DIANE - A Matchmaking-Centered Framework for Automated Service Discovery, Composition, Binding and Invocation on the Web

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

Michael Klein and Mirco Stern
University Karlsruhe, Germany
kleinm|mirco.stern@ipd.uni-karlsruhe.de

Abstract

One of the visions of service-oriented computing is to allow for the automatic discovery, composition, binding, and invocation of web services. To achieve this goal appropriate matchmaking is the single most important component. In this paper, we present our service description language and the matchmaking algorithms provided for it. Distinguishing characteristics of the language and the algorithms are: The ability to precisely capture requester preferences through fuzzy sets, the ability to express and use instance information for matchmaking, and an efficient approach to dealing with multiple effects. The approach described here has been extensively evaluated both in our own experiments and within the 2006 Semantic Web Services Challenge.

1 Introduction

An important goal of service oriented computing is to enable automatic, dynamic service discovery, binding and invocation at runtime.

Consider as an example an automotive manufacturer with several locally distributed factories. At each site, numerous different parts are needed to construct the cars, most of which are purchased from external component suppliers and delivered to the appropriate sites. The ordering process can be done via web services. Today, this requires considerable human effort: Suitable suppliers are basically manually identified and then statically bound within the application. In order to be able to flexibly react to the market, the manufacturer may want to change this. To achieve this, instead of statically binding to a service provider, the application implementing the order process could contain a semantic description of the required service. An appropriate matchmaking algorithm could then be used to find the best supplier for any given order at runtime. This would result in robust business processes that guarantee optimal choice of supplier for each order without requiring a human to manually adapt anything. The most crucial component of an architecture realizing such a scenario is an appropriate matchmaker meeting at least the following three requirements:

First, we need to be able to describe precisely what a requester wants. This is often neglected in current approaches, where a request is either formulated as a description of the perfect service or more abstractly as a goal that needs to be achieved. This works fine as long as a service is found that matches exactly the request. Often, however, a number of slightly deviating services will be found instead. Current approaches typically leave it to some generic heuristic implemented within the matchmaker to decide which of these services still fits and which one deviates too much and also to decide which of the services is the most desirable. This heuristic may or may not choose the service the requester would have chosen. What we need instead is a possibility for the requester to encode his preferences in the request and a matcher that uses this information. Then, we can guarantee that the correct service is chosen. This is a prerequisite to automation. Otherwise, users will not be willing to accept the results of the matchmaking and will not accept automatic service invocation.

Second, we need to be able to express precisely what a service offers. This will typically not be a single effect but a choice of related effects. For instance, a screw manufacturer will allow you to order different quantities of different types of screws and will deliver them to different locations. This has two implications for offer descriptions: First, services need to be configurable, i.e., the service description needs to contain a specification of possible input parameters. The matchmaking then has to include the appropriate configuration and binding of these parameters. Second, a description will necessarily be somewhat vague. It is unrealistic to assume that the manufacturer would list, e.g., each single type of screw in his service description. Rather, the description will group possible effects. The flip side of this is, that it will probably be impossible to deduce from the description alone whether a certain type of screw is available. To minimize the likelihood of choosing a service that is not able to really provide the desired effect, it would be helpful if some information about instances (e.g., the types really provided) would be made available in addition to the service description. Such instance information plays an important role in a second situation, too: Assume that a service provider expects a product code as input parameter, while a requester has specified material, length and gage as the input she is willing to provide. In this case, too, it would be nice to have instance information available to find an appropriate ANSI code and thus to be able to invoke the service.

Third, in real life settings, it will often be the case that a user request cannot be fulfilled by one service provider alone, but that a combination of providers is needed to achieve the desired goal. Here, it is particularly challenging to find mechanisms that are able to deal with requests that contain several interdependent goals. If we have a request for a quantity of brass and a quantity of copper screws, we can just split the request up into two separate ones, however, if the request is for screws and matching nuts, this will no longer work.

In this paper, we present our service description language DSD (DIANE Service Description) and an efficient matching algorithm for this language and explain how to extend it to deal with multiple effects.

2 A Motivating Example

Let us take a closer look at the car manufacturer. Assume that order management is realized with an application that contains templates of service requests which can be parameterized according to the current needs. One such template could be a request for screws for a certain purpose specifying acceptable materials, types and dimensions. This template could be parameterized with the desired quantity, delivery date and location. As an example, a factory in Karlsruhe could frequently require screws with a countersunk head, preferably crosshead but slotted will be accepted, brass or as an alternative steel and a size of 5x40 mm. The individual screws should be as cheap as possible and cost no more than 35 cents each. On a specific occasion, 10,000 of these screws are needed and 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. Quite obviously, the requester is interested in one out of a number of possible effects, e.g., he doesn't care whether the material is brass or steel, and would prefer crosshead over slotted screws. These preferences and their respective importance need to be encoded in the request in such a way that a matcher is able to determine which of two competing offers the requester prefers. Consider as an example the following suppliers:

- Supplier A offers steel screws with countersunk heads, Phillips crosshead, 30 cents/screw, to be delivered on December 5th.
- Supplier B offers brass screws with countersunk heads, slotted, 31 cents/screw to be delivered on December 8th.

In the following section, we will explain how our language allows a precise capturing of user preferences based on fuzzy sets of effects.

Now, consider a third supplier, Supplier C, which offers screws according to ANSI B18.6.4-1966. With only this information we do not know, whether Supplier C does indeed provide the kind of screw we need. Also, Suppliers A and B will probably offer many different types of screws. It is unlikely, that their service descriptions contain a precise list of types. Rather, they will state that they offer zinc, brass and steel screws with/without countersunk heads in sizes in a certain range. From this description, we do not know whether the precise type of screw needed by us is on offer. We will explain in Section 4 how instance information and knowledge services help with solving this problem. We will then describe in Section 5 how the matcher uses all the information available.

