Semantic Service Discovery with 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`

Summary. In this chapter, we introduce the DIANE Service Description (DSD) and show how it has been used to solve the discovery problems stated in the scenarios of the SWS-Challenge. We explain our solution as of the fifth SWS-Challenge workshop in Stanford, CA, USA (November 2007) and provide a discussion about its strengths but also shortcomings.

1 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) [KKRM05, KKRKS07a] we provide such a language together with an efficient matchmaking algorithm.

One main difference between DSD and other semantic service description languages is its own lightweight 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.

- Service providers offer 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. On the other hand requesters typically accept different services with different properties, e.g. a fast and expensive shipping or an inexpensive but slower one. Both is captured in DSD by *aggregational elements*. Thus, the effect of a service (request or offer) is typically a *set of states*. For offers, these are the states the service can potentially create, for requests these are the states the requester is interested in. In DSD, sets are declaratively defined which leads to descriptions as trees (see examples 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 different service offers and their effects is varying for a given requester (e.g., in the first 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 request descriptions. Set based descriptions allow expressing that quite different services are acceptable for a requester. Using fuzzy instead of crisp sets in these descriptions additionally allows to include all preferences of the requester in a request description – the larger the fuzzy membership value of a service in the described service set, the higher the preference of the requester for that particular service.

For processing a semantic service description language, an efficient *matchmaking algorithm* is needed. 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 crisp effect sets is necessary to get the best fitting subset of r's fuzzy effect sets. Or – in other words – how well is o's offer contained in what r requests and how should o be configured to maximize this value? 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 assumes a simple choreography. During matchmaking several web safe *estimation operations* 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 a contract between the provider and the client (in this case the matchmaking agent). After the best match is found that service can be invoked by executing a single *execution operation* which is supposed to produce the offered effects.

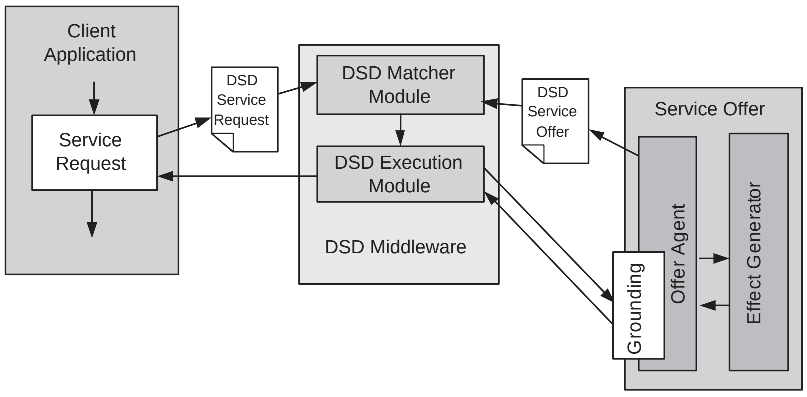

Fig. 1. DIANE Middleware architecture

The proposed concepts are implemented in the DIANE 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 DIANE Middleware can be found in [KKR06b].

2 Solving the SWS-Challenge discovery problems with DSD and the DIANE framework

The SWS Challenge poses two set of discovery problems, one related to finding a shipping provider for a given shipping request, the other one related to purchasing of IT hardware. At the previous SWS-Challenge workshops we have presented the most complete solution to those problems [KKRK06, KKR06a, KKR07b, KKR07a]. The complete solution including all offer and request descriptions, all additional files, an executable version of the DIANE Middleware and a technical description how to get the solution running can be found on the SWS-Challenge wiki[1].

[1] http://sws-challenge.org/wiki/index.php/Solution_Jena

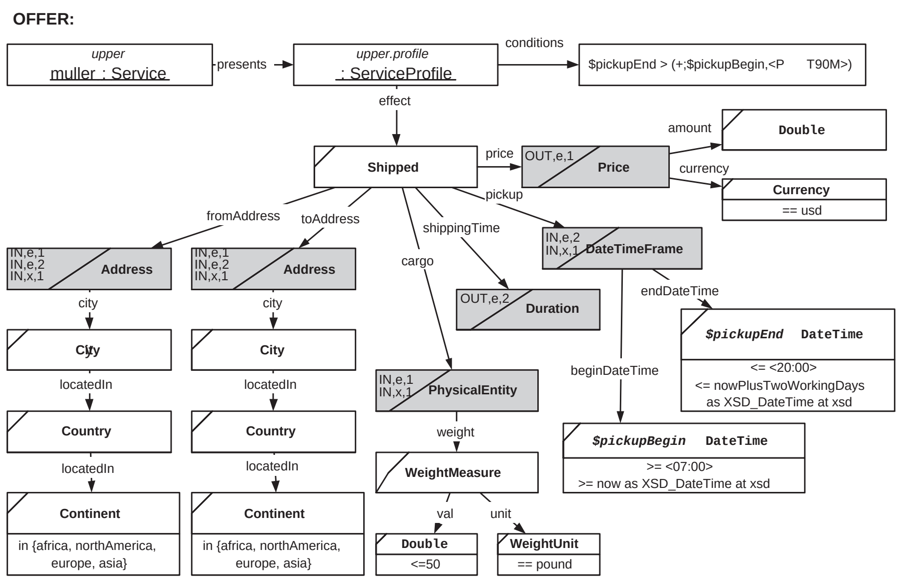

Fig. 2. Excerpt from the DSD description for the Muller shipment service.

In this section, we provide an in-depth description of our solution. We describe how offers and requests for the first and the second scenario have been described using DSD and particularly elaborate on the difficulties we encountered and how we solved them. At the end of the section we explain how services are invoked automatically by the DIANE middleware, either to gather additional information during the matchmaking or to invoke the best matching service found. We conclude this chapter with a discussion of the strengths and shortcomings of our approach.

