Discovery and Mediation using DIANE Service Descriptions

Ulrich Küster and Birgitta König-Ries

Institute of Computer Science
Friedrich-Schiller-Universität Jena
07743 Jena, Germany
`ukuester|koenig@informatik.uni-jena.de`

Abstract. In this paper, we introduce the DIANE Service Description (DSD) and show how it has been used to solve the mediation and discovery problems stated in the scenarios of the SWS-Challenge 2006[1]. We explain the solution built for the second SWS-Challenge workshop in Budva, Montenegro (June 2006) and show how it has been extended for the current third SWS-Challenge workshop in Athens, GA, USA.

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 RosettaNet PIP3A4 specifications[2]. 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. This solution was originally presented in [2]. We will recapitulate the central aspects of that solution and introduce the enhancements achieved meanwhile. 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. Finally, in Section 5 we will evaluate our approach, outline the directions of future work 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 par-

[1] www.sws-challenge.org
[2] http://www.rosettanet.org/PIP3A4

ticipants. The most important prerequisite is an appropriate semantic service description language – and with *DIANE Service Description* (DSD) [3] we 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 requesters (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 requester – 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 requester 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 overall 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 (e.g. by filling a predefined semantic request template) and sent to the middleware. There, the matcher module compares it to the available 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 grounding and finally returns the execution results to the client application. More detailed information how to integrate semantic service requests into existing processes using the DSD Middleware can be found in [4].

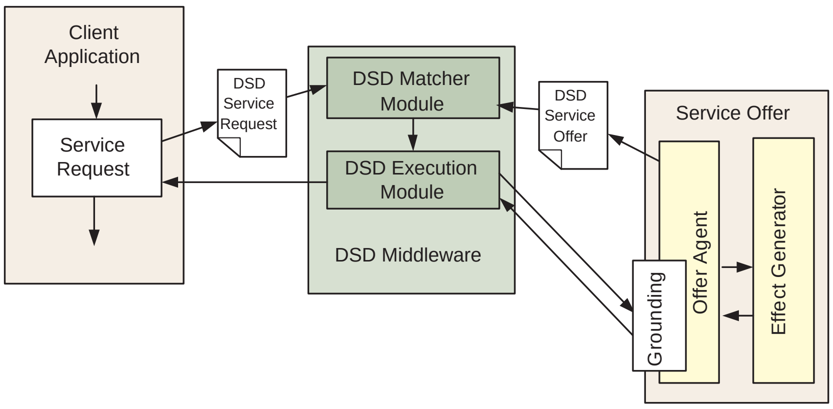

Fig. 1. DSD Middleware

3 Solving the discovery problem with DSD

The discovery problem posed by the SWS Challenge consists 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.

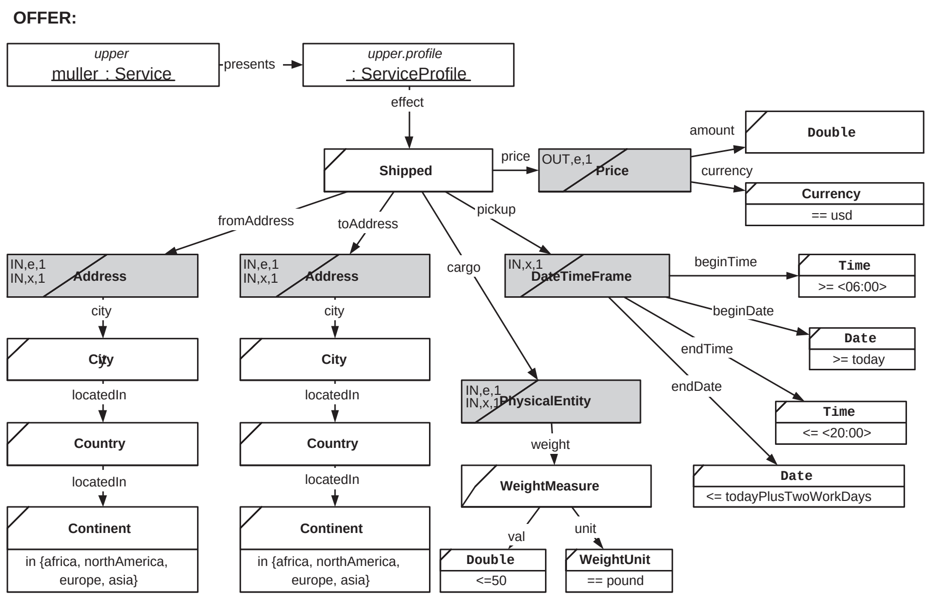

Fig. 2. Excerpt from the DSD description for the Muller shipment service.