Finally, if you look more closely at the request, you will notice, that it does not contain just one effect that needs to be achieved, but rather two: Once the service has been executed, we want to own the screws and we want to have them delivered to our factory. 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 or combination thereof 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. The goal of our approach is to build an automated matcher that is able to compose such 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 certain constraints within the service descriptions and we need an algorithm that is able to handle requests containing multiple connected effects. Sections 6 and 7 explain how this can be achieved.

3 Supporting Preferences in Service Requests

Any service description language aiming at automating service usage needs to capture a requested functionality as precisely as possible. A fundamental property of most service requests is that a user typically has a notion of a perfect service but accepts services that deviate from this perfect match to a certain degree. This is due to the fact that the probability of a perfect match is generally poor, especially since requests usually contain competing optimization goals (like for instance price versus quality) that need to be balanced. Clearly automated matchmaking that is limited to perfect identity matches will therefore fail in most cases. Thus many approaches to service description languages use a form of subsumption reasoning to compute matches. An offer matches a request if the offer's preconditions are a subset or subconcept of the request's precondition and the request's effects are a subset or subconcept of the offer's effects. Even though this is a major improvement, the similarity distance between an offer and a request is often not computed at all or computed by applying heuristics encoded in the matching algorithm. This is clearly insufficient in real world applications where typically competing optimization goals have to be balanced. Recapture the example from Section 2 where a requester wants to buy certain screws. It is reasonable to prefer a service that offers exactly the same type of screw for a lower price, but how to behave if beside the price some technical properties of the screw, shipping time or warranty conditions differ, too? And how to consider quality of service parameters? Only the service requester can provide means for the necessary fine-grained ranking of potential matches.

In order to still be able to fully automate service usage, we argue that a service request description needs to provide an easy but flexible approach to precisely capture user preferences used to rank potential matches correctly. It is important to design such an approach in a way that does not compromise efficient matching that really makes use of all information given by the requester. In the following we will present our approach to capture user preferences in service request descriptions that was first introduced in [13, 14]. In Section 7 we will show how these preferences are used during service matching.

Let us recap the basic characteristics of request descriptions. The service requester wants to have a certain operation done and does not think inevitably of a certain service. Often, several different services are suitable to fulfill his requirements. Thus, it is not very reasonable to denote one single

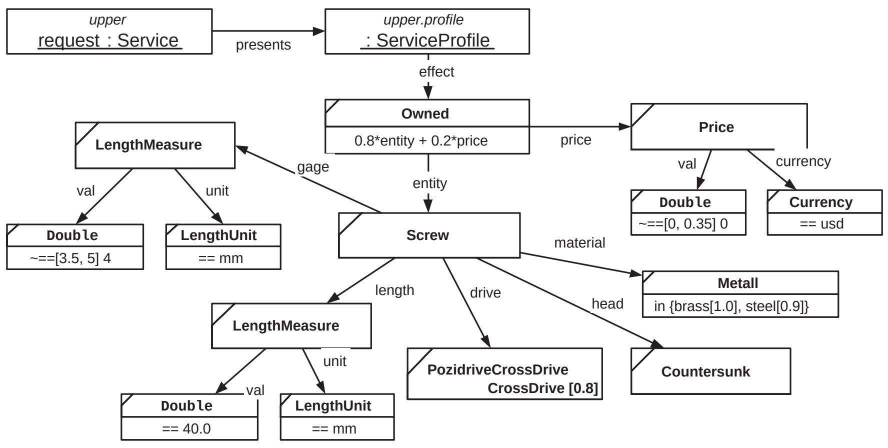


Figure 1: Excerpts from a DSD example request

instance. Additionally, often, the requester will be willing to accept slightly different effects. Thus, it would be more appropriate for the requester to describe a set of services that match his requirements together with preferences stating which members of this set he prefers over which others. Obviously, nothing is gained, if the requester is forced to explicitly write down the individual instances of the set and his preferences for each of them. If, however, the requester would be able to specify *declaratively* the set and the preferences, a giant step towards automatic service binding would be taken.

Therefore, we propose an ontology-based language (DSD) that has been designed regarding this requirement. A service in DSD is mainly described by the *effects* it offers or requests. These are captured as sets which are described either referring to known named instances (like *Karlsruhe*, which is an instance of the city concept) or by declaratively describing the attributes of the concept at hand (like describing the set of towns that are located in Germany and have more than 100.000 inhabitants). This leads to service descriptions as trees as shown in Figure 1 that shows the central aspects of a DSD request corresponding to the example from Section 2. To precisely capture user preferences, fuzzy sets may be used instead of crisp sets in request descriptions [14]. The only difference between fuzzy sets as introduced by Zadeh [28] and our notion of fuzzy set is that we restrict the way these sets may be constructed to using the following five elements:

Fuzzy Type Check Strategies allow the requester to accept different classes of instances in a set with the specified preference. In our screw example the requester accept screws with a Pozidrive crossdrive head with preference of 1.0 and screws with any crossdrive head with a lower preference of 0.8.

Fuzzy Direct Conditions refer directly to the instances contained in a set. These may either be directly listed with their preference value or – in case of DSD's primitive types which correspond to the primitive types from XML Schema – be declaratively described. An example for the first is an expression like `"in {brass[1.0], steel[0.9]}"` that describes slight preference for brass compared to steel, both being named instances from a metal ontology. Examples for the latter are expressions like `"∼==[3.5, 5] 4"` which describes a set of numbers where 4 is the preferred value but values within the range [3.5, 5] are accepted with proportionally decreasing preference.

Fuzzy Property Conditions To support declarative descriptions sets may be described by describing their attribute sets. In Figure 1 the *Screw* set is described by describing the sets of the attributes *gage*, *length*, *drive*, *head* and *material*[1]. Instances are member of the Screw set if their attributes are members in the corresponding attribute sets of the Screw set.