2.1 Offer descriptions for the first scenario

Figure 2 shows the offer description of the Muller shipment service as an example for all shipping services used in the first discovery scenario. We use *GDSD*, an informal UML like graphical notation of DSD for our illustrations because it is more compact and easier to understand than the more formal *FDSD* notation. By the example of the Muller service we detail in the following how the various aspects of the shipper's textual description from the scenario have been captured in our DSD descriptions of the services.

Figure 2 shows a service instance presenting a profile (the grounding has been ommitted) that offers a single effect set (diagonal lines in the upper left corner of a concept denote a DSD set). The set of `Shipped` states that can be created by the service are characterized by the *property conditions* of that set: The service collects a `cargo` at a certain `pickup` time and ships it from fromAddress to toAddress for a certain price within the stated shippingTime.

Offer input and outputs

Inputs (the configurability of an offer) and outputs of a service are described as variables and directly integrated into the description. In GDSD variables are denoted by grayed rectangles. Different types of variables are denoted by markers on the left side of a variable. The same concept may serve as variable for different purposes, thus multiple markers are allowed. The marker IN,e,1 on the fromAddress set in Muller's description for instance denotes that the value of this set needs to be given as input for the first estimation operation. More precisely IN declares the variable as input, while OUT would have specified an output, e is used to distinguish between web-safe estimation operations (e) and the final execution of the service (x) and the following number (1) is used to distinguish between different operations (more on this below).

Restrictions on package size and weight

Muller requires packages to weigh less than fifty pounds. This condition has to be captured by the cargo set in Muller's description using an appropriate *property condition.* As you can see in Figure 2 the cargo set's weight property points to a set of WeightMeasures whose value property in turn points to a set of Double values. By the *direct condition* "<= 50" this set of Double values is restricted to contain only values that are equal or smaller than fifty. Restrictions on the maximum length or width of a package could be added in a similar fashion by adding additional property conditions to the cargo set.

Restrictions on the operation range of the service

Muller operates only in Africa, Asia, Europe and North America. Thus the fromAddress and toAddress sets have been restricted in a similar way as the cargo set. Countries and continents are loaded from a location ontology and the direct condition "in {africa, northAmerica, europe, asia}" on the Continent set refers to the names of those ontological continent instances.

Shipping price

Muller does not publish shipping prices but provides an endpoint where prices can be inquired dynamically providing certain input data (like shipping addresses and the weight of the cargo). Such dynamicity is supported by the DIANE framework using *estimation operation* as introduced in Section 1. The marker OUT,e,1 on the price set in Muller's description denotes that the value for this set can be inquired by executing the associated first estimation operation. Accordingly both addresses and the cargo need to be given as input to inquire about the price (markers `IN,e,1`). Details on how the estimation operation is actually executed and how the handling of estimation operation is integrated into the matchmaking process will be given in Section 2.5. The other services (Runner, Racer, Weasel and Walker) did not offer an endpoint to inquire dynamically about the price and instead specified rules how to compute the price depending on the specifications of the shipping operations. Unfortunately DSD lacks direct support for such rule based computations. Therefore auxiliary endpoints to compute the shipping price have been created for those services similar to the one that was offered by Muller already. During matchmaking the computation of the shipping price is delegated to those endpoints in the same way that Muller's endpoint is used.

Pickup and shipping times

All services made similar restrictions to available pickup times for collection. In the case of Muller collection is possible between 7am and 8pm. This is encoded in Figure 2 using according direct conditions "`>= <07:00>`" and "`<= <20:00>`" on the properties of the `pickup` set similarly as described above. Additionally, collection is only possible in the future (but in the case of Muller no advance notice was required) and at most two working days in advance. To deal with these temporal aspects we exploited the fact that DSD instances are internally represented as Java classes and created special `Date`, `Time` and `DateTime` instances `now`, `today` or `todayPlusTwoWorkingDays`. These instances contain code that accesses the system time to capture the intuitive semantic. In the case of `todayPlusTwoWorkingDays` Sundays are not considered working days (which is why we could not use an expression like "`now + <P2D>`"). Finally the services made restrictions on the length of the pickup interval (in the case of Muller at least 90 minutes). This requires to pose a condition on an arithmetic combination of attributes (the end of the pickup interval minus the begin of the pickup interval has to be greater than a certain duration). Such so-called *multi attribute conditions* can be added to the conditions property of a `ServiceProfile`. In Figure 2 the condition "`$pickupEnd > (+;$pickupBegin,<PT90M>)`" references the sets labelled `$pickupEnd` and `$pickupBegin` and states that values for the former must be greater than values for the latter plus a duration of ninety minutes. Originally, this was not supported by DSD. If the required pickup interval is given in the request it is easy to check whether this interval adheres to the restrictions of an offer. If, however, a request does not give a precise interval but, for instance, only specifies that collection is impossible after or before certain times, the matchmaker has to determine an interval that suits both the requirements of the offer as well as the needs of the requester. Note that this interval has to be determined automatically in order to be able to configure the offer automatically to facilitate automated invocation of estimation as well as execution operations. However, in the presence of multi attribute conditions an optimal configuration of an offer cannot be determined locally for the attributes anymore. Currently, our matchmaker does not support globally optimized configuration under such conditions. The current implementation guarantees that any determined configuration is correct, but the algorithm is not complete. Under certain circumstances the matchmaker will fail to determine an optimal or even any valid configuration at all although such a configuration exists. It is planned to address this issue in our future work.

