DIANE - An Integrated Approach to Automated Service Discovery, Matchmaking and Composition

Ulrich Küster and Birgitta König-Ries
Institute of Computer Science
Friedrich-Schiller University Jena
Jena, Germany
ukuester|koenig@informatik.uni-jena.de

Mirco Stern and Michael Klein
Institute for Program Structures and Data Organization
University Karlsruhe, Germany
mirco.stern|kleinm@ipd.uni-karlsruhe.de

ABSTRACT

Automated matching of semantic service descriptions is the key to automatic service discovery and binding. But when trying to find a match for a certain request it may often happen, that the request cannot be serviced by a single offer but could be handled by combining existing offers. In this case automatic service composition is needed. Although automatic composition is an active field of research it is mainly viewed as a planning problem and treated separatedly from service discovery. In this paper we argue that an integrated approach to the problem is better than seperating these issues as is usually done. We propose an approach that integrates service composition into service discovery and matchmaking to match service requests that ask for multiple connected effects, discuss general issues involved in describing and matching such services and present an efficient algorithm implementing our ideas.

Categories and Subject Descriptors

H.3.5 [**Information Systems**]: Online Information Services—*Web-based services*; D.2.12 [**Software Engineering**]: Interoperability; D.2.m [**Software Engineering**]: Miscellaneous

General Terms

Algorithms

Keywords

Automated service composition, service discovery, service matchmaking

1. INTRODUCTION

An important vision of service oriented computing is to enable dynamic service binding, i.e., it should become possible to automatically choose and invoke service providers at runtime. To achieve this, appropriate means to describe and match services are needed. In our previous work we have presented such means, namely a service description language and an efficient matching algorithm for this language [5, 6, 7]. However, when trying to find a match for a certain request it may often happen, that the request cannot be serviced by a single offer alone but could be handled by combining existing offers. Traditional service matching mechanisms fail in these cases but techniques of automatic service composition offer a solution. Although automatic composition is a very active field of research it is mainly viewed as a planning problem and usually treated separately from service discovery. Most composition approaches do not deal with semantic discovery at all and many use proprietary request (or better goal) description languages suitable for planning and composition, but not for discovery. Furthermore, most frameworks for automated service composition have no or at least only a very limited notion of the quality of a created composition or preferences among alternative compositions. This is partly due to the fact that classical planning (where many service composition approaches root in) usually does not even attempt to find an optimal plan, but only to find a plan at all. Furthermore, expressing precise preferences that usually involve competing optimization goals (like quality versus price) is a difficult problem on its own. Clearly automated service composition could benefit here from service discovery and matchmaking where preferences among competing offers naturally play a central role. Thus, in this paper we propose an approach that integrates service composition into service discovery and matchmaking. On the one hand we can leverage previous work on preference modelling for use in automated service composition, on the other hand, we increase the likelihood to be able to service a given request in a fully automated fashion by integrating automated composition into service discovery and matchmaking. In this paper we focus on a class of service requests that ask for multiple connected effects (like purchasing of a good *and* delivery of the purchased good to a certain destination). We discuss general issues involved in describing and matching such services and extend the matching algorithm introduced in [5, 7] in theory and practice to be able to successfully match such requests.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

2. A MOTIVATING EXAMPLE

Imagine an automotive manufacturer with several locally distributed factories. At each site, several parts are needed to construct the cars, most of which are purchased from external component suppliers and delivered to the appropriate sites by third party shippers. As an example, a factory in Karlsruhe could

- require a quantity of screws with a countersunk head, zinc plated and a size of 5x40 mm
- which have to be delivered to Karlsruhe ideally on December 8th, but not earlier than December 3rd.

In order to detect appropriate service providers, these effects need to be captured by a semantic service request description. We call requests like this one, that contain more than one effect (in this case purchasing and delivery) which are dependent on each other (e.g. delivery has to depart at the location where the screws are purchased), requests with *multiple connected effects*. Generally the formulation of any service request should be driven by the desired effects, only. It should not be necessary for the requester to take into consideration which services are available at any given time. This is particularly important in dynamic environments, where the set of service providers may change over time. While the manufacturer will probably need screws on a regular basis, instead of statically binding to a certain (combination of) provider(s), the most appropriate service providers should be identified anew each time the respective request is issued. Thereby, we ensure that each time the best matching provider is chosen (e.g., changes in prize or quality are taken into account) and that we consider all service providers that are available at this point of time and only those.

The requester does not necessarily have knowledge about the service landscape at execution time, and thus, doesn't know at which granularity services are offered. This implies, that it should not be necessary for the requester to divide the request into parts so that each part can be fulfilled by a single service offer. As an example, the following set of offers could be present:

- Service 1: a component supplier could sell 2000 different types of high quality screws to customers within the EU. Screws can be picked up at a number of listed warehouses (e.g. Hamburg, Paris, Berlin, ...).
- Service 2: a shipper could deliver goods within Germany, Austria and Switzerland, if the goods do not need any cooling or freezing and their packages adhere to certain maximum weights and sizes.
- Service 3: a component supplier could sell and ship 1000 different types of screws and bolts to customers within Germany and Austria.

For a human, it is quite obvious, that either a combination of the first two services or the third service alone might be able to fulfill the service request. For an automated matcher this problem is far from trivial. The matcher does not only have to realize that a combination of services provide the desired effects, it also has to ensure certain constraints on the composition like whether Service 2 is able to ship from Service 1's location to Karlsruhe and whether the package that Service 1 will provide the screws in adheres to Service 2's restrictions on weights and sizes.

This is, where planning-based approaches to composition have their weakness. A planner typically will be able to determine that the request could be answered by a combination of a manufacturer and a shipment service. However, since these planners are executed separate of service discovery, the enforcing of the constraints has to be handled separately. Also, there is no guarantee, that suitable services will be available at runtime. In our approach, on the contrary, only currently valid offers are taken into consideration, constraint enforcement is an integrated part of service composition and different combinations of providers are ranked according to the requester's preferences.

