Semantic Mediation between Business Partners - A SWS-Challenge Solution using DIANE Service Descriptions

Ulrich Küster and Birgitta König-Ries
Institute of Computer Science, Friedrich-Schiller-University 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 problem stated in the scenarios of the SWS-Challenge[1]*. We provide a consolidated description of our approach that we presented at the first three SWS-Challenge workshops and briefly discuss its strengths and drawbacks.*

1 Introduction

The Semantic Web Services Challenge [7] 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 of 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[2] specifications. The other scenarios deal with semantic discovery of services.

In this paper we present a consolidated description of our solution to the mediation scenario which is based on the DIANE Service Description language DSD and the DIANE Middleware built around it. In the following section we will introduce and explain DSD in general to lay the foundation to show how it has been used to solve the mediation scenario in Section 3, in Section 4 we will briefly cover related work 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) [3, 5] 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.

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

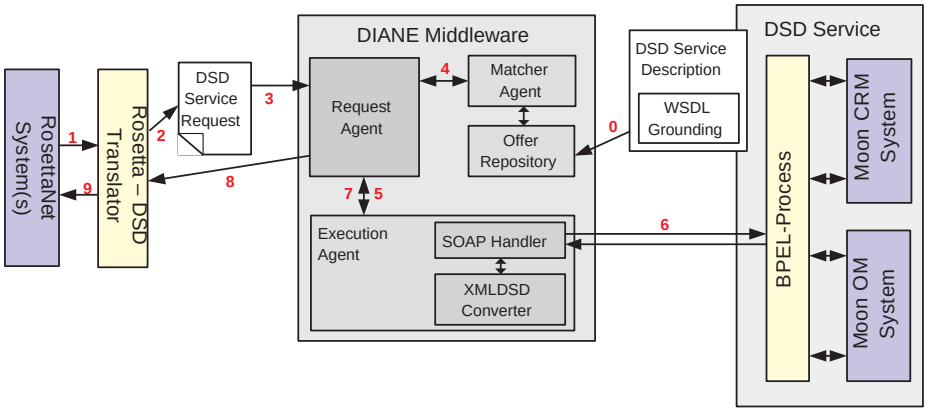

Figure 1. Architecture of the solution to the first mediation scenario

- 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 using *fuzzy sets* instead of crisp ones in service request descriptions thereby 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 answer the following question: How well are o's crisp configurable effect sets contained in r's fuzzy effect sets and which configuration of o's effect sets will maximize this value? Our implementation solves 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 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* that produces the offered effects.

3 Solving the mediation problem with DSD

In this section we will describe how we solved the mediation problem using DSD and the DIANE Middleware. We will first present our solution to the original version of the scenario and then describe the changes that were necessary to adapt that solution to the second version of the scenario.

3.1 Solution to the original scenario

The architecture of our approach is displayed in Figure 1. The main idea of our approach is to view the mediation problem as a discovery problem. Moon needs to advertise its functionality as a service offer described in DSD. Meanwhile the RosettaNet system needs to express the required functionality (purchasing of products) as a DSD service request. The DIANE Middleware will then discover the Moon service as a suitable offer for the purchasing request and automatically handle all invocation details. In the following we will describe this approach in detail.

Exposing Moon as a DSD service offer. As mentioned, the existing Moon systems on the right hand need to be described as a DSD service offer. Unfortunately the simple choreography supported by DIANE (see Section 2) is not directly compatible with the stateful complex choreography of the Moon systems. DIANE was designed for automated comsumption of functionality offered as a service. Services designed for this end can usually be consumed by a single invocation whereas services that require a complex back and forth of messages are typically designed for human interfaces. To overcome the incompatibility between the simple choreography supported by DIANE and the complex one offered by Moon we manually created a BPEL process that wraps the Moon systems. This BPEL process offers the single operation required by DIANE to consume Moon's functionality and handles the necessary process mediation internally. Thanks to the good tool support available for BPEL the creation of this process was easy and straightforward. A DSD service description covering Moon's offer has been written and published to the DIANE Middleware (Figure 1 - 0). Figure 2 shows excerpts from that offer descrip-

tion (some attributes and the grounding information have been omitted). For the following an intuitive understanding of the offer is sufficient. Moon presents a profile with a single effect set that describes the state of the world that will be reached by a service invocation. In that state something will be owned by the service requester. Moon allows the requester to input an `ItemSet` concept specifying order information and the list of items to be purchased (marker "`IN,x,1`" on the ItemSet concept). After the invocation a corresponding ItemSet with updated information (containing price, order status etc.) will be returned to the requester (marker "`OUT,x,1`" on the ItemSet concept).