Finally, the offers declare the expected shipping times depending on the pickup time and whether the shipping is national or international. Muller for instance ships in 2/3 (domestic/international) business days if collected by 5pm. Like in the case of the shipping prices DSD does not support such rule based evaluations directly. To overcome this limitation we created auxiliary services to compute the shipping time within an estimation operation (exactly like the shipping prices). Therefore the `shippingTime` set is declared as an out value of the second estimation operation (marker `OUT,e,2`) and the addresses and the pickup interval are declared as input of that estimation operation (markers `IN,e,2`).

2.2 Request descriptions for the first scenario

Overall DSD request descriptions are built similarly to DSD offer descriptions. Figure 3 shows the request corresponding to Goal C3 of the first discovery scenario. The structure of the request resembles the one of the Muller offer. Addresses, and the cargo to be shipped are provided. While the price is specified as a set (denoted by the diagonal line in the upper left corner of the concept), addresses and the cargo to be shipped are provided as concrete instances because no variation is allowed by the requester. Note that the city instances ("`bristol`" and "`moonCity`") in the addresses refer to ontological instances, because the `city` property – unlike `street`, `email` or `zipCode` which are of the primitive `String` type – refers to a complex entity type with publicly known instances stored in the ontology. Thus 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.

No pickup or shipping time is specified since Goal C3 poses no requirements on these properties. It does however specify a price limit of \$20. This could have been modelled by a direct condition "`<= 20`" on the `amount` property of the `Price` set. Instead we chose to additionally model preference for lower prices by using a fuzzy `Double` set for the `amount` property. The fuzzy direct condition "$\sim$`==[0,20] 0`" requests the set to be fuzzily equal to 0 where the given interval `[0,20]` denotes the boundaries of the fuzzy equal. Thus the double value 0 will match perfectly (membership value 1), all values greater than 20 will not match at all (membership value 0) and values in between 0 and 20 will match with linearly decreasing membership value.

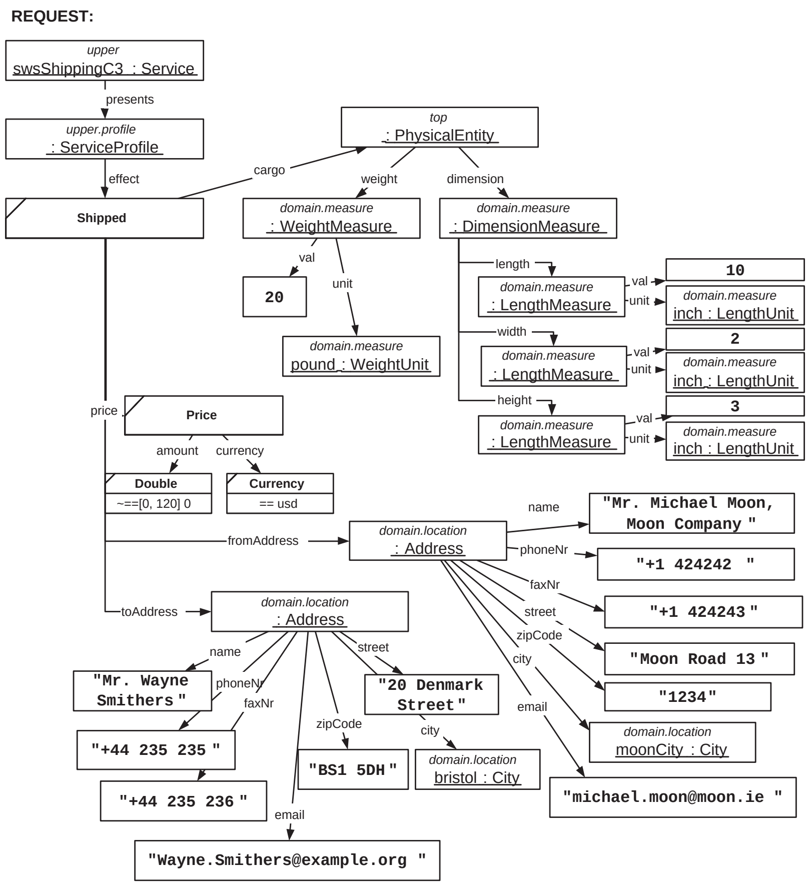

Fig. 3. Excerpt from the DSD request description of Goal C3

Goal D1 differs from the other discovery goals in that this goal asks for shipment of two packages and thus enforces two invocations of the corre-

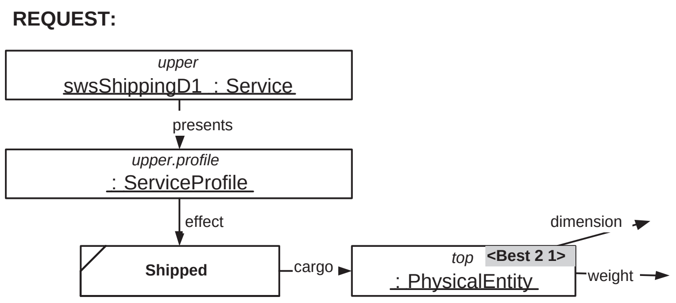

Fig. 4. Excerpt from the DSD request description of Goal D1

sponding shipping provider service. In our first 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* on any set in a DSD description may be used to change this semantic. In Figure 4 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 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. Allthough Goal D1 could be successfully modelled 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 work.