Fuzzy Connecting Strategies control how membership values derived from fuzzy property conditions are combined to compute the membership value of the superordinate element. By default the membership value of the attribute values are multiplied, but a number of custom functions like minimum, maximum or weighted sum may be used to customize this semantic. In Figure 1 a weighted sum is specified in the Owned set declaration: `"0.8*entity + 0.2*price"`. Thus the membership value of an instance to the Owned set is determined by combining the membership values of it's entity and price properties as specified, thus emphasizing the type of screw to be purchased at the expense of its price.

Fuzzy Missing Strategies encode what to do when an offer simply doesn't provide some type of information that appears in the request. Based on the requester's knowledge about the real world the user may decide to ignore missing information or to assume to a certain degree that a requirement will be met by a service not specifying it.

This may seem a bit complicated at first glance. But keep in mind that the requester here is not the end user but some application developer or specialist. These conditions are specified once for each template. The end user then simply instantiates this template with his parameters. The higher the fuzzy membership value (normalized to $[0, 1]$) of an instance in a set is,

[1]A number of other attributes have been omitted for the sake of simplicity.

the higher the preference of the user for this particular instance.

The expressivity of the preference language has been deliberately restricted compared to more complete proposals like [10] to allow only for the expression of preferences that can be locally evaluated. This is necessary in order to ensure efficient matchmaking.

4 Configurable Instance-Based Service Offers

While service requests are characterized by the need to express preferences among various potential matches, a central characteristic of service offers is that a service usually provides many possible similar effects. A screw manufacturer for instance does not only sell one particular screw but offers a huge variety of articles. Usually it doesn't make sense to create one offer description for each of the potentially offered products (i.e. for each possible effect). First, most providers will not be willing to publish complete information about all of their products to a public repository. Second, availability of items will usually change rather dynamically, thus resulting in a huge number of updates to the repository, which is usually not feasible. Sometimes it is even impossible to create a single offer description for each possible effect: A shipping service can impossibly publish descriptions for all available pair of shipping locations. Thus one wants to create offer descriptions that capture all or at least a group of many similar effects at once. Nevertheless such an offer description must describe the available effects as precise as possible. This has two major consequences.

First, although an offer typically offers many effects (A single screw manufacturer sells many different screws), a requester or an automated matchmaker needs to be able to choose the particular desired effect for the actual invocation. This is realized in DSD via *variables* which are a special type of sets. Offers may specify that elements of their descriptions are variables (e.g. the size and material property of a screw set). These need to be filled by the requester prior to the service invocation, thereby choosing the particular desired effect. By using variables, offers become configurable.

Second, if a screw manufacturer were to precisely describe each product available within its offer description, that description would become unmaintainable big. Furthermore much of the necessary information about a particular screw is stable publicly available domain knowledge. This information can be shared. This can be done by using the concept of *instances*[2].

Generally, in order to obtain machine-understandable and processable service descriptions, an understanding about the structure and the semantics of these descriptions is needed. Both can be achieved by the use of a commonly agreed vocabulary in form of *ontologies*. When modeling the knowledge about a domain of the real world in the form of an ontology, one typically distinguishes between a domain schema providing structuring information and instances of the concepts defined in the schema.

The schema alone does not suffice to adequately describe services, since services typically refer to individual *instances* and change the state associated with these instances. In our example, the delivery service will change the location state of the screws to Karlsruhe (an instance of the "location" concept). If a description language does not allow to use instances, we can only describe that the location state of the screws will be changed. At this level of abstraction it wouldn't be possible to distinguish a transport company operating in Germany from one delivering in Korea only. If this were the case, one would need to determine at service execution time whether a service provider is really able to fulfill a request. The result of lacking instance information is a shift of effort from the discovery to the execution phase. To avoid this, DSD offers the possibility to use instance information.

Clearly, what is needed are means to access distributed, maybe private instance information. In DIANE this information access is modeled as a specific kind of service, a *knowledge service*. Our matcher determines that the service description might match depending on the available instances and uses a knowledge service in order to compute the final match value.

This service allows to declaratively query for needed instances, i.e., the requester is able to restrict the set of returned instances by specifying constraints. In our example, it should be possible to determine which types of screws a certain provider offers that match the requirements specified earlier (e.g., brass or steel, countersunk head etc.). Such a knowledge service could be provided by the provider of the original service. Such a provision would be beneficial to the service provider since its two-step approach allows for a precise description of the offered service without the need for a huge service description specifying each individual screw. The matcher retrieves

[2]Screws can be identified by their ANSI type and the properties of a particular screw, e.g., its dimensions as identified by its ANSI type are the same, regardless of the supplier offering it. Thus it is sufficient to encode this information once and reuse it in the form of domain knowledge.

the original service description, determines that it might match depending on the available instances, and then - and only then - accesses the instance provisioning service to compute the final match value.

5 The Basic Matching Algorithm

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 in order to be able to really determine how well an offer fits a request. In DSD this configuration is done by choosing and assigning concrete values to all variables in an offer description. Only this way the best matching service can be invoked automatically later. 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" which can be thought of as a real number normalized to the interval [0,1]. The basic problem of matching a DSD offer o against a DSD request r can 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 [12, 13, 14].

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 as explained in Section 3. The matching value of an inner node is basically computed by combining the matching values of it's child nodes using the specified connection strategy. DSD's fuzzy set based rating features 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 [14] for details) but it is important to mention that DSD has been designed in a way that allows the matcher to largely 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[3] to cover multiple connected effects also. We will illustrate this issue in the next section.