3.1 Offer descriptions

Figure 2 shows the offer description of the Muller shipment service. The service collects 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:

- 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 frame may be provided as input for the execution step. As stated in Muller's description, pickup time must be between 6AM and 8PM. Pickup may be requested immediately, but at most two working days in advance. Our original solution did not contain these restrictions since DSD was lacking the notion of *now* or *today*. Special instances have been added to DSD to capture these semantics. Unfortunately - as of now - DSD is not able to perform arithmetic computations on values specified in restrictions like those above. Thus a special date instance *todayPlusTwoWorkDays* has been created to capture Muller's requirement, that collection may be ordered a maximum of two working days in advance. It would be more elegant to express such a restriction for instance as "`<= today + <P2D>`". DSD is currently being extended to cover such arithmetic expressions. Such arithmetic

expressions are also needed to express that there must be at least an interval of 90 minutes for collection. Currently this requirement cannot be expressed with DSD properly.

– 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

Figure 3 shows the request corresponding to Goal 4a of the discovery scenario. The structure resembles the one of the Muller offer. Addresses, 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 pickup time has been restricted although this was not required by the goal description. The price of the package is not restricted, in fact the corresponding property is not even modelled. This is due to the fact that Goal 4a doesn't specify such a restriction, please refer to [2] for an example where the price is restricted in a fuzzy manner that encodes a maximum price as well as preference for lower prices.

Goal 4a differs from the other discovery goals in that this goal asks for shipment of two packages and thus enforces two invocations of the corresponding shipping provider service. In our previous solution this goal could not be expressed with DSD. However, DSD's effect sets provide a natural mechanism to deal with such requests. The standard semantic of a DSD request effect set is that one (the best) effect out of the specified effect set should be provided. *Iteration directives* may be used to change this semantic. In Figure 3 the iteration directive `"<Best 2 1>"` within the PhysicalEntity set describing the cargo to be shipped encodes that the best two effects described by this set should be provided (the two corresponds to the first parameter in the directive `"<Best 2 1>"`). The matcher thus binds the corresponding variable in the offer description