All shipping services of the scenario return the actual pickup time as well as the price of the shipping operation within the response of the final invocation. The scenario did not require to make use of this information. However, it is possible to specify this fact in the offer descriptions and to mark corresponding concepts in the request as *request out variables*, thereby declaring that this information is required by the requester as output of a service invocation. In this case the matchmaker ensures that the information indeed is provided by the offers at hand and links the request out variable to the corresponding concept in the offer. After the service invocation the DIANE middleware extracts the values from the response and forwards it to the requester. This way requirements on the output of an operation can be specified in a request and are guaranteed by the middleware.

2.3 Offer descriptions for the second scenario

The second scenario contains three descriptions of imaginary online shops that sell electronic products. A concrete list of 19 available products is given statically in the scenario description but a listing of available products by product type (like a list of all offered notebooks) can also be obtained by calling a specific operation at the service's endpoints. Solutions were supposed to indicate how they would address a more realistic situation with hundreds or thousands of available products which change dynamically.

We think it is unrealistic to assume that extensive catalogue data can be included in offer descriptions which are published to a service repository. Among the reasons are the dynamicity of a large catalogue which would require an enormous number of updates, the sheer size of service descriptions that enlist thousands of products but also privacy issues that keep providers from revealing too much information about their current stock. We deal with these issues in more detail in [KKR07c] and address the situation in the context of the SWS-Challenge by using *dynamic offer descriptions*.

Dynamic product listings

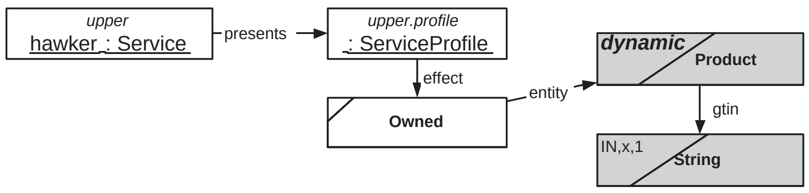

Fig. 5. Excerpt from the DSD description for the Hawker vending service.

Figure 5 shows the relevant excerpts from the offer descriptions of the Hawker vending service (all grounding information has been omitted). Without the listing of the products little information can be included in the static offer description. Basically the offer simply states, that Hawker sells products given it's GTIN number (a fictionary identifier used in the scenario) as input. One could use regular estimation operations to inquire about the available products. For performance reasons and since an offer like hawker is not very meaningful we created a special operation: Concepts tagged as *dynamic sets* may have an associated estimation operation that will be evaluated right at the beginning of the matchmaking process. Recall from the discussion about pickup times above that the matchmaker may have to determine appropriate values for inputs of estimation (and execution) operations if these are not provided in the request. To be sure that all input values of an operation have been determined already a complete traversal of the descriptions at hand is necessary. Since at the beginning of the matchmaking process it is not known whether all necessary input values have been determined by the matchmaker already, the corresponding operation must not have any specific `IN` variables. Instead the corresponding concept description from the request will be given as input. In the case of Hawker, Hawker's grounding simply extracts the type of Product seeked by the requester and then invokes Hawker's endpoint to list the available products of this type. The returned xml listing will be converted to a list of DSD based instances (using mapping information from Hawker's grounding) and Hawker's offer will be annotated with the retrieved instances, thus providing a concrete up-to-date listing of the available products. The matchmaking will then be performed based on that listing. By changing it's grounding and using additional information from the provided request concept beside the type of product requested, Hawker may finetune the procedure in order to avoid to return too long product listings.

2.4 Request descriptions for the second scenario

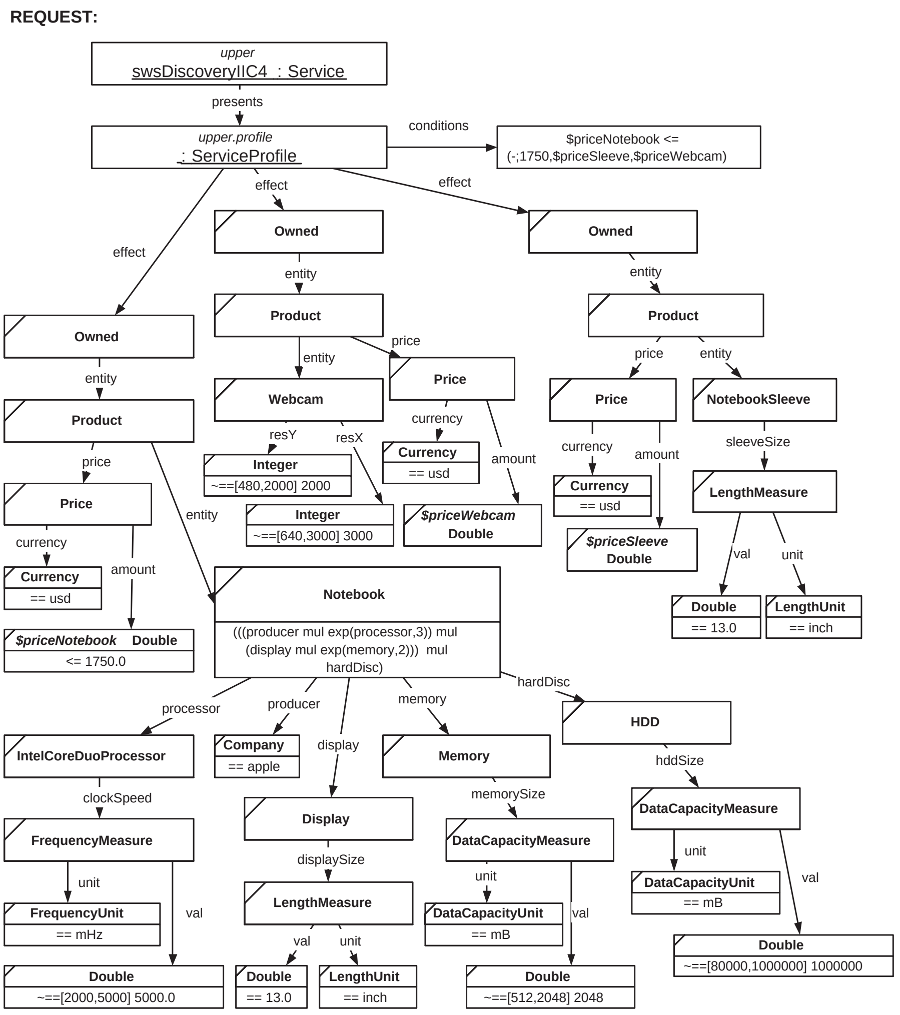

Fig. 6. Excerpt from the DSD description for Goal C4 showing preferences.