6 Handling Multiple Effects: Foundations

At first glance it seems straightforward to extend the procedure explained above from handling single effect services to also cover multiple effects, but on closer inspection some problems arise. These problems are mainly due to the fact that multiple effect services usually contain relations between effects. Recall our example. 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.

6.1 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,

[3]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.

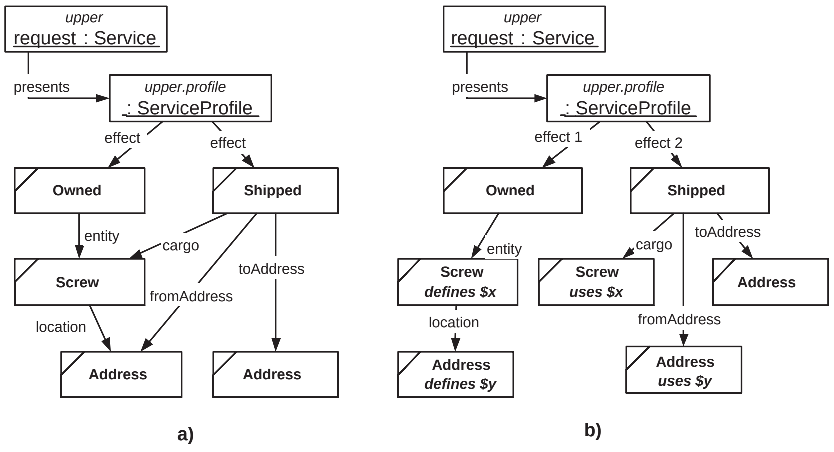


Figure 2: Intuitive approach for connected effects (a) and Alternative *value propagation semantics* (b)

this is important, e.g., 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 a) 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 descriptions 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 can't 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, a number of problems arise. 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 offers (including their various different configurations) that have to be compared against the requested effect i. This 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 depend on the departure and destination of the route.

6.2 "Value Propagation" - Semantics

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 2 b). 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[4]. 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 the offer descriptions we plan to use to service a request in a way that prevents to configure one offer in a way that precludes to use the other offers for the other effects. This can be done in linear time in the number of configuration options as will be explained in Section 7. 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.

[4]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 7

7 A Matching Algorithm for Multiple Effects

A structural overview of the extended matching algorithm using the example from Section 2 is shown in Figure 3. 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.

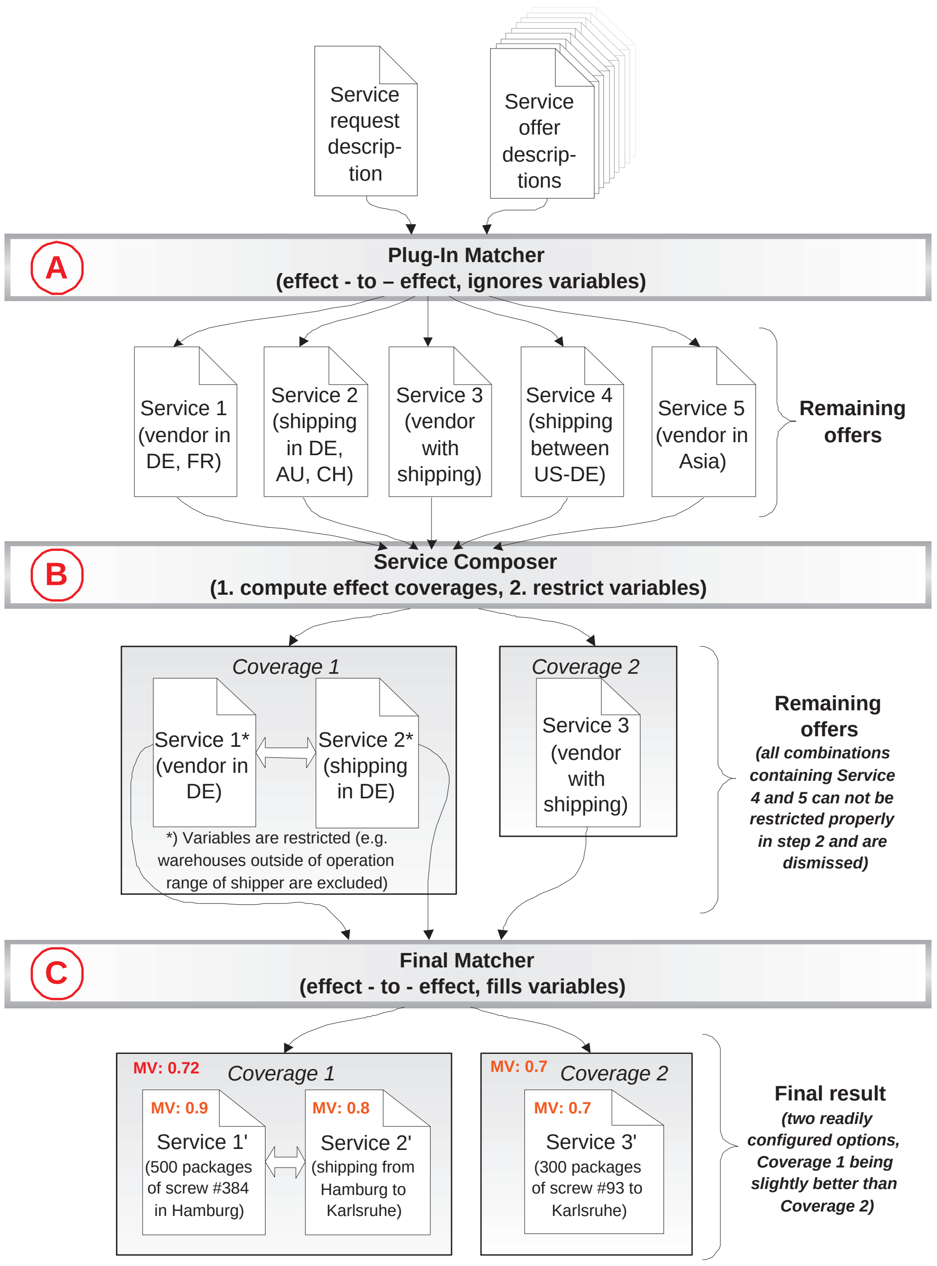
The Plug-In Match The first extension to the basic matching procedure for single effects is that of using a *plug-in* semantics for matchmaking. That is, when we start matching an available offer with a request we only ask whether the offer is capable of fulfilling a *subset* of the requested effects. 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 3) 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[5]. The check whether a single offer effect matches a certain request effect is done by calling the algorithm for matching single effects from Section 5.

