Efficient Discovery of Collision-Free Service Combinations

Roman Vaculín¹, Katia Sycara²

¹ Institute of Computer Science, Academy of Sciences of the Czech Republic (vaculin@cs.cas.cz)
² The Robotics Institute, Carnegie Mellon University (katia@cs.cmu.edu)

Abstract

Majority of service discovery research considers only primitive services as a suitable match for a given query while service combinations are not allowed. However, many realistic queries cannot be matched by individual services and only a combination of several services can satisfy such queries. Allowing service combinations or proper compositions of primitive services as a valid match introduces problems such as unwanted side-effects (i.e., producing an effect that is not requested), effect duplications (i.e., producing some effect more than once) and contradictory effects (i.e., producing both an effect and its negation). Also the ranking of matched services has to be reconsidered for service combinations. In this paper, we address all the mentioned issues and present a matchmaking algorithm for retrieval of the best top k collision-free service combinations satisfying a given query.

1 Introduction

Traditionally, matchmaking algorithms [11, 8, 3] used in service discovery components consider only one service as a suitable candidate satisfying a service request while combinations or compositions of services are not considered. This is motivated by requirements service discovery components need to fulfill: service registries are expected to store large numbers of services and at the same time the best matching set of services for a given query has to be retrieved in a timely manner (ideally, in order of milliseconds). Such a combination of requirements makes it difficult to employ full-fledged composition algorithms [13, 1, 10] during the discovery process simply because their time complexity is unacceptable for the discovery purposes (composition algorithms usually assume a much smaller number of services and operate in order of seconds or minutes). In this paper, we focus on addressing this problem by enriching discovery algorithms with basic composition capabilities in a controlled manner that guarantees (1) the same flexibility as the one of classical matchmaking of individual services and (2) a modest time complexity increase compared to individual services matchmaking.

Specifically, we modify the matchmaking conditions for individual services as introduced by Paolucci et. al. in [11] to allow a combination of several services as an acceptable match for a given service request. This has to be done carefully though, since allowing combinations of services can lead to efficiency problems, as identified in [4] where Benatallah et. al. show that finding an optimal combination of services covering the request can be NP-hard under certain conditions. We explore a similar direction by allowing the combination of services satisfying the request to be returned as a relevant match — we call it a combined match. While a combined match addresses the situation when a single service matching a given request does not exist, it also introduces new problems. Since one combined match can consist of several services that together are able to satisfy the service request, various collisions between those matching services have to be reconsidered. For example, one single effect (e.g., a booked plane ticket) might be delivered by more than one service in the combined match leading to an undesirable situation. Similarly, undesired side-effects can be produced by the combined match — e.g., if a flight reservation is provided only in a package with a hotel reservation, the hotel reservation might be an unwanted side-effect for a requester who needs a flight ticket only. Finally, two services can produce contradictory effects (such as making and canceling a reservation). Depending on the requester’s needs each of these situations might cause a problem and thus making a combined match useless. Therefore, the discovery component has to be able to avoid collision in matches.

Retrieving all matching service combinations can be a problem because of the possibly big size of such a set. To deal with this problem we devise an algorithm for re-
trivial of the best top \( k \) matching combinations with respect to an aggregate ranking function that computes the overall matching degree of service combinations. We show, that if the overall ranking function is monotonic and monotonic in all its parameters, the retrieval of top \( k \) service combinations without undesired and contradictory effects can be performed with the time complexity \( O((m \log m) \cdot n) \) worse than the time complexity of the individual service retrieval for a request with \( n \) outputs or effects, with \( m \) being the maximum number of advertisements able to produce some output or effect in the request. We also show that retrieving service combinations without effect duplications is \( \mathsf{NP} \)-hard.

The contributions of this paper are the following. We give a formal characterizations of the combined match and of the possible collisions that need to be avoided building mainly on OWL-S concepts of inputs, outputs, pre-conditions and effects. We discuss the problem of aggregate ranking function for combinations of services. Further, we present an efficient matching algorithm to support a collision-free combined match and we show that under many circumstances the combined match can be computed in about the same time as the basic individual service match (with an exception of the case where effect duplications need to be avoided). We do not consider computation of full-fledged service compositions (employing chaining of services), and we address the service matchmaking on the types level only while not considering instance (data) level matchmaking.

The rest of the paper is structured as follows. In Section 2, we motivate the problem by an example from the process mediation domain. Section 3 introduces the basic terminology which is followed by definitions of matching conditions of individual services and of a combined match in Section 4. In Section 5 we introduce possible collisions and the ranking of a combined match. Section 6 introduces the matching algorithm and its properties. In Sections 7 and 8 we discuss the related work and conclusions.

## 2 Motivating Example

We demonstrate the matching sets of services on the problem of process mediation where it arises very naturally. In the process mediation, the goal is to achieve interoperability of two or more possibly incompatible processes. Figure 1 presents an example of the mediation problem between a hypothetical requester (Figure 1a) and a provider (Figure 1b) from the flights booking domain.

The requester’s process starts with the Login atomic process that has two inputs, ?userId\(^1\) which is an instance of the UserID class and ?password of Password type, one output ?logResult of boolean type and a conditional effect expressing that the predicate LoggedIn(userId) will become true if the value of ?logResult equals to true. Similarly the process continues by executing other atomic processes. Input and output types used in processes refer to a simple ontology showed in Figure 2.