Figure 6 shows excerpts of the description corresponding to Goal C4 of the second scenario. The request asks to buy an Apple notebook, a webcam and a notebook sleeve with certain properties. The requests of the second scenario are more complex than those of the first scenario. In this section we will discuss the features of the requests that were not used within the first scenario in a similar way already.

Competing Request Preferences

Goal C4 states a price limit for the overall purchase but prefers better notebooks as long as that limit is satisfied. Additionally it defines a ranking of preferences to detail what constitutes a better product: more processor power is most important and more RAM is more important than a bigger harddisc.

DSD is very well suited to capture finegrained and competing preferences using fuzzy sets. Preference for more processor power, more memory, a larger harddisc and a higher resolution of the webcam are encoded using fuzzy direct conditions in the corresponding sets of the request description in the same way as preference for lower shipping prices was encoded in the first scenario. The processor speed may serve as an example. The fuzzy direct condition "∼`==[2000,5000] 5000`" in the value set of the processor speed attribute is used to build a fuzzy set of Double values. The requested value is 5000, but values from the range $[2000, 5000]$ are included fuzzily with linearly increasing degree of membership and preference. This encodes that the requester requires the processor speed to be at least 2000 mHz and prefers faster processors. Typically some notebooks may have a bigger harddisc while others may have more processor power. To rank notebooks in such a situation the importance of each attribute must be captured. In DSD this is done using fuzzy *connecting strategies*. To determine the degree of match of two DSD concepts (either two sets or a set and an instance) the types are compared, any direct conditions (like "`== usd`") are applied and the degree of match of every property is determined. By default the degree of match of the two concepts is then computed as the product of the type match value, the match value that results from applying the direct conditions and the product of the matching results from the property conditions. The way how the results from the property conditions are combined may be changed by specifying a custom *connecting strategy*. In the `Notebook` set in Figure 6 you can see a formula that corresponds to $producer \times processor^3 \times display \times memory^2 \times hardDisc$. Thus the results from the referenced properties are not simply multiplied but the result of the `memory` and `processor` property condition are squared respectively cubed. Since all matching values are from the closed interval $[0, 1]$ a lower result from the `processor` or `memory` property will have a stronger reducing influence on the overall matching value than results from the `hardDisc` property, thereby encoding stronger preference for a fast processor and a lot of memory compared to a large hardDisc. More information on how to encode user preferences using fuzzy sets can be found in [KKRM05, KKRKS07a]. Despite the fact that most preferences could be encoded very intuitively, one goal posed difficulties. Goal B2 of the second scenario prefers black notebooks compared to white ones, but prefers to buy the white one if it is significantly (more than \$100) less expensive than the black one. Such a rule-based preference does not map very well to the fuzzy set-based preference mechanism of DSD. Since the price limit was set to \$1800 a price difference of \$100 results in a difference of the matchvalue of roughly 0.055 (100/1800). Therefore the preference values for the color were chosen to reflect exactly this difference: black [1] and white [0,944]. A weighted sum of $0.5price \times 0.5color$ however does not result in the desired behaviour. The problem is that any black notebook regardless of the price or any product whatsoever that does not cost much would result in a matchvalue of at least 0.5. Therefore the connecting strategy was extended to $min(color, price, 0.5 \times price + 0.5 \times color)$. However, this way the matchvalue was soon dominated by the matchvalue for the price and the weighted sum didn't influence the outcome any more. To resolve this issue, the influence of the weighted sum was strengthened using a polynom yielding: $min(entity, price, (0.5 \times price + 0.5 \times entity)^5)$. This way the desired behavior could be successfully achieved.

Simple service composition

Beside competing request preferences and dynamic product listings the second scenario introduces some simple form of service composition in four different flavors. The above shown Goal C4 is the most complex one combining unrelated composition with global conditions and preferences as will be discussed in the following.

Unrelated composition

As can be seen, the request asks for three different products. This can be expressed by simply asking for multiple effects to be provided. Therefore Figure 6 shows three `Owned` effect sets, each corresponding to one article. The DSD matchmaker will use a multi-phased approach to match such requests. In a first phase offers are matched with regard to whether they are able to provide at least a subset of the requested effects. In a second phase offers are combined in a way that each combination (called *effect coverage*) provides each effect and each effect only once. In a third phase the matchvalue for complete effect coverages is determined and the best combination is picked. This algorithm has been introduced in [KKRKS07b]. Thus DSD is well capable of solving unrelated composition problems.