Computing Compositions After having performed the plug-in match the compositions can be computed (marker B in Figure 3). This involves two

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



Service request descrip-tion
Service offer descrip-tions
A
Plug-In Matcher
(effect - to – effect, ignores variables)
Service 1 (vendor in DE, FR)
Service 2 (shipping in DE, AU, CH)
Service 3 (vendor with shipping)
Service 4 (shipping between US-DE)
Service 5 (vendor in Asia)
Remaining offers
B
Service Composer
(1. compute effect coverages, 2. restrict variables)
Coverage 1
Service 1* (vendor in DE)
Service 2* (shipping in DE)
*) Variables are restricted (e.g. warehouses outside of operation range of shipper are excluded)
Coverage 2
Service 3 (vendor with shipping)
Remaining offers
(all combinations containing Service 4 and 5 can not be restricted properly in step 2 and are dismissed)
C
Final Matcher
(effect - to - effect, fills variables)
MV: 0.72
Coverage 1
MV: 0.9
Service 1' (500 packages of screw #384 in Hamburg)
MV: 0.8
Service 2' (shipping from Hamburg to Karlsruhe)
MV: 0.7
Coverage 2
MV: 0.7
Service 3' (300 packages of screw #93 to Karlsruhe)
Final result
(two readily configured options, Coverage 1 being slightly better than Coverage 2)

Figure 3: Overview of the Multiple Effect Matching Process described in Section 7

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 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 3 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 alter 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 3). 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 3). 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 5, 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 resprectively, 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[6].

8 Evaluation

Unfortunately, there is no unified evaluation framework or test bed for semantic discovery, matchmaking and composition. The IEEE Web Service Challenge[7][4] focussed on purely syntactic matchmaking and composition based on matching WSDL part names. This year semantics were added in

[6]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.

[7]http://www.ws-challenge.org/

form of inheritance relationship between parts as implemented using complex XML Schema types, but this type of semantics is still very low level and thus the evaluation provided by the contest is not applicable to our framework. More recently the Semantic Web Service Challenge 2006[8][20] defined two realistic problem sets, one concerned with mediation aspects, the other one with semantic service discovery. The challenge aims at developing a common understanding of various technologies used to automate mediation, choreography and discovery using semantic annotations and intents to explore the trade-offs among them. It is the most comprehensive and independent evaluation test bed for such approaches we know of. At the last SWS-Challenge workshop in Budva, Montenegro, DIANE has presented the most complete solution to the discovery scenario and was the only approach able to solve the mediation as well as the discovery scenario with the same technology[15]. Recently, plans to create a SWS Matchmaker Contest were announced on the mailing list of W3C's Semantic Web Services Interest Group[9]. The contest will focus on the retrieval performance of WSMO and OWL-S matchmakers and it will be interesting to compare the performance of the DSD Matcher with other matchmakers using the same suite of test cases. Unfortunately, the contest is still in the process of planning and no test bed is available yet.

Therefore, apart from the evaluation within the SWS Challenge, we created our own evaluation scenarios. The aims of this evaluation were threefold: We wanted to show that DSD matches individual services efficiently (i.e., with reasonable effort) and effectively (i.e., produces the match results users would expect) and that the multi effect matching is efficient. In the following subsections we will look at each of these issues in turn.

8.1 Evaluation of matching efficiency

We have implemented the approach described in this paper and thereby provide a proof-of-concept that the matching algorithm is indeed implementable. We have tested the implementation with a number of test cases answering a number of different questions. For the tests, 500 requests were generated. Also, for each request 30 more or less suitable offers (with a match value greater than 0) were generated. We measured the time it took the matcher to perform a single match with these test sets. Depending on the size of the

[8]http://sws-challenge.org/

[9]http://lists.w3.org/Archives/Public/public-sws-ig/2006Aug/0038.html

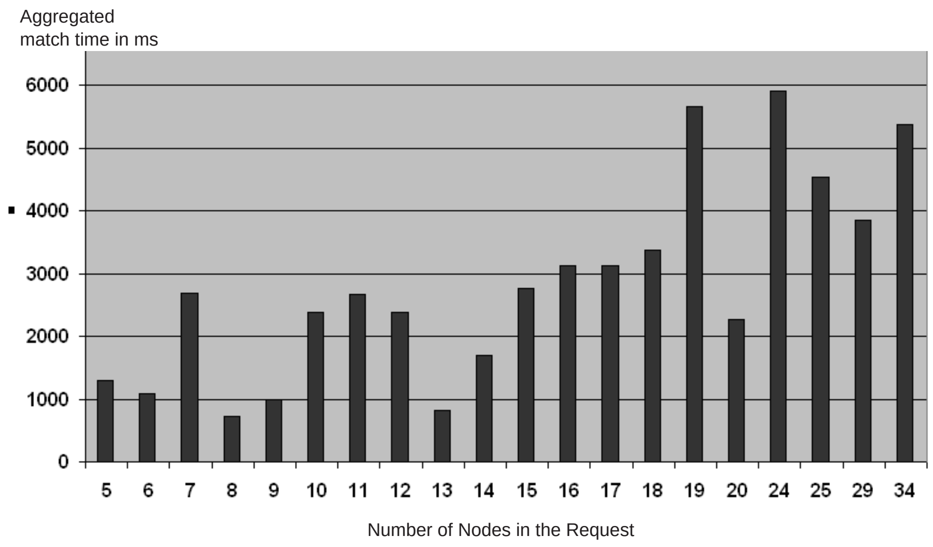


Figure 4: Overall runtime depending on the size of the request

request (the number of nodes in the query graph), this time varied between 0.9 and 75 ms.