The goal of our approach is to build an automated matcher that is able to compose services, provides fine-grained and precise ranking among competing offers (single ones as well as automatically composed ones) and is able to automatically invoke the best offer. Thus, we need the ability to express constraints like the ones listed above within the service descriptions and we need an algorithm that is able to handle requests containing multiple connected effects.

In the following two sections we will quickly recap previous work on describing services and matching such descriptions without the ability to cover multiple connected effects. In Section 5 we will introduce a conceptual foundation for extending the language and the matching algorithm to cover multiple connected effects and enable automatic composition for these cases. In Section 6 we will explain the implementation of our ideas in our matching algorithm in detail. Thereafter, we will evaluate our approach, give an overview of the related work and conclude.

3. WHAT IS DSD?

DSD (DIANE Service Descriptions) is a service description language based on its own light-weight ontology language that is specialized for the characteristics of services and can be processed efficiently at the same time. For the following an intuitive understanding of DSD is sufficient, since the concepts of the matching algorithm are not specific to DSD. We will explain the basic features of DSD on the basis of a simple example shown in Figure 1 that shows a request for a shipment service formulated in DSD. More comprehensive information about DSD and the matching algorithm can be found in [5, 6, 7].

The basis for DSD is standard object orientation which is extended by four additional elements:

- *operational elements:* Services change the state of the real-world (or the information space). Operational elements allow to express this world-altering capacity. In Figure 1, you'll find the operand *effect*, describing that we are looking for a service that creates a *state* of the world where something is shipped. We view these operational elements as 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. Concepts like states are declaratively defined which leads to descriptions as trees as seen in Figure 1.

- *aggregating elements:* A service is typically able to offer not one specific effect, but a set of similar effects. A shipment service for instance will be able to offer transport of different kinds of goods from one arbitrary destination to another. That means, services offer to provide one out of a set of similar effects. Requesters on the other hand, typically are looking for the perfect service, but accept services that deviate from this service to a certain degree with lower preference (e.g. a slightly more expensive service is acceptable if no cheaper service is available). Thus, the effect of a service (request or offer) is typically not a single but a *set of states*, depicted in DSD by a small diagonal line in the upper left corner of a concept. This way, offers describe the set of possible effects they can provide and

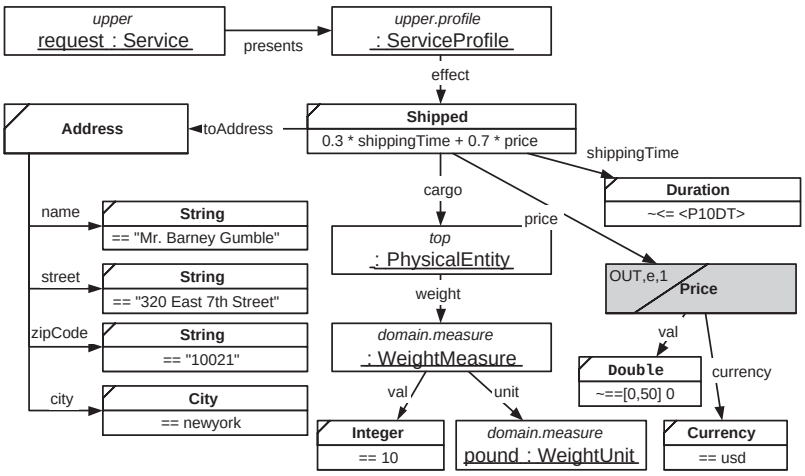

Figure 1: Simplified DSD shipping request

requests describe the set of effects they are willing to accept.

- *selecting elements:* While a service will offer different effects, upon invocation, the requester needs to choose which of these effects the service should provide in this specific instance. Meanwhile, after service execution the requester might need to receive results about the specific effect performed by the invocation. In DSD, *variables* (denoted by a grayed rectangle) are used to support this. In offers, variables allow the requester to configure the offer and to choose the particular desired effect to provide. In the request seen in Figure 1, the requester requires to retrieve the exact price of the shipment after service invocation.

- *rating elements:* This type of element is used in service requests only. As mentioned, requesters will typically be willing to accept services with slightly differing effects although with differing preference. These preferences can be expressed by fuzzy instead of crisp sets in request descriptions – the larger the membership value, the higher the preference. Rating elements are a key feature of DSD and are the main feature used to choose among competing offers. This will be explained in more detail in the following section.

4. THE BASIC MATCHING ALGORITHM FOR SINGLE EFFECTS

A matcher for service descriptions generally has two tasks: First and foremost it has to determine if - and how well - an offer description fits a given request description. But if service offers are configurable (and usually they are) it also has to find an optimal configuration of the offer. This is necessary to determine precisely how well an offer fits a request and to be able to automatically invoke the best offer service later [8]. In DSD this configuration is done by choosing and assigning concrete values to all variables in an offer description during the matching process.

Since any description language should be capable of expressing preferences among competing offers, the result of matching a request with an offer should not be a Boolean value but rather some metric of the degree of correspondence. In the following we refer to this result as a "matching value" normalized in the interval [0,1]. The basic problem of matching a DSD offer o against a DSD request r can thus be stated as follows: *Compute the fuzzy containment value (out of* $[0,1]$*) of o's effect sets in r's effect sets and - while doing this - where possible configure o in a way that maximizes this value.* Since DSD request as well as offer descriptions are trees stemming from similar ontological concepts (classes), an obvious basic technique for comparing both descriptions is a graph matching approach. Beginning with the root element of type Service, the two descriptions are traversed synchronously and compared step by step. This algorithm has been introduced in [5, 6, 7].