Translating from RosettaNet to DSD In order to treat a RosettaNet purchase order as a DSD service request a mediator needs to convert the incoming RosettaNet XML message into an ontological DSD request. Every relevant element of the quite complex RosettaNet messages has to be translated to the corresponding DSD concept. Unfortunately no graphical editor tool exists to define mappings between XML and DSD in a convenient way. Because of this lacking tool support and the complexity of the RosettaNet messages we decided to manually code a translator that performs the necessary conversion instead of specifying declerative mapping rules. Our translator has been written in Java and exposed as a web service. When the endpoint receives a purchase order (Figure 1 - 1) it sends the required acknowledgement, parses the purchase order and constructs a corresponding DSD purchasing request (Figure 1 - 2). This request is then sent to the DIANE Middleware (Figure 1 - 3) that will take care of all invocation details. After a successful invocation of the Moon offer, the middleware will return the results of the invocation to the requester – in this case the Rosetta-DSD Translator (Figure 1 - 8). The translator then uses the returned DSD instance data to construct a proper purchase order confirmation message to be sent to the RosettaNet System (Figure 1 - 9).

Service invocation by the DIANE Middleware The Request Agent of the middleware processes any service requests sent to the DIANE Middleware on behalf of the requester in a fully automated fashion. When it receive a DSD request (Figure 1 - 3) it will call the Matcher Agent (Figure 1 - 4) to receive a list of service offers matching the request ordered by decreasing degree of match. The Matcher Agent will call a plugable offer repository to receive the DSD descriptions of the available service offers, in particular the one corresponding to the wrapped Moon systems. It will then match those offers with the request. During matchmaking it will not only determine how well a particular offer matches the request, but where necessary also configure that offer in an optimized way with regard to the request. That means it will determine values for all input variables of

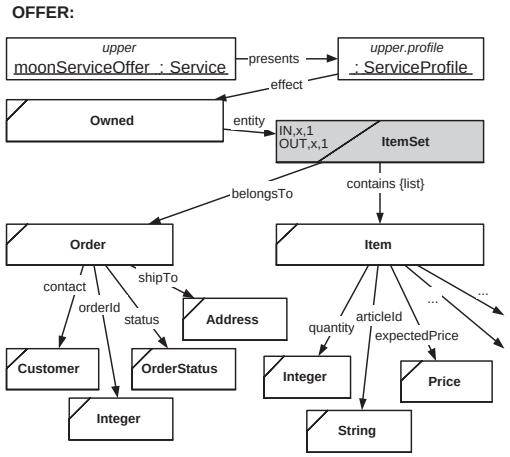

Figure 2. Excerpts from Moon's service offer

the offer description and ensure that all outputs required by the requester will be provided. Details on how the matchmaking is performed are published in [6, 5] and can also be found in our solution to the SWS-Challenge discovery problems[3].

The Request Agent receives the list of readily configured service offers from the Matcher Agent, picks the best matching offer and forwards it to the Execution Agent (Figure 1 - 5). The Execution Agent exploits the information provided in the grounding of that offer to actually perform the service invocation. Figure 3 shows excerpts from the grounding specification of the Moon service. A single SOAP operation is defined which specifies the endpoint to call (`endpoint`), a path to an xml message template (`xmlTemplatePath`) as well as mappings to convert between ontological DSD data and XML messages and vice versa (`mappingIN` and `mappingOUT`). DIANE follows a pragmatic approach for that purpose that was introduced in [4]. For each SOAP operation the already mentioned `xmlTemplatePath` property points to an empty XML message template that has to be deployed together with the service description at the DIANE middleware. Mappings have to be specified in the grounding that define how to fill the template with the values from the properly configured offer description.