A figure that is more interesting than how long a single comparison takes is how long it takes the matcher to compare a request with all offers present, since this is the time a requester needs to wait for an answer. While in the experiment above, requests were compared only to offers where we knew beforehand that the matching value would be greater than 0, now, we assume a mix of matching and non-matching offers. Each request was compared to 1000 offers, 1% of which matched, while the other 99% were selected randomly. Figure 4 shows the evaluation results for this case. The x-axis depicts the number of nodes in the request, the y-axis the aggregated match time. Since the elimination of non-matching offers occurs early on in the matching process, these offers have little influence on the overall time requirement. The comparison of a request with a non-matching offer takes on average 2.5 ms. This results in overall run times ranging between a little less than a second to a maximum of six seconds. We believe that this is an acceptable response time. Also, one should keep in mind that the implementation was done as a proof-of-concept implementation with little attention paid to performance.

8.2 Evaluating the Effectiveness of the Matching

While it is nice to know that the matcher works efficiently, it is as important to ensure that it produces the results expected by the users. To evaluate the effectiveness of our approach, we tried to answer two questions: First, is it possible to express real-life requests using DSD? Second, when these requests are matched, do the results match the expectations of the users?

To answer the first question, we ran an experiment where we asked users with no knowledge of DSD to come up with service requests for a book buying and a travel scenario. For the travel scenario we distinguished between end user requests and requests that would typically be embedded into an application. The latter are particularly important as this is the type of request DSD was developed for. Beforehand, DSD experts designed appropriate domain ontolgies. The experts then tried to translate these requests into DSD using the given ontologies. During the experiment, a total of 195 requests were produced[10] and analyzed. Depending on the ease of translation, the requests were grouped into 3 categories:

- *green*: The request can be expressed completely in DSD using the provided ontologies.
- *yellow*: The request can be expressed completely in DSD, however, the domain ontologies needed to be adapted.
- *red*: The service could not be described because DSD was not expressive enough.

Figure 5 shows the results for different types of requests. Since DSD's focus is on application requests, these results are very promising.

Finally, we checked whether offers and requests that were developed independently were correctly matched. These tests resulted in a precision of 1.0 and a recall of 0.7. If you look at the matching algorithm, it becomes obvious, why such a result is to be expected: The matcher matches conservatively, i.e., if in doubt, it discards an offer. This ensures that no false positives occur, but may result in discarding of suitable offers. Overall, we can be sure that if a service is chosen, it will be one whose effect the requester really wants.

[10]A complete list of these requests as well as results of the evaluation can be found here: http://hnsp.inf-bb.uni-jena.de/DIANE/benchmark. Unfortunately, this is as of today in German, an English version will be made available soon.

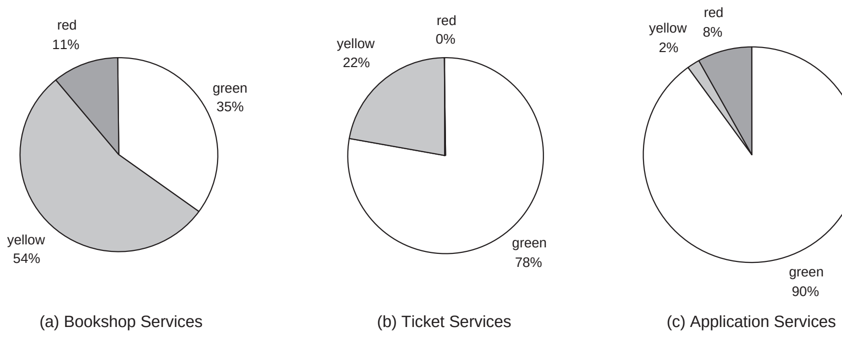


Figure 5: Ability to express requests

8.3 Evaluating Multi-Effect Matching

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.

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

Component	Complexity	Meaning of variables	
Prematch	$O(n)$	n:	Overall number of offers
Plug-In Match	$O(\sigma_1 \cdot n \cdot m)$	m:	Mean number of different configurations of an offer effect
Computing Compositions	$O((\sigma_1\sigma_2 \cdot n)^{const})$	σ_1:	Selectivity of the pre-matcher, $\sigma_1 \ll 1$
Result Computation	$O(\sigma_3 \cdot (\sigma_1\sigma_2 n)^{const} \cdot m)$	σ_2:	Selectivity of the plug-in-matcher, $\sigma_2 \leq 1$
		σ_3:	Selectivity of the composition computation, $\sigma_3 \leq 1$

Table 1: Overview: Complexity of the components

other two parameters by a discussion of our architecture's complexity which is given in Table 1 as an overview. The architecture implemented differs from the algorithm introduced in Section 7 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.

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 significantly 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 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. 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 dozen or hundred services at the very most.

Creating Coverages: The Multi-Effect-Manager 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 6 and 7 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.