Unrelated composition with global condition

Goal C4 also states an overall price limit of \$1750 for the complete purchase. This can be expressed in DSD by using the multi attribute conditions that were introduced in Section 2.1. The condition shown in Figure 6 reads as $\$priceNotebook \leq 1750 - \$priceSleeve - \$priceWebcam$ and encodes the given requirement (\$priceNotebook, \$priceSleeve and \$priceWebcam reference the value of the sets which are labeled accordingly). This way the goal could be expressed and solved but the same limitations that were discussed in Section 2.1 apply. Our algorithm is correct but currently not complete. Goal C4 needs to be solved by composing different properly configured services. The right services need to be chosen on a global level and the right products need to be purchased from each of these services. Our current matchmaking algorithm splits a request for multiple products into a a set of single requests, one for each product to be purchased. When determining the configuration of a service (i.e. choosing a product), it performs a local optimization with respect to the requirements of the desired product at hand instead of a global optimization with respect to the total set of requested products. Depending on the available products and the concrete global request at hand it may happen that after the first product is chosen, too little money remains to add the other two products. In this case the matchmaking will fail and no product will be purchased. A backtracking mechanism could ensure completeness in such cases. Backtracking, however, could result in an exhaustive search of the configuration space of the services at hand, which is not feasible anymore. Therefore, a more intelligent solution is needed. For many cases the problem at hand could likely be overcome with techniques from constraint optimization problem solving. Furthermore, a solution to a similar problem was presented in [KKRKS07b]. We are therefore optimistic to be able to solve the issue in our future work.

Unrelated composition with global condition and preference

Preferences have been discussed above already and the combination with composition does not add any difficulties in the setting of our solution.

Correlated composition

Goal C2 requests a notebook and a compatible docking station. Thus the two requested products are correlated and cannot be handled seperately. DSD is capable to express such correlations and to correctly compose and configure multiple offers accordingly [KKRKS07b]. Unfortunately, we were nevertheless unable to solve goal C2 correctly. Compatibility of docking stations and notebooks in the scenario is given by a property of each docking station that holds a list of the GTINs of the compatible notebooks. To ensure whether a notebook is compatible to a docking station one has to check whether the notebook's GTIN is contained in the docking stations compatibility list. Currently the DIANE framework lacks sufficient support for matching of list-based attributes to handle this case.

2.5 Service interactions

In this section we provide details about how DIANE performs the necessary interactions with a service endpoint to facilitate automated service consumption. We first deal with how estimation operations are integrated into the matchmaking process and then describe how the actual invocations of the services are carried out.

Integrating service interactions into the matchmaking

As mentioned in Section 1 our matchmaker follows a multi-phased approach [KKRKS07b]. The main idea behind this decision is to reduce the number of offers remaining in the matchmaking process before the most expensive matchmaking tasks are performed. The basic idea of the matchmaking algorithm is to traverse the request description tree and to match each concept r_i from the request with the corresponding concept o_i from the offer. As mentioned before the matchvalue of r_i and o_i is thereby built by comparing the types of the concept, applying any direct conditions and then combining the match values retrieved from recursively comparing the properties (property conditions) of r_i and o_i. This structured approach to matchmaking allows to collect precise information about which parts of an offer did not match with the request. In a first run not only obviously unsuitable offers are filtered, but also information about whether a particular estimation operation should be executed is collected. This is the case when a concept from an offer that is declared as estimation out variable was neither a perfect match nor a definite fail using static information alone.

Thus after a first run only the estimation operations that offer information about such concepts will be executed [KKR07c]. When matching those goals of the shipping scenario for instance, which do not specify a price limit, the corresponding price information will not be gathered, since it has no influence on the outcome of the matchmaking. Similarly when matching Goal C3 of the shipping scenario with the available offers, the actual price of the Weasel offer will not be inquired since it is already known after the first matching run that Weasel does not ship to the United Kingdom and is therefore unsuitable anyway.

After the estimation operations are executed, the information returned by the service endpoints will be used to update the offer descriptions and another matchmaking run will be performed, yielding the most accurate and up to date results possible.

Performing service invocations

The information that is necessary to automatically invoke a service (regardless whether this is in the context of an estimation or execution operation) is specified in the grounding part of an offer description. Figure 7 shows excerpts from the grounding specification of the Bargainer service. Two SOAP operations are defined, the first used to execute the service, the other one used to complete the service description (i.e. to gather the dynamic product listings). Both specify the SOAP action header to use (`soapAction`) and the endpoint to call (`endpoint`). The `setReference` property of the `SOAPOfferCompletionOperation` is used to map the operation to the dynamic set it belongs to. To lower ontological DSD data to XML messages to be sent to a service and to lift XML data extracted from the service's response

```
supports = anonymous SOAPServiceGrounding at upper.grounding [
        // THE ORDER OPERATION
        soapOperations += anonymous SOAPExecuteOperation at upper.grounding [
                soapAction = "order",
                xmlTemplatePath = "bargainerOrderProductsTemplate.xml",
                endpoint = "http://sws-challenge.org/shops/Bargainer",
                mappingIN += anonymous XmlDsdMapping at upper.grounding [
                        // ommitted due to space limitations
                ]
        ],

        // THE DYNAMIC OFFER COMPLECTION OPERATION
        soapOperations += anonymous SOAPOfferCompletionOperation at upper.grounding [
                setReference = $products,
                soapAction = "list",
                xmlTemplatePath = "bargainerListProductsTemplate.xml",
                endpoint = "http://sws-challenge.org/shops/Bargainer",
                mappingIN += anonymous XmlDsdMapping at upper.grounding [
                        // ommitted due to space limitations
                ],
                mappingOUT += anonymous XmlDsdMapping at upper.grounding [
                        // ommitted due to space limitations
                ],
                ...
        ]
        ...
],
```

Fig. 7. Excerpt from the grounding of Bargainer's offer description

to ontological DSD data, DSD follows a pragmatic approach that was introduced in [KKR06b]. For each operation an empty XML message template has to be deployed together with the service description at the DIANE middleware. The `xmlTemplatePath` property of the specified operations in Figure 7 points to that file. Mappings have to be specified in the grounding that define how to fill the template with the values from the properly configured offer description.