`mappingIN` definitions are used to create the inputs of an operation, thus lowering from DSD data to XML. The shown example specifies the variable from the offer's description to use (`variable`) and an XPath expression that identifies the XML node in the message template to fill with data from that variable (`dataNodePath`). Nested attribute mappings are specified to serialize attributes into subnodes – both in the context of the variable and node used

[3] A paper about our discovery solution will be presented at the same workshop like this paper

```
supports = anonymous SOAPServiceGrounding [
soapOperations += anonymous SOAPExecuteOperation [
    xmlTemplatePath = "  moonXmlRequestTemplate.xml   ",
    endpoint = "  http://hnsp.inf-bb.uni-jena.de:8080/active-bpel/
                    services/MoonMediatorBPELLinkTypeService“    ,
    mappingIN += anonymous XmlDsdMapping [
        variable = $itemSet,
        dataNodePath = "DSDOrderRequest",
        attributeMappings += anonymous XmlDsdAttributeMapping [
            attributePath = "/contains",
            subNodePath = "Item",
        attributeMappings += anonymous XmlDsdAttributeMapping [
            attributePath = "/articleId",
            subNodePath = "articleId"
        ],
        attributeMappings += anonymous XmlDsdAttributeMapping [
            attributePath = "/quantity",
            subNodePath = "quantity"
        ],
        attributeMappings += anonymous XmlDsdAttributeMapping [
            attributePath = "/buyerItemNumber",
            subNodePath = "lineNumber"
        ],
        ...
    ],

    mappingOUT += anonymous XmlDsdMapping [
        variable = $itemSet,
        dataNodePath = "DSDOrderResponse",
        attributeMappings += anonymous XmlDsdAttributeMapping [
            attributePath = "/belongsTo/status",
            subNodePath = "GlobalPurchaseOrderStatusCode",
            converterClassName = "swschallenge.UnMarshaller",
            converterMethodName = "getOrderStatusEntity"
        ],
        attributeMappings += anonymous XmlDsdAttributeMapping [
            attributePath = "/belongsTo/orderId",
            subNodePath = "OrderID"
        ],
        attributeMappings += anonymous XmlDsdAttributeMapping [
            attributePath = "/contains",
            subNodePath = "Item",
            attributeMappings += anonymous XmlDsdAttributeMapping [
                attributePath = "articleId",
                subNodePath = "itemID"
            ],
            ...
```

Figure 3. Excerpt from the grounding of Moon's offer description

by the parent mapping. For basic DSD types (basically the types corresponding to the basic XML schema datatypes) standard serialization will be used by default but custom serialization can be plugged in if necessary. Once the given XML template is properly filled, the correct message will be automatically sent to the corresponding endpoint, in this case the BPEL process that wraps the Moon systems (Figure 1 - 6).

The response needs to be interpreted to make the results of the service invocations available to the middleware. This is accomplished by `mappingOUT` definitions that are used to process the outputs of an operation and lift the XML data returned by a SOAP invocation to ontological DSD instances. They work in the same way as the `mappingIN` definitions. In Figure 3 custom deserialization is used to deserialize an *order status* string into an ontological DSD instance. The corresponding mapping simply specifies the name of a Java class (`converterClassName`), which has to be deployed at the middleware. The middleware will then load that class and use the specified method (`converterMethodName`) to deserialize the given value (`subNodePath`) into the correct DSD attribute (`attributePath`). The results of the invocation are then returned to the Request Agent (Figure 1 - 7) which sends them back to the client (the Rosetta-DSD Translator) in response to the DSD service request (Figure 1 - 8).

3.2 Adapting to the scenario change

One of the goals of the SWS-Challenge is to examine how adaptable systems are to change by measuring how much effort is involved to adapt a particular solution to a change in the scenario. Thus, after the first solution described above was submitted, a changed second mediation scenario was released by the SWS-Challenge organizers. In this section we describe the changes that were made to adapt our solution to the changed scenario.

There were two main changes between the first and that 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 SWS-Challenge mediation scenario, 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 systems 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. Since in our view orders to different addresses are most properly modeled as different service requests, we decided to have this process being 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 customer's expections regarding price and shipping time, the item has to be scheduled for production at Moon's system. This change was 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, in our approach the change should be hidden inside of Moon's systems. Therefore we adapted our BPEL wrapper correspondingly and redeployed it. 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.