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 com-

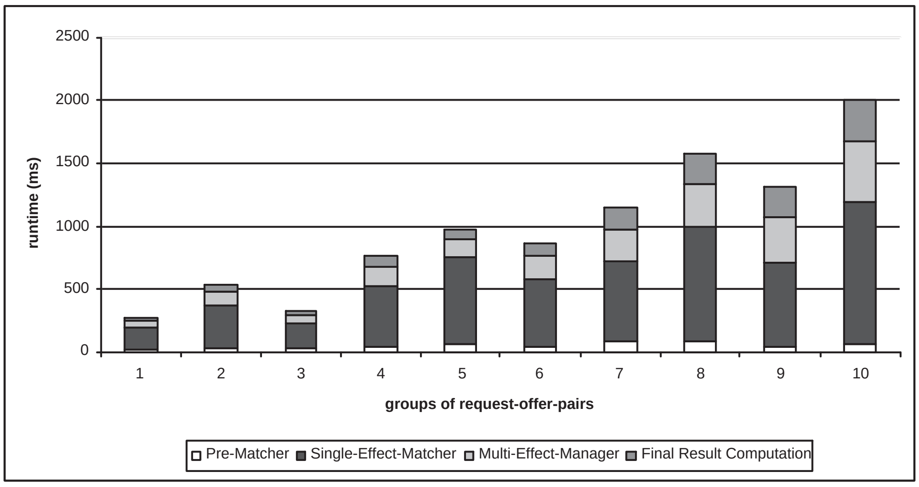


Figure 6: Distribution of the overall runtime among the components for groups of similar requests

plexity of the services. *Remember that we have shown above that the number of services offered is not critical in terms of performance. Thus, a relatively small number of services was sufficient for the evaluation.* These 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 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 0.203s to 2.078s with a mean of 0.953s.

Figure 6 shows the mean shares of the overall runtime of the different matching components grouped by similar requests. Those requests contained in group 1 to 5 each yield a single matching composition while the remaining groups yield 2, 4, 5, 4, and 4 compositions per request, respectively. Also, the groups differ in the number of effects per request, as described above.

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.

9 Formal Basis and Expressivity of DSD

For service description, typically highly expressive description logic languages are used in order to be able to capture both the application domain of the service as well as the service's functionality within that domain. As service matchmaking has to rely on the standard reasoning techniques of these languages, very complex and intractable matching algorithms arise. Often, they are only able to consider parts of the complete service descriptions, which leads to poor results.

Therefore, for DSD a new ontology language (*DIANE Elements*, short *DE*) has been created by combining concepts from frame languages, description logics, fuzzy logic and modal logic. The major design decision was to start from a simple language and extend it exactly by that functionality that is needed to describe the characteristics of services. Because of that, DE is divided in two parts: (1) a simple frame-based ontology language to describe the domain (called *DE-I*) and (2) a small set of additional logic based extensions that are exclusively needed to capture a service's functionality (called *DE-II*).

DE-I is based on typical object-oriented concepts like inheritance, associations, primitive data types, enumerated data types and a separation of instances and classes. However, DE-I also contains meta-attributes which can be attached to certain definitions, e.g. in order to differentiate between public and private classes or to distinguish defining and derived attributes.

Within DE-II, additional constructors for the service-specific part of the description are put on top of DE-I. The most important one is the constructor for a declarative set, which can be compared with the constructor for complex concepts in description logic. As these sets can be built up using equivalents to atomic negation, intersection, value restriction, limited extension restriction and nominals, their core expressivity can be given as $\mathcal{ALO}^{(D)}$. However, in order to fully express the characteristics of services, DE-II has been extended by the notion of fuzziness (i.e. elements can gradually belong to the set), configurability (i.e. the members of a set can be adjusted via variables), dependency (i.e. the elements in one set can influence the elements of another set like it is used in the value propagation) and the notion of being an “effective” set (i.e. whether its members are able to alter the world’s state)[11]. Thus, the expressivity of DE cannot be directly compared with existing description logics or frame based logics. However, the existence of a matching algorithms that operates efficiently in practical situations shows the advantage of such a service description specific ontology language.

10 Related Work

The most closely related work with regard to discovery is the WSMO-MX Matchmaker [8]. 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 most other discovery approaches is DSD’s concept of precise fine-grained preferences and ranking. Most matchers proposed for OWL-S (see e.g. [18] for a typical example) rely on the subsumption matching of inputs and outputs described above and do not take the effects of the service into account. The matcher proposed recently in [25] additionally matches service product and classification. In contrast, DSD’s matching is purely state-based. This has two advantages: First, the DSD Matchmaker 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 proposed. In [9] the developers argue that this abstraction is necessary due to performance considerations. Likewise, OWL-S [5], 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.

[7] is similar to our work in its explicit acknowledgment of the need for

[11] A formal definition of the exact semantics of the constructors in DE can be found in [11] where an axiomatic semantics of DE is given by mapping the language elements onto expressions from FOL.

offer descriptions to represent numerous variants of a concrete service. It extensively discusses its implications on service matchmaking using description logics inference. However, most problems identified by [7] are avoided by our approach since our matcher not only passively compares an offer with a request but – where allowed by the offer – actively configures the offer at hand to maximize the match value. This is a feature that distinguishes our work from all description logic based approaches.

SMART [2] is one of the few approaches that also use fuzziness to express user preferences in service requests. The authors do so by leveraging previous work on modeling fuzzy rules with description logics [1]. A single rule specifies the discrete preference level of a certain combination of attribute values. On the one hand this is a promising approach since a foundation based on description logics integrates nicely with OWL-S and WSMO and can benefit from existing reasoning and verification tools for these logics. On the other hand it remains unclear whether this approach can be implemented efficiently while we provided an efficient implementation for our approach. Additionally our approach that works on continuous numbers supports more finegrained preferences with less modeling effort. Furthermore our matcher supports optimal configuration of an offer whereas [2] has no notion of offer configuration. Compared to [2] Straccia [26] recently presented a more expressive fuzzy variant of $\mathcal{SHOIN}$, the description logic commonly used for the semantic web. The comments regarding description logic based service matchmaking made above apply to this work, also. Besides that, the increased expressivity naturally comes at the price of computational complexity. In contrast, DSD has been designed with a rather limited complexity to maintain efficient processability.