The matching value of a leaf of a description is computed using the fuzzy membership value of the offer element set in the corresponding request element set. In our example (see Figure 1), the requester is willing to pay any price below 50 USD but would prefer lower prices over higher ones[1]. Alike the requester would like to have the shipping time less than ten days, but is willing to accept a standard deviation up to ten percent again with linearly decreasing preference. The matching value of an inner node is basically computed by combining the matching values of it's child nodes. The way how to do this (called *connecting strategy*) can be configured by a requester. Naturally any requester prefers shorter shipping time and lower prices, but in the example the requestor emphasizes that the prize is more important to him than the shipping time by specifying how to compute the membership value for the Shipped set by using a weighted sum $(0.3 * shippingTime + 0.7 * price)$ of the membership values for the shippingTime and the price. DSD's rating features (fuzzy sets combined with customizable connecting strategies) allow to prescribe the desired effect very precisely yet allow for a certain flexibility, thus maximizing the likelihood of finding an appropriate service.

The issues related to the optimal configuration of an offer during the matching process are beyond the scope of this paper (see [7] for details) but it is important to mention

[1]Actually the formula "$\sim==[0,50]\ 0$" denotes that a price of 0 is requested, but a deviation up to 50 is accepted with lower preference.

that DSD has been designed in a way that largely allows the matcher to optimize variable fillings locally. This ensures efficient processability. To maintain this feature is one of the biggest obstacles in extending the basic matching algorithm for single effects[2] to cover multiple connected effects also. We will illustrate this issue in the next section.

5. CONCEPTIONAL FOUNDATION OF HANDLING MULTIPLE EFFECTS

Although at first glance it seems straightforward to extend the above explained procedure that is capable of handling single effect services to also cover multiple effects, on closer inspection some problems arise. These problems are mainly due to the fact that multiple effect services usually contain relations between effects. Recall the example from Section 2. Here, the place of manufacture needs to correspond to the location where the shipping company picks up the goods. Also, the restrictions on the shipping service's delivered weights and sizes need to correspond to the packages used by the manufacturer.

Such connections between effects are the main reason why we need service descriptions containing multiple effects. If the effects were unrelated, the user could as well simply pose multiple requests each one consisting of a single effect. It turns out that when using straightforward extensions to the service description language the matchmaking becomes inefficient and doesn't scale anymore. This problem as well as our solution is in the focus of this section.

5.1 An Intuitive Approach

The central issue in describing a service covering multiple effects is how relations between those effects can be expressed. Regarding our example, we are looking for a way to request a particular quantity of screws (Effect 1: owning screws) as well as their delivery within a certain amount of time (Effect 2: delivery). Since at the time of establishing the request it is unclear which suitable manufacturer is available, it is also unknown where the delivery departs. All that can be said is that the shipping service has to pick up the screws where the manufacturer is located. That is, the place where the manufacturer is located and the departure of the delivery are the same. Furthermore, the purchased screws are the good that is to be transported. This is important for instance to compute the price of the shipment depending on the weight of the good.

An intuitive approach to describing such a connection is by having both effects pointing to the same concept. This is shown in Figure 2 which shows the central aspects of a request description corresponding to the example introduced in Section 2. Clearly, this is sufficient to express the desired request. However, it breaks the tree structure of the service description which poses a problem when variables have to be filled by the matcher. In the example the address of the preferred warehouse needs to be given to the screw manufacturer and the address of that warehouse as well as certain properties of the screws need to be given to the shipping service. But the address of the warehouse cannot be optimized locally. If one wants to match this request with a composition of two distinct offers, one for purchasing the screws and one for shipping them, several problems arise.

[2]Note that the term *single effect* is somewhat misleading since the above mentioned procedure does work for multiple effects if they are unconnected and thus do not break the tree structure of DSD service descriptions.

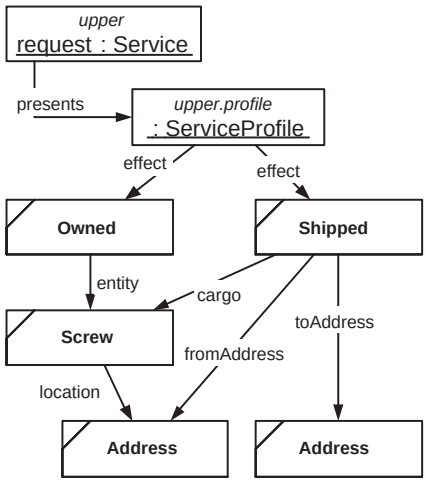

Figure 2: Intuitive approach for connected effects.

Assume that we found a manufacturer having warehouses in Hamburg and in Paris and assume that we choose the warehouse in Paris since this is closer to Karlsruhe where we need the screws. It might turn out later when configuring the shipping provider that it would have been much better, if we had chosen the warehouse in Hamburg because shipping within Germany is cheaper than shipping between France and Germany. It might even turn out that we cannot find a shipper that operates between France and Germany at all. Thus the location of the warehouse cannot be optimized locally. To find the optimal configuration a matcher would have to build all possible combinations of warehouses and shipping options and compare each with the request. Note that there are two sources of choice for the matcher. There may be different service offers (like competing screw manufacturers) as well as different configurations of each of these offers (like different screw types and warehouses of a single manufacturer). Thus, more precisely, the matcher would have to build all possible combinations of all configurations of all different offers. In general, with n effects, the complexity amounts to $O(m_1 \cdot m_2 \cdot \ldots \cdot m_n)$ where m_i is the number of different configurations of the various offers for one effect that have to be compared against the corresponding requested effect i, which can be estimated to $O(m^n)$ for m being the maximum of all m_i. This complexity is not tractable in practice, since m can grow to very large numbers: Even if there are not that many suitable offers (which is reasonable to assume), there will usually be many ways to configure each offer, e.g. shipping services will offer shipment to a huge number of locations, possibly yielding different prices.

