the corresponding endpoint. The reply will be interpreted using similar mappings thereby making the results of the service invocations available to the middleware which will return it to the service requestor.

4 Solving the mediation problem with DSD

The architecture of our approach is displayed in Figure 4. The existing Moon systems on the right hand are wrapped into a DSD service. Unfortunately the simple stateless choreography supported by DSD (see section 2) is not directly compatible with the stateful complex choreography of the Moon systems. Thus

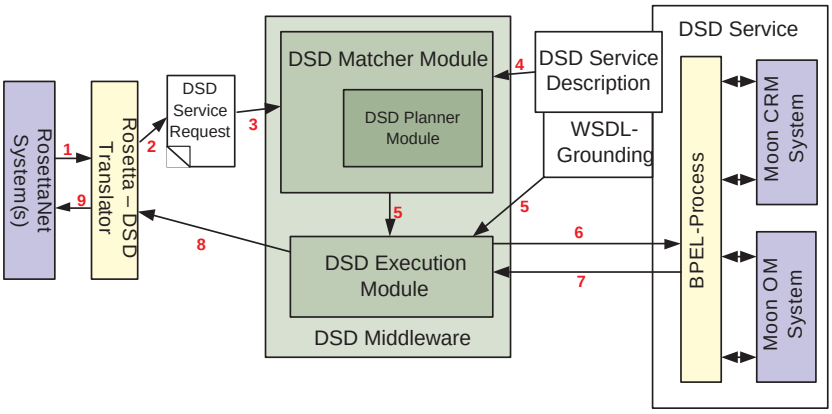

Fig. 5. Proposed mediator architecture

a BPEL process has been written that wraps the Moon System. That process handles the stateful complex interactions with the Moon systems while exposing a simple stateless interface as needed by DSD externally. This way Moon's system can be made available as a DSD service.

A Rosetta-DSD translator (left hand) has been created and made available as webservice. Upon reception of a RosettaNet purchase order message (1) a corresponding service request is created (2) and forwarded to the DSD Middleware (3). The Middleware collects available offers, in particular the one corresponding to the wrapped Moon systems (4) and sends the best matching offer together with its grounding to the DSD Execution Module (5). The DSD Execution Module then transforms the data accordingly and invokes the service. In this case a message to the BPEL process wrapped around the Moon systems is created and the BPEL process is invoked (6). The BPEL process handles the interaction with Moon's systems and returns the result to the DSD Execution Module (7) which forwards it back to the Rosetta-DSD translator. Finally that translator transforms the results into a purchase order confirmation message as defined by RosettaNet and replies to the originally calling service (9).

Finally we would like to outline the necessary changes in order to adapt our solution for the first version of the mediation scenario to the second one. In the second version the delivery address may be optionally specified on an item level instead of the global order level. Thus, items need to be grouped by address and one order per address should be created. This would have to be done in the RosettaDSD translator that would perform the grouping and issue a single DSD request for every order to be created. Furthermore if an item is rejected by Moon's order management system, in version two this item may be scheduled for production at Moon's new production management system. This change further increases the complexity of the choreography used to interact with Moon's systems. Since DSD currently does not support such complex stateful choreographies, these changes would have to be incorporated into the BPEL

process that has already been used in the first version of the scenario to wrap Moon's systems to expose a simplified interface. Both changes have not been implemented yet but are pretty straightforward.

5 Evaluation and Summary

In this paper, we have described how DSD and the DIANE middleware can be used for service discovery and to enable cooperation between existing systems. While the DSD approach is very well suited for the discovery scenario, the solution proposed for the mediation problem is admittedly not as powerful as one would like it to be. In particular the need to manually create a BPEL process in order to deal with Moon's order management system's complex choreography is unsatisfying. What is needed here is to enable DSD to deal with such choreographies in a better way. We are currently working on that.

In other aspects the solution is more powerful (and thus more complex) than actually needed for this simple scenario. By using DSD as mediation technology Moon and RosettaNet systems do not only become interoperable with each other but with any DSD based system. This advantage of using semantic technology to mediate between the systems does not really become visible, until the scenario becomes more complex and involves more possible partners.

References

1. DERI Stanford: Semantic web service challenge 2006 - challenge on automating web services mediation, choreography and discovery. http://sws-challenge.org/wiki/index.php/Main_Page (2006)
2. Klein, M., König-Ries, B., Müssig, M.: What is needed for semantic service descriptions - a proposal for suitable language constructs. International Journal on Web and Grid SErvices (IJWGS) **1** (2005) 328–364