4 Related Work

A comprehensive coverage of the related work on (semantic) system mediation between heterogeneous systems would go far beyond the scope of this paper. Therefore we restrict ourselves to a brief coverage of the other established solutions to the SWS-Challenge mediation scenario.

The joint team from Politecnico Milano / CEFRIEL presented an approach [1] that is based on sophisticated software engineering methods and tools rather than semantic technology. The solution is therefore far less complex than ours, but on the other hand – since it is a directly mediated approach (see next section) – we expect it to be also less powerful in scenarios which are more complex than the current one.

The solution by DERI [2] is the one most similar to ours. Their team uses WSMX (Semantic Web Service Execution Environment) as a semantic middleware similar to our use of the DIANE Middleware. They also use a specifically created adapter to connect the RosettaNet system to WSMX and create a semantic service request from the purchase order (like our RosettaNet-DSD Translator). WSMX will then discover and invoke a matching service offer as the DIANE Middleware does. However, WSMX is able to handle complex process mediation by means of abstract state machines. Thus their solution does not require a service to be consumable by a single invocation as DIANE does. In this aspect their solution is more powerful than ours, but also more complex.

5 Evaluation and Summary

In this paper, we have described how DSD and the DIANE middleware can be used to enable cooperation between existing systems. We have described our original solution to the first version of the mediation scenario as well as the changes we had to make to adapt that solution to the second version of the mediation scenario.

The heart of our approach was to view the mediation scenario as a discovery problem. This way we could leverage our discovery-centric DIANE framework for our solution. We admit, that at a first glance on the one hand the solution looks more complex than actually needed to mediate between two systems. On the other hand, it still seems less powerful than one would like it to be since most of the process mediation is done by a manually coded BPEL process. Both result from the fact that our approach is a strongly decoupled one. We do not mediate directly between RosettaNet and Moon, but instead mediate between RosettaNet and DSD and then between DSD and Moon. For the necessary data mediation, this results in increased complexity since it requires an additional translation. For the necessary process mediation, we decided to limit the complexity we are willing to support by requiring any service offer to be consumable via a simple interface that does not require a complex message exchange. Thus we had to move some process mediation functionality into the Rosetta-DSD translator (sending of acknowledgement messages) and a lot of it into the BPEL wrapper that hides Moon's complex choreographies.

However, there is a strong benefit of our decoupled approach. By using DSD as mediation technology, Moon and RosettaNet systems do not only become interoperable with each other but with any DSD based system. No manual adaptations will be needed, neither to service the RosettaNet purchase order with any service offer that is described in DSD nor to use Moon as a provider for any DSD-based purchasing request. This advantage of using semantic technology to mediate between the systems, however, does not really become visible until the scenario becomes more complex and involves many more possible partners, i.e. until combined discovery and mediation scenarios are regarded.

References

[1] M. Brambilla, S. Ceri, D. Cerizza, E. D. Valle, F. Facca, and C. Tziviskou. Coping with requirements changes: SWS-Challenge phase II. In *Second Workshop of the Semantic Web Service Challenge 2006*, Budva, Montenegro, June 2006.

[2] T. Haselwanter, P. Kotinurmi, M. Moran, T. Vitvar, and M. Zaremba. Dynamic B2B integration on the semantic web services: SWS Challenge phase 2. In *Second Workshop of the Semantic Web Service Challenge 2006*, Budva, Montenegro, June 2006.

[3] M. Klein, B. König-Ries, and M. Müssig. What is needed for semantic service descriptions - a proposal for suitable language constructs. *International Journal on Web and Grid Services (IJWGS)*, 1(3/4), 2005.

[4] U. Küster and B. König-Ries. 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 ICSOC06*.

[5] U. Küster, B. König-Ries, M. Klein, and M. Stern. DIANE - a matchmaking-centered framework for automated service discovery, composition, binding and invocation on the web. *To appear in: International Journal of Electronic Commerce (IJEC) - Special Issue on Semantic Matchmaking and Retrieval*, 12(2), 2007.

[6] U. Küster, B. König-Ries, M. Klein, and M. Stern. DIANE - an integrated approach to automated service discovery, matchmaking and composition. In *Proceedings of the 16th International World Wide Web Conference (WWW2007)*, Banff, Alberta, Canada, May 2007.

[7] C. Petrie. It's the programming, stupid. *IEEE Internet Computing*, 10(3), 2006.