Thus, we argue that configuring multiple effects by using an intuitive *global optimization* of the fillings isn't a suitable approach for on the fly composition. The central problem here is that the time performance of the algorithm will drastically deteriorate. Opposed to our example in Figure 2, the number of possible parameter fillings can be very high in *every* effect involved. In addition, the number of connected effects can easily exceed two. For instance, the requester could demand an insurance for the delivery, whose price could also

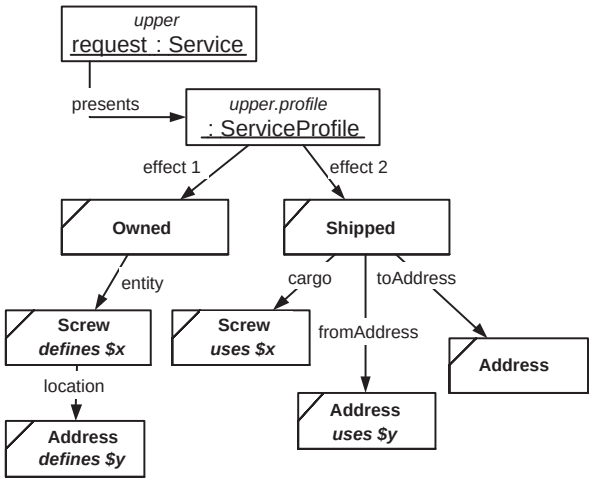

Figure 3: Alternative *value propagation semantics*.

depend on the departure and destination of the route.

5.2 "Value Propagation" - Semantics for Describing Multiple Connected Effects

In order to address this problem, we propose a "*value propagation*"-semantics, i.e. the idea is to modify the semantics of when an offer is considered optimal. At the heart of our proposed solution is the introduction of an ordering on the effects. This ordering is defined by the requester who thereby is given a further ability to express his preferences. The idea now is to *locally optimize* the parameter's fillings based on the ordering as defined by the user. This approach is illustrated in Figure 3. Here, the effect of buying screws is considered to be more important by the user and thus is chosen to be effect number one. Consequently, this is the effect that is processed first by the matchmaker. The parameters are filled optimally with respect to this effect with the only constraint that the filling must not be chosen in a way that completely precludes to find a match for the other effects[3]. Thus, the effect is regarded mainly in isolation and defines the location, which is stored in a variable $\$x$.

The chosen configuration might restrict the possible choices for those parameters contained in the following effects. Regarding the example, after having chosen the warehouse's location in x, this value restricts the possible fillings of the route's departure (in fact, it completely determines the filling which, in general, isn't necessarily the case). In order to take this into consideration, these restrictions (which can only occur on connections) will be propagated in order to be taken into account when configuring the following effects, hence the name "value propagation".

Using this approach we first have to annotate all offer descriptions we plan to use combined to service a request to prevent choosing a configuration for one offer that precludes to combine it with the other offers. This can be done in linear time in the number of configuration options as will be explained in Section 6. We then can match the effects in isolation, thus we have to calculate m_1 matching values for the first effect, take the optimal result and calculate the m_2 matching values for the second effect with it and so on. Altogether, this way the matching algorithm has a complexity of $O(m_1 + m_2 + \ldots + m_n)$ or $O(m \cdot n)$ if m is the maximum of all m_i, thus leading to a linear complexity. Naturally this comes at a price. Although our modified semantics yields a well defined result which can be stipulated by the user (by ordering the effects), we are not looking anymore for a globally optimal solution. However, we believe that this restriction is not critical: In real world examples, effects will often be naturally ordered, i.e. one effect can be seen as a "main effect" whereas the others are dependent effects (e.g., in our example, owning a correct screw is the main goal of the requester, its delivery is necessary but dependent on this main effect). Furthermore, in contrast to a heuristic which limits the search space without letting the requester influence the limitation, our strategy allows to restrict the search space in a well-defined, user-driven and predictable manner.

After having explained the basic concepts of our approach we will now introduced the extended matching algorithm for multiple connected effects.

6. A MATCHING ALGORITHM FOR MULTIPLE EFFECTS

A structural overview of the extended matching algorithm using the example from Section 2 is shown in Figure 4. To master the complexity of the matching process and to improve efficiency the matching process is performed as a three step process; each step will be discussed in turn.

The Plug-In Match. The first extension to the basic matching procedure for single effects is that of using a plug-in semantics for matchmaking. That is, when we start matching an available offer with a request we only ask whether the offer is capable of fulfilling a subset of the requested effects. The aspect of completely fulfilling a request is delayed until later in the matching process. This approach is motivated by the insight that even if an offer doesn't completely cover a request, it still can be used in a composition of services and consequently shouldn't be dismissed.

To improve efficiency we first ignore all composition related issues and concentrate on reducing the number of offers that need to be regarded during the following composition process as much as possible. This is done by matching offers in isolation. Note that variables cannot be filled correctly without regarding the restrictions related to connections between requested effects (i.e. without regarding issues related to the composition process). By deferring the consideration of connections we reduce code complexity through separating concerns and improve the efficiency as this work isn't done on obviously unsuitable offers.

Thus, as a first step in the matching process (marker A in Figure 4) each offer is examined wrt. whether all of its effects match the requested ones. This corresponds to the above mentioned plug-in match. On the one hand we do not care whether an offer provides all requested effects, on the other hand an offer providing an effect that is not requested is obviously unsuitable[4]. The check whether a single offer

[3]E.g. if no available shipping provider ships outside of Europe, it is assured that no screw factory outside of Europe will be chosen as provider of the screws. This constraint can be assured efficiently, as will be shown in Section 6

[4]We are aware that in many cases an offer that provides more effects than requested might be a match ("We get even more than we asked for"). However, the unrequested effect might just as well be harmful, if it results in buying (and