A work that is close to ours in the way it integrates matchmaking and composition has been presented in [21]. Ragone et al. build on a new inference problem – *concept abduction* which is an extension of subsumption – defined for description logics and used for resource matchmaking in [6]. Intuitively – given concepts S (supply) and D (demand) – abduction refers to finding a concept H such that $S \sqcap H \sqsubseteq D$. Thus, a minimal H conforms to the restrictions missing in S in order for S implying D. Ragone et al. present a sound algorithm that applies this inference to the web service composition problem. Intuitively S conforms to a service offer that partially covers the request D (like our screw provider), contraction is used to identify H, the part of the request that is still uncovered (like the shipping of the screws) thus allowing to iteratively try to cover H with another offer. While this is a very interesting approach, it is not suitable (and not intended) for fully automated service usage as is DIANE. First and foremost it does not regard user preferences and does not aim at finding an optimal solution. Beside that, description logic subsumption is problematic when used in a fully automated way for service matchmaking. A complete discussion of this topic is beyond the scope of this paper, but basically subsumption is a too selective check if configurability of the offer is not taken into account (see also [7]).

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, 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 [27, 3] provides dynamic binding of services, but works with composite service templates and does not attempt to dynamically synthesize service compositions as we do. The same is true for the SELF-SERV framework [29]. Both deal with selecting component services optimized with regard to certain global optimization criteria like overall monetary cost, but are lacking finegrained user preferences as realized by DSD's fuzzy sets. More important, the selection of component services is performed using linear programming techniques. This is possible since they assume independent service communities (in the terminology of [29]) of interchangeable services. In contrast, our approach acknowledges that certain combinations of services may be incompatible (like a screw provider located in Taiwan and a European shipper) and is able to compose services of different granularity. Given a request for purchasing, shipping and insurance of the shipping we could compose shipping-only services, purchasing-only services, insurance-only services, purchasing+shipping services, shipping+insurance services and purchasing+shipping+insurance services. This does not map to the service communities used in [29].

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 [16, 19] for an overview. The application of the HTN-Planner Shop2 to the web service composition problem by the Mindswap group [24, 17] 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 [23] 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 [22]. 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.

11 Summary

Precise, reliable and efficient matchmaking is the key requirement when trying to achieve automatic service discovery, composition, binding, and invocation. In this paper, we have presented our service description language, DSD, that has been designed particularly with the requirements of matchmaking in mind and the matchmaking algorithms developed for DSD. Distinguishing characteristics of the language and the algorithms are: The ability to precisely capture requester preferences through fuzzy sets, the ability to express and use instance information for matchmaking, and an efficient approach to dealing with multiple effects. The approach described here has been extensively evaluated both in our own experiments and within the 2006 Semantic Web Services Challenge.

In our ongoing work we study further enhancements to DSD to support different types of automatic service composition as well as the integration of non-functional properties in the matching process. Additionally, we plan to take a closer look at possible mappings between other service description languages, in particular WSML, and DSD. On a more general note, we are interested in the development of benchmarks for the evaluation of semantic web services frameworks. Here, the SWS Challenge offers a good starting point, but more work is needed.

References

[1] S. Agarwal and P. Hitzler. Modeling fuzzy rules with description logics. In *Proceedings of Workshop on OWL: Experiences and Directions*, Galway, Ireland, November 2005.

[2] S. Agarwal and S. Lamparter. Smart - a semantic matchmaking portal for electronic markets. In *Proceedings of 7th IEEE International Conference on E-Commerce Technology (CEC 2005)*, München, Germany, July 2005.

[3] 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.

[4] M. B. Blake, K. C. Tsui, and A. Wombacher. The EEE-05 challenge: A new web service discovery and composition competition. *eee*, 00:780–781, 2005.

[5] 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.

[6] S. Colucci, T. D. Noia, E. D. Sciascio, F. M. Donini, and M. Mongiello. Concept abduction and contraction for semantic-based discovery of matches and negotiation spaces in an e-marketplace. *Electronic Commerce Research and Applications*, 4(4), 2005.

[7] S. Grimm, B. Motik, and C. Preist. Variance in e-business service discovery. In *Proceedings of the ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications*, Hiroshima, Japan, November 2004.

[8] 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.

[9] 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.

[10] W. Kießling. Foundations of preferences in database systems. In *VLDB*, pages 311–322. Morgan Kaufmann, 2002.

[11] M. Klein. *Automatisierung dienstorientierten Rechnens durch semantische Dienstbeschreibungen (in German)*. PhD thesis, Friedrich-Schiller-Universität Jena, Jena, Germany, February 2006.

[12] 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.

[13] 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.

[14] 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.

[15] U. Küster, B. König-Ries, and M. 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.

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

[17] 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.

[18] 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.

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

[20] C. Petrie. It's the programming, stupid. *IEEE Internet Computing*, 10(3):96, 95, 2006.

[21] A. Ragone, T. D. Noia, E. D. Sciascio, F. M. Donini, and S. Colucci. Fully automated web services orchestration in a resource retrieval scenario. In *2005 IEEE International Conference on Web Services (ICWS2005)*, Orlando, FL, USA, July 2005.

[22] 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.

[23] 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.

[24] 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.

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

[26] U. Straccia. A fuzzy description logic for the semantic web. In E. Sanchez, editor, *Fuzzy Logic and the Semantic Web*, Capturing Intelligence, pages 73–90. Elsevier, 2006.

[27] 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.

[28] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8, 1965.

[29] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *Proceedings of the Twelfth International World Wide Web Conference (WWW2003)*, pages 411–421, Budapest, Hungary, May 2003.