Figure 8 shows excerpts from the mapping definitions from the Bargainer offer's grounding used for the `SOAPOfferCompletionOperation`. `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`). Depending on the type of the variable standard serialization is available, but in the case at hand a custom Java class is specified and used to deliver the proper product type in Bargainer's classification for a given product (`converterClassName` and `converterMethodName`). This class has to be deployed at the DIANE middleware and will be instantiated using reflections.

Once the given XML template is properly filled the correct message will automatically be sent to the corresponding endpoint. The reply needs to be interpreted to make the results of the service invocations available to the middleware, either to return them to the service requestor or to use them during the matchmaking process.

This is accomplished by `mappingOUT` definitions that are used to process the outputs of an operation, in this case by lifting the XML listing of available products to DSD instances. They work similar to `mappingIN` definitions but

```
mappingIN += anonymous XmlDsdMapping at upper.grounding [
        variable = $products,
        dataNodePath = "ProductCategory",
        converterClassName = "org.swschallenge.shops.ProductCategoryConverter",
        converterMethodName = "convert"
],
mappingOUT += anonymous XmlDsdMapping at upper.grounding [
        variable = $products,
        dataNodePath = "/ProductList/Product",
        // gtin
        attributeMappings += anonymous XmlDsdAttributeMapping at upper.grounding [
                attributePath = "gtin",
                subNodePath = "productID"
        ],
        // mapping for Notebooks and NotebookDescriptions
        attributeMappings += anonymous XmlDsdAttributeMapping at upper.grounding [
                subNodePath = "self::node()[productCategory=\"Notebook\"]",
                className = "dsd.schema.domain.computer.Notebook",
                attributePath = "entity",
                attributeMappings += anonymous XmlDsdAttributeMapping at upper.grounding [
                subNodePath = "name",
                attributePath = "deviceSpecs"
        ],
        // elements of ProductDescription
        // macs are produced by apple
        attributeMappings += anonymous XmlDsdAttributeMapping at upper.grounding [
                subNodePath = "prodDescription[contains(brand,\"Mac\")]",
                attributePath = "producer",
                converterClassName = "[...]util.converter.InstanceNameConverter",
                converterMethodName = "getInstance",
                constantValue = "dsd.instance.domain.economy.Company.apple"
        ],
        ...
```

Fig. 8. Mapping definitions from the grounding of Bargainer's offer description

in the example in Figure 8 illustrate some more features. As shown in the example, mappings can be specified in a nested way (which allows to handle nested lists). The `subNodePath` and `attributePath` properties identify an attribute of the variable and a descendent node of the XML node used by the parent mapping definition. If the `subNodePath` XPath expression evaluates to an empty list the mapping is not executed which allows to specify different mappings e.g. for different types of products (like notebooks, docking stations, etc.). In Figure 8 this is used to set the producer of a product to the instance `apple` if and only if the brand node in the given XML contains the string "`Mac`". The name of the instance to instantiate is provided as static value in the mapping and a converter class is used to retrieve a complex entity instance by its name. The lightweight mapping mechanism described above was sufficient powerful and flexible enough to not only cover the automated invocations in the shipping discovery scenario but also to handle the dynamic product listing of the second scenario.

3 Discussion and Summary

We have described how DSD and the DIANE middleware has been used to solve nearly all parts of the SWS-Challenge discovery goals. In this final section we will briefly discuss the lessons learned and the strengths and weaknesses of the DIANE approach in a structured way.

3.1 Domain ontologies

The scenarios did not require very heavy-weighted ontologies with rules and restrictions. Thus our lightweight ontology language was very well suited to describe all aspects of the domains at hand and we did not encounter any difficulties there. In particular the temporal semantics could be addressed easily by creating special DateTime instances that directly capture the temporal semantics of *now* or *today* by accessing the local system time.

3.2 Offer descriptions

Most aspects of the offers like the operation range of the shippers, the restrictions on package size or weight or the rates on request of the Muller shipping offer could be easily modelled in DSD. The main limitation with regard to the offers was the lack of direct support for rules. In the first scenario rules were needed to compute shipping prices and expected shipping times in dependency of certain attributes of the shipment. We circumvent this limitation by delegating the evaluation of rules to external entities (in our solution we used web services, but we could have used local method calls or any other mean, too). The integration of this delegation could be easily done since DSD already supported to gather additional information during the matchmaking (estimation operations). Some constraints on the possible collection time in the shipping scenario required to pose conditions on arithmetic combinations of different properties of a service description. Originally DSD was lacking support for such conditions but this feature (multi attribute conditions) has been added to DSD. However, as discussed in Section 2 this feature is problematic in combination with optimal offer configuration (which can not be achieved efficiently anymore). This is one of the fields of future work.

3.3 Request descriptions

Most aspects of the goal descriptions could be easily expressed in DSD. Shipping discovery based on destination, weight, price, temporal requirements or any combination could be directly solved. The same is true for the product purchasing scenario goals. DSD proved very capable of expressing fine-grained user preferences in requests via fuzzy sets to enable powerful ranking of services. Modelling of those preferences was very straightforward and intuitive except for one case where preferences were given based on rules (see Section 2.4). The composition goals of both scenarios could be expressed (except for Goal C2 which will be discussed below). In the second scenario we chose to ask for multiple effects in the request. In the one composition goal of the first scenario we chose to use iteration directives that change the set-based semantics of that request. Although the iteration directives were sufficient for the problem at hand, their implementation within the DSD Middleware as proof of concept is currently still incomplete and needs to be completed. This is subject of ongoing work.