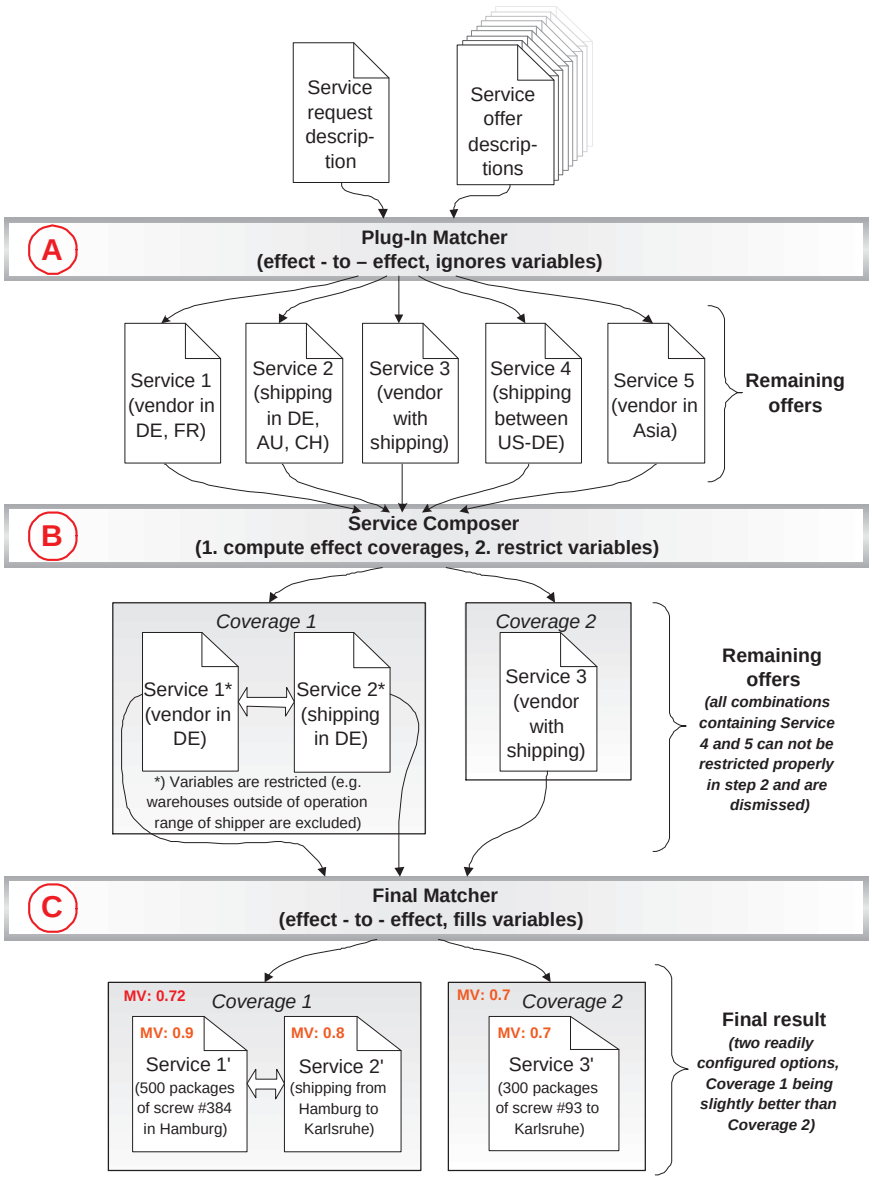

Figure 4: Overview of the Multiple Effect Matching Process described in Section 6

effect matches a certain request effect is done by calling the algorithm for matching single effects from Section 4.

Computing Compositions. After having performed the plug-in match the compositions can be computed (marker B in Figure 4). This involves two tasks. First we compute all possible compositions (called *coverages*) of the offers left by the plug-in match so that every requested effect is covered by an offer exactly once. This can be thought of as decomposing the request's effects into subsets so that for each subset a suitable service provider exists.

After having computed the possible coverages we need to prepare these coverages for the matching with the value propagation semantics (which will form the last step of the matching process). The problem is, that if there is a relation specified in the request like the common location in our example and a corresponding connection is not specified in the offer (for instance because the effects stem from distinct services that are merely composed to yield a coverage of the request), a wrong filling of parameters might prevent a successful matchmaking. Assume we use Coverage 1 with Service 1 having warehouses in Germany and France and Service 2 that ships within Germany, Austria and Switzerland. Assume now we match the vending request effect first and choose a French warehouse (e.g. in Paris) as it is closer to Karlsruhe where we need the screws. When matching the shipping request effect later the coverage will fail since the chosen shipper does not service France. If we had chosen Hamburg as the warehouse's location in the first place the composition would have been suitable. There are two possible ways to address this problem. On the one hand, we could implement some kind of backtracking that tries to

paying for) something we didn't want. A complete discussion of this issue is beyond the scope of this paper.

re-configure the first effect in case this happens. Unfortunately, this approach has an exponential complexity (which to avoid was the primary task of the value propagation semantics). We take an alternative approach at this point and compute the cut on the parameters involved from the different effects that need to be aligned. Service 1 has warehouses in Germany and France and Service 2 ships within Germany, Austria and Switzerland, thus the cut on both sets yields all German warehouses of Service 1 and restricts Service 2 to their locations. If a coverage yields an empty cut like a composition of Vending Service 5 and Shipping Service 4 in Figure 4 that do not share a common location it will be dismissed. The cut computation can be done efficiently (in the best case by a symbolic comparison (e.g. if the instances in question are numbers), in the worst case by iterating over the possible fillings (e.g. all listed warehouse locations)). After computing the cut we annotate the offer descriptions used in the coverage at hand to restrict the concerned parameters to the instances contained in the cut, i.e. to those instances that are contained in all effects that need to be connected. Thus we remove all warehouse locations outside Germany from the offer description of the screw manufacturer and alter the description of the shipper used in Coverage 1 as if it accepted only addresses in Germany as pickup locations (compare to Figure 4). This will allow us in a final step to fill the parameters (in this case choosing a concrete warehouse) by again solely regarding a single effect at a time but still be safe not to choose a filling that will break the composition in another effect.