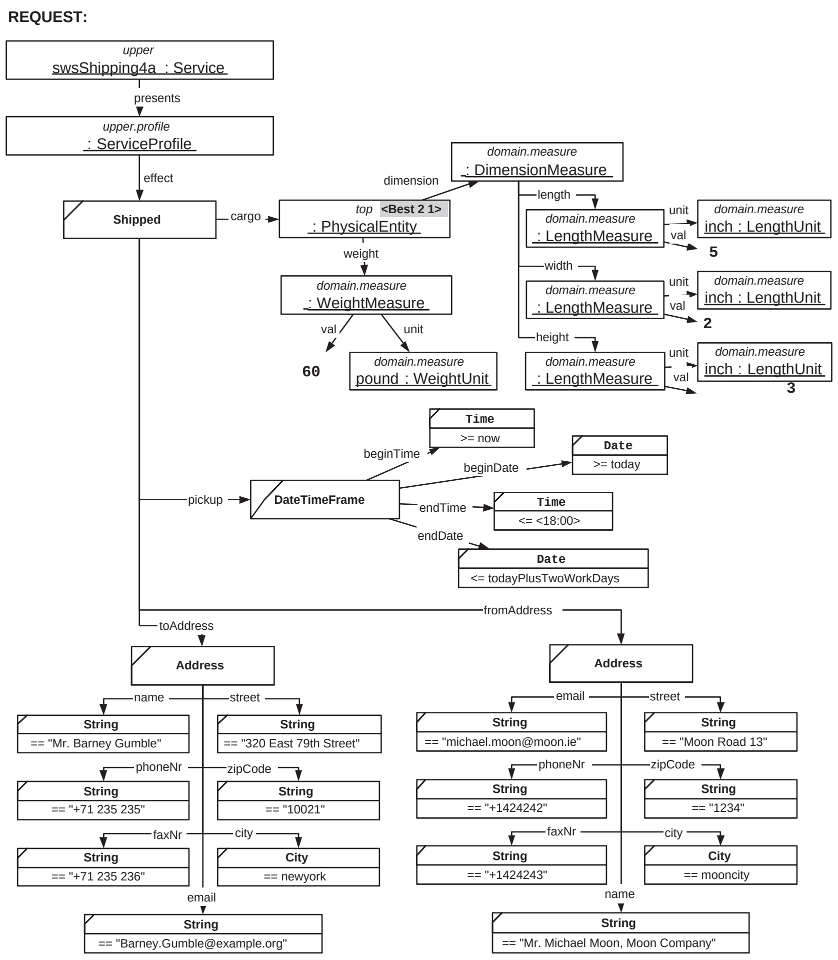

Fig. 3. Excerpt from the DSD request description of Goal 4a

with a set of two corresponding instances instead of a single value. For automated service invocation such a binding will either result in two invocations of the picked shipping offer (one for each package) or a single invocation with two package specifications sent to the service - depending on the interface of the service described in its grounding. Iteration directives had not been available when the original solution to the discovery scenario has been finished for the Second SWS-Challenge workshop in Budva, Montenegro. The current modelling of Goal 4a is an enhancement achieved since. Allthough Goal 4a could be successfully modelled, as of now the work on iteration directives is an ongoing effort. Thus not all possible cases of usage are currently supported by our implementation and a full discussion of the complete semantics is beyond the scope of this paper. This is clearly a direction of future work.

```
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

The descriptions of the other discovery goals look very similar to Goal 4a as shown in Figure 3. However, one more thing to mention is that not nearly 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 and allows for very fine-grained user preference based matching.

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. 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

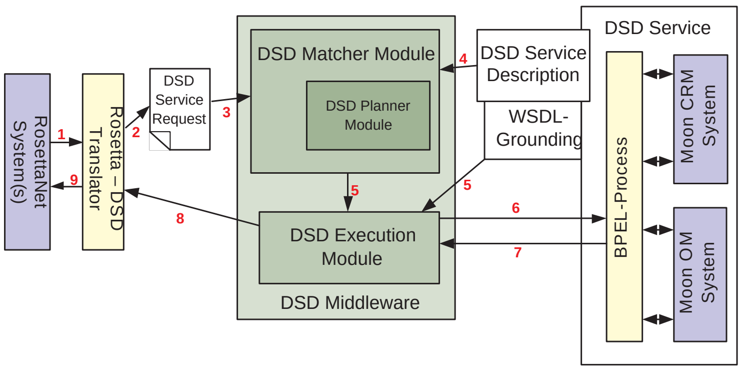

Fig. 5. Proposed mediator architecture

The architecture of our mediation 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 a BPEL process has been written to handle the stateful complex interactions with the Moon systems and expose 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 matches them with the request. During the matching process the offers will be configured properly (e.g. the id of the article to be purchased will be set). If no matching service can be found and automated composition or additional knowledge gathering is necessary the DSD Planner Module attempts to solve these issues. After having determined the best matching offer this offer and its grounding is send 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).

There were two main changes between the first and the second version of the mediation scenario, changes of the data format of RosettaNet messages and changes in the choreography of Moon's systems. In the scenario at hand RosettaNet messages contain a purchase order that contains several items to purchase. In the first scenario the ship-to address had to be specified on the purchase order level, in the second scenario it could optionally also be specified on the item level. Since Moon's system are not able to deal with ship-to addresses on the item level it was intended to split such purchase orders into several orders, each containing the items that were to be shipped to the same address. This process is handled by the Rosetta-DSD translator, thus no change to the rest of the solution was necessary. On the side of Moon's systems a new Production Scheduling System was introduced. If a particular item is rejected by Moon's Stock Management System (which corresponds to the previous Order Management System), the production price and schedule may be obtained from the Production Scheduling System. If the given data meets the customers expections regarding price and shipping time, the item has to be scheduled for production at Moon's system. This change is purely within the choreography of Moon's system, from the point of the RosettaNet customer it doesn't make much of a difference whether an item is scheduled for production or already stocked, as long as price and shipping time constraints are met. Thus the change is completely handled by the BPEL wrapper used to expose a simple interface of Moon's systems to the DIANE middleware. This wrapper process has been adapted correspondingly and redeployed. No change to the middleware and not even to the DSD description of Moon's offer has been necessary. Both adaptations were straightforward and easy to implement on top of the previous solution.

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.

We have outlined our original solution to both scenarios as presented at the last SWS-Challenge workshop in Budva, Montenegro. We described the necessary changes to our solution in order to solve the Mediation Scenario 2 on top of our solution to Mediation Scenario 1. We furthermore introduced some enhancements to the original discovery solution. These are an improved (yet still incomplete) dealing with temporal semantics and a solution of Discovery Goal 4a by defining and implementing extensions to the original DSD semantics.

There are two main directions of future work. First, the dealing with temporal semantics is still unsatisfying. In particular it would be nice to be able to express constraints with arithmetic computations on time instances (end of pickup time frame minus begin of pickup time frame needs to be larger than a certain amount of time). Beside that, DSD is currently not able to compare time values specified in differing time zones. Both issues are objective of ongoing efforts. Second, although sufficient for the discovery scenario at hand, the iteration directives used to request multiple effects at once (e.g. shipping of multiple packages) need to be finalized and their implementation within the DSD Middleware as proof of concept needs to be completed. This, also, is ongoing work.

Overall, the DSD approach has proven to be very well suited for the discovery scenario. The solution proposed for the mediation problem is probably not as powerful as one would like it to be since most of the process mediation is done by a manually coded BPEL process. However, 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. Petrie, C.: It's the programming, stupid. IEEE Internet Computing **10** (2006) 96, 95
2. Küster, U., König-Ries, B., Klein, M.: Discovery and mediation using DIANE service descriptions. In: Second Workshop of the Semantic Web Service Challenge 2006 - Challenge on Automating Web Services Mediation, Choreography and Discovery, Budva, Montenegro (2006)
3. 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
4. Küster, U., König-Ries, B.: Dynamic binding for BPEL processes - a lightweight approach fo integrate semantics into web services. In: Second International Workshop on Engineering Service-Oriented Applications: Design and Composition (WESOA06) at 4th International Conference on Service Oriented Computing (ICSOC06), Chicago, Illinois, USA (2006)