3.4 Reasoning and matchmaking

DSD does not rely on standard logic for matchmaking but uses a custom set-based reasoning operation *subset*. This allows to express request preferences using fuzzy sets and in particular acknowledges the fact that offers usually need to be configured and should be configured in an optimal way. Thus DSD matchmaking does not only check whether an offer instance is suitable for a request instance, but also determines the best configuration of an offer. DSD has been designed to do this efficiently without iterating over all possible configurations, largely by allowing for local optimizations in many cases. This interferes with the multi attribute conditions that have been introduced to capture certain restriction on pickup times in the first scenario (see Section 2.1) and global restrictions on the price of a complete purchase in the second scenario (see Section 2.4). Due to performance considerations our current matchmaking implementation does not guaranteed anymore that the determined configuration is optimal or that an existing valid composition is found if offers or requests use multi attribute conditions. To improve on this issue is ongoing work.

Finally, DSD allows to use lists as properties of a concept. Unfortunately the current matchmaking implementation does not completely support such properties. This prevented us from solving Goal C2 of the second scenario (where compatibility of notebooks and docking stations is given as a list property of the docking stations that lists the compatible notebooks). To add the necessary support for matchmaking of list-based properties is future work, too.

3.5 Service interactions

DSD and DIANE have been designed to support fully automated invocation of services, thus the need to execute service operations (estimation or execution operations) did not pose severe difficulties to our approach. The pragmatic approach to mapping between XML and DSD data has proven to be sufficiently flexible and powerful to cover the scenarios. The practical experience however has shown that it is quite cumbersome and in particular error-prone to define these mappings without appropriate tool support. This has highlighted once more that powerful editing tools (which DIANE is currently still lacking) are an essential prerequisite for more widespread or daily use of any semantic web service technology.

Regarding estimation operations we believe that these are a particular strength of our approach. In [KKR07c] we argue that the ability to include dynamic information into the matchmaking is essential for any service matchmaking framework. Unfortunately however, this can easily compromise the efficiency of any matchmaking algorithm since the matchmaking time will quickly be dominated by the time spend to call external webservice's endpoints to gather that dynamic information. It is thus a key strength of DIANE that its structured graph-matching approach to service matchmaking allows to precisely determine which parts of an offer description matched how well – an important difference to the related work. This knowledge can then easily be used to inquire precisely only that dynamic information which will influence the outcome of the matchmaking.

3.6 Difficulty to switch from one problem level to another

It was one of the assumptions of the SWS-Challenge that the advantage of using semantic technology compared to traditional programming should be proven by showing that semantic based approaches would cope more easily with changes in the scenarios.

In our experience quite a bit of effort was involved in building a first running solution to the challenge. This is mainly due to two reasons. First, DIANE - as a research prototype - is partly lacking the tool support that one would wish to have (this is particularly true for the grounding definitions). Second, some scenarios required to add new features to the DIANE framework. However, to add these features (like multi attribute conditions) to the framework is a one time effort related to the development of the language and framework and should pay off when more scenarios become available that make use of these feature but do not require new ones.

Aside of these issues we do not feel that a lot of effort was necessary to switch from one problem level to another one. In particular little effort was needed to move from the first scenario to the second one. This is due to the fact that DIANE uses a generic set-based and not a domain-dependent rule-based approach to matchmaking. The principle behind DSD is to describe what offers can provide, what requests are seeking and have the matchmaking done by generic domain-independent matchmaking rules. Thus, when switching from the first scenario to the second scenario, we had to create the needed domain ontologies (describing IT hardware) and we had to describe the offers and requests. However, we did not have to specify matchmaking rules since these remain the same for all scenarios.

References

[KKR06a] Ulrich Küster and Birgitta König-Ries. Discovery and mediation using diane service descriptions. In *Third Workshop of the Semantic Web Service Challenge 2006 - Challenge on Automating Web Services Mediation, Choreography and Discovery*, Athens, GA, USA, November 2006.

[KKR06b] Ulrich Küster and Birgitta König-Ries. Dynamic binding for BPEL processes - a lightweight approach to 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, December 2006.

[KKR07a] Ulrich Küster and Birgitta König-Ries. Semantic service discovery with DIANE service descriptions. In *Proceedings of the International Workshop on Service Composition & SWS Challenge at the 2007 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2007)*, Silicon Valley, USA, November 2007.

[KKR07b] Ulrich Küster and Birgitta König-Ries. Service discovery using DIANE service descriptions - a solution to the SWS-Challenge discovery scenarios. In *Fourth Workshop of the Semantic Web Service Challenge - Challenge on Automating Web Services Mediation, Choreography and Discovery*, Innsbruck, Austria, June 2007.

[KKR07c] Ulrich Küster and Birgitta König-Ries. Supporting dynamics in service descriptions - the key to automatic service usage. In *Proceedings of the Fifth International Conference on Service Oriented Computing (ICSOC07)*, Vienna, Austria, September 2007.

[KKRK06] Ulrich Küster, Birgitta König-Ries, and Michael Klein. 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, June 2006.

[KKRKS07a] Ulrich Küster, Birgitta König-Ries, Michael Klein, and Mirco Stern. DIANE - a matchmaking-centered framework for automated service discovery, composition, binding and invocation on the web. *International Journal of Electronic Commerce (IJEC)*, 12 - Special Issue on Semantic Matchmaking and Retrieval(2), 2007.

[KKRKS07b] Ulrich Küster, Birgitta König-Ries, Michael Klein, and Mirco 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.

[KKRM05] Michael Klein, Birgitta König-Ries, and Michael 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):328–364, 2005.