Final Result Computation.. As indicated above there are two steps left. We still need to optimally configure each offer and compute the overall matching value of each coverage. Due to the value-propagation semantics and work in the preceding steps this again can be done by considering the effects in isolation according to the order defined by the requester (marker C in Figure 4). For each offer in each coverage, each effect is matched against the corresponding request effect. This is again mainly done using the algorithm introduced in Section 4, this time regarding variables and filling them optimally, thereby configuring the offer. When the request is matched against Coverage 1, the Owned Effect will be matched first. Assume the warehouse in Hamburg will be picked. Consequently the pickup address of the shipping service is set to its address. Overall the resulting configurations of Service 1 and Service 2 might yield matching results of 0.9 and 0.8 respectively, leading to a result of 0.72 for the overall compositions (note that the way how to combine the single results from each effect can be configured by the requestor). Similarly the resulting configuration of Service 3 might yield a result of 0.7 which corresponds to the overall result of Coverage 2. Consequently, the composition of Service 1 and Service 2 will be returned as the best offer[5].

7. EVALUATION

As discussed in Section 1 our integrated approach to service discovery and composition tackles situations where simple discovery looking at a single matching service fails, but a composition of existing services is able to fulfill a goal. This approach has to suit an important requirement: In order not to give up on the idea of dynamic discovery, the composition has to be performed on the fly reacting to a request. Thus, efficiency of the composition algorithm is a central issue. In this section, we evaluate our composition approach with a special focus on this requirement. We will show that its time consumption is well suited for real world deployments and that our approach scales well with the number of available offers.

[5]Actually it is quite debatable how to value a composition compared to an offer from a single provider for various reasons. Generally a composition will contain more potential points of failure during execution. Thus it might be reasonable to add a preference-penalty for compositions depending on the number of services involved.

Component	Complexity
Prematch	$O(n)$
Plug-In Match	$O(\sigma_1 \cdot n \cdot m)$
Computing Compositions	$O((\sigma_1\sigma_2 \cdot n)^{const})$
Result Computation	$O(\sigma_3 \cdot (\sigma_1\sigma_2 n)^{const} \cdot m)$

Table 1: Overview: Complexity of the components

n	Overall number of offered services
m	Mean number of different configurations of an offer effect
σ_1	Selectivity of the pre-matcher, $\sigma_1 \ll 1$
σ_2	Selectivity of the plug-in-matcher, $\sigma_2 < 1$
σ_3	Selectivity of the composition-process, $\sigma_3 < 1$

Table 2: Meaning of the variables.

7.1 Scalability

When it comes to scalability, three parameters have to be discussed: The size of the request description (or number of requested effects), the number of available offers and the number of ways to configure a single offer by choosing different fillings for its input variables.

Regarding the first parameter we feel that it is unrealistic to have requests coverings thousands of effects. A realistic number would be anywhere between one effect for a simple request, e.g. searching a printer or reserving a flight, two effects for a large class of services that involve payment, up to maybe ten effects for more complex tasks. Keep in mind that in order for a request consisting of multiple effects to make sense, these effects need to be related to each other. Otherwise they constitute separate requests.

Thus in the following we treat the number of effects and the size of the request description as constant and address the issue of scalability in the other two parameters by a discussion of our architecture's complexity which is given in Table 1 as an overview. The notation used throughout this discussion is provided in Table 2. The architecture implemented differs from the algorithm introduced in Section 6 in that an additional selective matching run is performed before the actual algorithm starts. This *prematch* processes only those parts of the service descriptions which can be processed extremely efficiently in order to reduce the number of services that need to be considered by the actual matching algorithm. In case of question it acts conservatively, thus it doesn't dismiss a service that might match (recall of 1.0).

The pre-matcher: The pre-matcher considers each offer exactly once and doesn't check for differences regarding variable fillings. Thus the complexity of the pre-matching is in $O(n)$. Despite that, the pre-matcher has a very good selectivity $\sigma_1 \ll 1$. Thus $\sigma_1 \cdot n$, the number of offers that subsequent steps need to regard, is typically by several orders of magnitude smaller than n.

The plug-in-matcher: In the next step, every remaining offer is again considered exactly once but unlike in the prematch all information from the service descriptions is used, in particular the options how to fill input variables are regarded. This results in a complexity of $O(\sigma_1 \cdot n \cdot m)$ where m is the mean number of ways to configure a single offer effect. The number of distinct configurations of an effect depends on the use case but can reach magnitudes of thousands or more (e.g. transportation may be possible to all German cities yielding differing prices). This complexity is paid off by a very precise matching result eliminating all offers that are unsuitable on an effect-to-effect basis (selectivity $\sigma_2 < 1$). As a result the number of offers that have to be considered during the next steps of the matching process will typically range from a couple of services to a few dozend or hundred services at the very most.

Creating Coverages: The Service-Composer determines those combinations that should be considered further. In our current implementation, we simply consider every combination of offers that is suitable for covering the request's effects. This yields a polynomial number of coverages with respect to the available offers: $O((\sigma_1 \cdot \sigma_2 \cdot n)^c)$ (where c is an upper bound for the number of requested effects). After determining the possible coverages, the multi-effect-manager is concerned with a preparation step for filling the parameters, in which the cut on those sets of possible parameter fillings is computed that influence each other due to connections between the corresponding effects. This cut computation can be performed without knowing the precise matching value a single parameter filling will yield and can thus be performed in time linear in the size of the coverage being inspected. It therefore doesn't increase the theoretical complexity bound. Coverages with empty cuts are dismissed leading to a selectivity of σ_3.

Computing the Final Result: In order to compute the final result the exact parameter fillings have to be determined. Thanks to the previous cut computation and the value-propagation semantics discussed in section 5 and 6 this can be done on a single effect basis for each coverage, yielding a complexity bound of $O(\sigma_3 \cdot (\sigma_1\sigma_2 \cdot n)^c \cdot m)$. The computation of the final matching value performed as a last step is linear in the number of remaining coverages and thus uncritical.

Summing up, the time consumption of our approach is dominated by the number of coverages computed on the one hand and the number of possible variable fillings on the other hand. By introducing the value propagation semantics we were able to avoid a combinatorial explosion of the last one.

In the following section we will introduce our experimental results that indicate that for typical scenarios the unavoidable cost of determining the optimal variable fillings still outweights the cost of determining the coverages (i.e. the composition). Thus we expect our approach to be well suited for real world scenarios under the given constraint of dynamic composition.

7.2 Experimental evaluation

In order to test our approach with regard to the requirement of efficient composition, we ran a series of tests with our implementation. We created a set of 11 service offers which were designed according to some real world web services in order to have a realistic complexity of the services. (Although this is not a huge number this is not critical in terms of the evaluation. Only service offers that suit pretty well are considered further after the prematching and the prematching scales extremely well. Thus, even if we had used a much bigger number of offers, the number of offers considered during most of the matching process would not have increased drastically.) The test services were chosen from the domains cinema reservation, flight reservation/booking, hotel reservation/booking as well as combinations thereof representing travel agencies' services. Note that in our modelling the booking of a hotel or flight involves charging a credit card as a further effect.

After establishing the offer descriptions we asked a volunteer to come up with 11 service requests from the domains hotel, flight and cinema. The requests differed in the number of effects ranging from one to four as well as the number of connections which also ranged from one to four. Overall, for the 11 requests there existed 40 different possibilities of matching a request with a subset of the available services such that the requests can be fulfilled. The runtime to compute each of these fourty offer combinations ranged from 0.2s to 2.3s with a mean of less than 1s.

Figure 5 shows the mean shares of the overall runtime of the different matching components. The measurements shown correspond to the 40 different possibilities of matching a request with a composition of offers, but for reasons of presentation similar results have been merged, thus resulting in the ten (instead of fourty) displayed measurements. It can be seen in the figure that each run is dominated by the single-effect-matcher (plug-in match) rather than the multi-effect-manager (creating the compositions) thus underpinning our claim that the composition performed does not compromise the efficiency of the matching as such.

8. RELATED WORK

The most closely related work with regard to discovery is the recently presented WSMO-MX Matchmaker by Kaufer and Klusch [3]. WSMO-MX is a hybrid matchmaker for WSML services that borrows the graph-matching approach from the DSD Matchmaker, but combines it with other concepts developed within other matchmakers which the DSD Matchmaker is lacking. What distinguishes the DSD Matchmaker most from WSMO-MX, as from all other discovery approaches is DSD's concept of precise fine-grained preferences and ranking. Most matchers proposed for OWL-S (see e.g. [11] for a typical example) rely on subsumption matching of inputs and outputs and do not take the effects of the service into account. The matcher proposed recently in [17] additionally matches service product and classification. In contrast, DSD's matching is purely state-based. This has two advantages: First, DSD compares whether a service provides the desired effect or not, second, DSD can abstract from differing interface and find functionally matching services even if their interface differ.

For WSMO, a discovery mechanism that abstracts from the individual effect to a desired, more generic goal is pro-

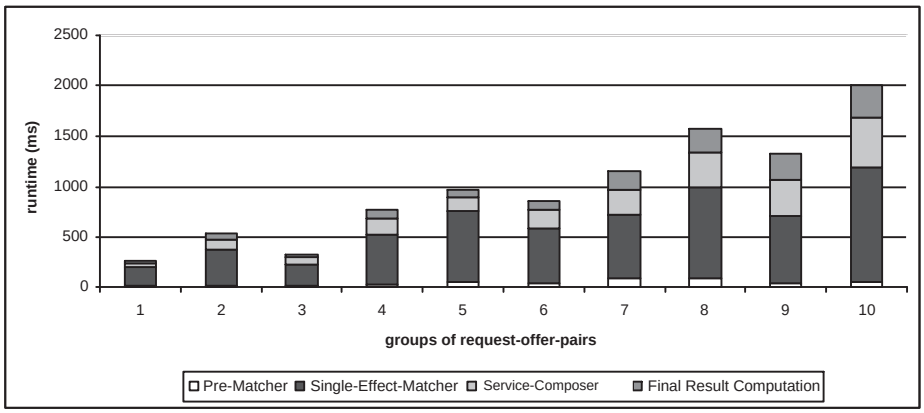

Figure 5: Distribution of the overall runtime among the components for groups of similar results

posed. In [4] the developers argue that this abstraction is necessary due to performance considerations. Likewise, OWL-S [2], e.g., inherits difficulties with reasoning on the instance level from the description logics it is based on. Tools for OWL-S thus tend to not use instance information. In both cases, the effort of determining whether a found service is really able to answer a specific request does not disappear, but is simply shifted to the execution phase.

None of the above mentioned matchmakers integrate automated composition like our approach does. This is probably due to the fact that service composition is usually viewed as a problem separated from service discovery and matchmaking, thus most approaches focus on one of both. This differs from our approach where we use service composition as an integral part of service matchmaking in order to address dynamically on the fly situations where no single service does match a request. One of the main benefits from integrating the service composition into the matchmaking is that we do not only attempt to find some fitting composition but instead are able to find the composition that suits best to the given request's precise preferences. To the best of our knowledge this distinguishes our composition approach from related work.

The METEOR-S framework [18, 1] provides dynamic binding of services, but works with composite service templates and does not attempt to dynamically synthesize service compositions as we do. Although METEOR-S stresses the importance to select component services optimized with regard to certain global optimization criteria like overall monetary cost, it is lacking finegrained user preferences as realized by DSD's fuzzy sets.

Regarding service composition there are many approaches that apply sophisticated AI-Planning techniques to this domain. A comprehensive discussion of these approaches is beyond the scope of this article, see [9, 12] for an overview. The application of the HTN-Planner Shop2 to the web service composition problem by the Mindswap group [16, 10] may serve as an example. Although accomplishing impressive results in terms of the necessary compromise between planning power, completeness and expressivity on the one hand and planning complexity on the other, in [15] Sirin et. al. identify a number of severe problems related to the original approach (e.g. the complete lack of a notion of preference among competing plans). In the same work they extend their approach to address template based compositions (similar as METEOR-S does). They couple Shop2 with an OWL-DL reasoner and are thus able to overcome some of the shortcomings. However, this way complexity issues related to OWL-DL reasoning are introduced to the otherwise very efficient planning process [14]. It remains an open problem whether precise, expressive and yet efficient discovery as realized by the DSD Matchmaker can be efficiently coupled with more general planning as for instance realized by Shop2.

Finally a large number of composition approaches deal mainly with chaining of services. This typically addresses situations where either additional knowledge gathering or some form of data transformation is needed to service a user request with the available offers. An exemplary example for such an approach is [13]. Problems typically addressed by chaining approaches are complementary to the problem of multiple connected effects as dealt with in this work.

9. SUMMARY

In this paper we motivated the need to integrate automated service composition into matchmaking. In particular we dealt with a common class of service requests asking for multiple connected effects that – depending on the service landscape at that time – cannot be serviced by any single one of the available offers. As a first step towards being capable of automatically dealing with such requests, we have considered the problem on a conceptional level. By using a "value propagation" semantics for service descriptions that cover multiple effects we succeed in avoiding exponential complexity for determining an optimal configuration of the offers used for a composition. The basis of our approach to service composition on a practical level is the idea of first using a plug-in matching, i.e. when examining available offers we look whether they are suitable for fulfilling at least a subset of the requested effects. The underlying idea is that if they don't completely cover the request they still can be used as a component in a composition. Our final step towards an efficient composition approach is to compute a cut on possible parameter fillings in separate but connected effects. This enables us to avoid a costly backtracking in the process of filling parameters. We implemented our concepts within the DSD matchmaker and thus integrated automated composition into the matchmaking process. On the one hand

this way we enabled the matchmaker to successfully match a large class of requests it couldn't successfully match previously. On the other hand we could leverage distinguishing features of our matchmaker for the composition process, most notably its sophisticated means of dealing with user preferences to choose among competing alternatives. Thus we are not only able to automatically synthesize a composition but find the best available option with regard to the preferences of the user as specified in its request. In our evaluation we presented a theoretical analysis of the complexity of our approach as well as experimental measurements of the runtime performance. The results emphasize the claim that our approach is scalable and efficient enough to be used for dynamic on the fly matchmaking and composition, even in real world settings.

10. REFERENCES

[1] R. Aggarwal, K. Verma, J. A. Miller, and W. Milnor. Constraint driven web service composition in meteor-s. In *Proceedings of the 2004 IEEE International Conference on Services Computing (SCC 2004)*, Shanghai, China, September 2004.

[2] M. H. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. V. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. P. Sycara. Daml-s: Web service description for the semantic web. In *First International Semantic Web Conference (ISWC2002)*, Sardinia, Italy, June 2002.

[3] F. Kaufer and M. Klusch. WSMO-MX: a logic programming based hybrid service matchmaker. In *Proceedings of the 4th IEEE European Conference on Web Services (ECOWS2006)*, Zürich, Switzerland, December 2006.

[4] U. Keller, R. Lara, A. Polleres, I. Toma, M. Kifer, and D. Fensel. WSMO web service discovery - WSML working draft 12 11 2004. Technical report, DERI, 2004.

[5] M. Klein and B. König-Ries. Coupled signature and specification matching for automatic service binding. In *Proceedings of the European Conference on Web Services (ECOWS 2004)*, Erfurt, Germany, September 2004.

[6] M. Klein and B. König-Ries. Integrating preferences into service requests to automate service usage. In *First AKT Workshop on Semantic Web Services*, Milton Keynes, UK, Dezember 2004.

[7] 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):328–364, 2005.

[8] 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 4th International Conference on Service Oriented Computing (ICSOC06)*, Chicago, Illinois, USA, December 2006.

[9] U. Küster, M. Stern, and B. König-Ries. A classification of issures and approaches in service composition. In *Proceedings of the First International Workshop on Engineering Service Compositions (WESC05)*, Amsterdam, Netherlands, December 2005.

[10] U. Kuter, E. Sirin, B. Parsia, D. Nau, and J. Hendler. Information gathering during planning for web service composition. *Journal of Web Semantics*, 3(2-3):183–205, 2005.

[11] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In I. Horrocks and J. A. Hendler, editors, *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2002.

[12] J. Peer. Web service composition as AI planning - a survey. Technical report, University of St. Gallen, Switzerland, 2005.

[13] S. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for web service composition. In *Proceedings of the 11th International WWW Conference (WWW2002)*, Honolulu, HI, USA, 2002.

[14] E. Sirin and B. Parsia. Planning for semantic web services. In *Semantic Web Services: Preparing to Meet the World of Business Applications, workshop at the Third International Semantic Web Conference (ISWC 2004)*, Hiroshima, Japan, November 2004.

[15] E. Sirin, B. Parsia, and J. Hendler. Template-based composition of semantic web services. In *Aaai fall symposium on agents and the semantic web*, Virginia, USA, November 2005.

[16] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for web service composition using SHOP2. *Journal of Web Semantics*, 1(4):377–396, 2004.

[17] N. Srinivasan, M. Paolucci, and K. Sycara. Semantic web service discovery in the owl-s ide. *hicss*, 6:109b, 2006.

[18] K. Verma, K. Gomadam, A. P. Sheth, J. A. Miller, and Z. Wu. The METEOR-S approach for configuring and executing dynamic web processes. Technical Report 6-24-05, LSDIS Lab, University of Georgia, Athens, Georgia, USA, 2005.