Dashed arrows between parts (a) and (b) of Figure 1 represent symbolically possible mappings between requester’s and provider’s processes. Often the identified data incompatibilities or missing pieces of information require external services to be used in order to construct a meaningful mapping between processes. Consider, for example, the requester’s SearchFlight atomic process and the provider’s SearchFlightOne process. The requester has the inputs \( I_{\text{SearchFlight}} = \{ (?from, FromCity), (?to, ToCity), (?depTime, USDepTime), (?retTime, USRetTime) \} \), while the provider expects the inputs \( I_{\text{SearchFlightOne}} = \{ (?from, AirportFromCode), (?to, AirportToCode), (?depTime, ISODepTime), (?retTime, ISORetTime) \} \). The mapping cannot be constructed directly, since the input types of processes do not match. Differences between \( I_{\text{SearchFlight}} \) and \( I_{\text{SearchFlightOne}} \) define an information gap that can be used to construct a query for the discovery service. An ideal service which would bridge the identified gap has to consume \( I_{\text{SearchFlight}} \) as its inputs and produce \( I_{\text{SearchFlightOne}} \) as its outputs. It is very unlikely that there would ever exist one single service satisfy such a requirement. If combinations of services are allowed to be matched, the chances of a successful match are much higher. In our particular case, a combination of external services AirportCityToCode and USTimeToISO can be used as a match bridging the gap as shown in Figure 1. After the discovery service returns such a set of services as a valid match, the process mediation component can use a composition algorithm to construct the sought mapping by employing the newly discovered services.

---

\(^{1}\) In our notation, variable names are distinguished by a question mark.
3 Preliminaries

When publishing service capabilities to the discovery service, a service advertisement is used, while a service request is used when searching for a service providing required capabilities. For purposes of this paper we consider only inputs, outputs, preconditions and effects (IOPEs) in service advertisements. A service advertisement A is a tuple \( A = (I, O, P, E) \), where I and O are sets of typed input and output parameters of the advertised service, i.e., \( I, O = \{ (?v, T) | ?v \in Var, T \in Types \} \), and P and E are sets of preconditions and effects respectively. \( Var \) is a set of input and output names (we assume each input and output name to be unique) and \( Types \) is a set of types (either primitive XSD types or OWL classes defined in some ontology). We use \( \text{Var}_I = \{ (?v, \_ ) \in I \} \) and \( \text{Var}_O = \{ (?v, \_ ) \in O \} \) to denote the set of input names and output names respectively. \( \text{Var}_{Free} \) is used to denote free variables which are not used in \( \text{Var}_I \) or \( \text{Var}_O \), i.e. \( \text{Var}_{Free} \cap (\text{Var}_I \cup \text{Var}_O) = \emptyset \).

Similarly, we define a basic service request R as a tuple \( R = (I, O, E) \), where I and O are sets of typed input and output parameters of the requested service, E is a set of required effects. The set of inputs I contains parameters the requester has readily available and wants them to be used by the requested service in order to produce requested outputs O and effects E.

We represent preconditions and effects as expressions in the form of conjunction of description logic atoms enriched with OWL datatypes. An expression is a conjunction of atoms. An atom can be one of the following expressions: \( C(s) \) (concept atom), \( \neg C(s) \) (negation of a concept atom), \( R(s, t) \) (property atom), \( \neg R(s, t) \) (negation of property atom), where \( C \) is an OWL [2] class name, \( R \) is an OWL property (an object property or a datatype property), and \( s \) and \( t \) are variables, individuals or data values. In preconditions variables from \( \text{Var}_I \cup \text{Var}_{Free} \) can be used, while in effects variables from \( \text{Var}_O \cup \text{Var}_{Free} \) are allowed. We assume that the negation is not allowed in the definition of the concept C, and can only be used in the form of a negated concept atom in the expression, i.e. \( \neg C(s) \).

We introduce a notion of the requester’s state in the service request. The purpose of the requester’s state is to provide more information to the matcher about the state of the world as seen by the requester that the requester is willing to share. Specifically, the requester’s state might contain set of valid facts about the world in the time when the request is made, and the set of additional available data which might be possibly used by the requested service. Thus, the requester’s state is a tuple \( S = (F, D) \), where \( F \) is a set of valid facts as seen by the requester, and \( D \) is a set of additional typed data that the requester has available.

We define a state enhanced service request by adding the requester’s state to the basic service request, i.e., the state enhanced service request is a tuple \( R = (I, O, E, S) \), where \( S \) is the requester’s state.

Example: Considering the situation of the requester in Figure 1 for the SearchFlight atomic process, the requester’s state could look like:
\[
S_{SearchFlight} = \{ F = \{ \text{LoggedIn}(\{\text{userId}\}) \},
D = \{ (?\text{userId}, \text{User}\text{ID}), (?\text{password}, \text{Password}),
(\text{logResult}, \text{boolean}), (?\text{sessionId}, \text{Session}\text{ID}) \}\}
\]

The requester’s state \( S_{SearchFlight} \) captures that, as a result of some previous actions, the fact \( \text{LoggedIn}(\{\text{userId}\}) \) is valid, and that some more data, such as \( (?\text{sessionId}, \text{Session}\text{ID}) \), is available to the requester. Such an information can be employed by the matcher to make a more informed match.

4 Beyond Individual Services Matching

In this paper we focus on the so called combined match. A combined match consists of a set of service advertisements which do not depend on each other (i.e., chaining is not allowed). A combined match presents a computationally less demanding alternative to full-fledged service compositions. We derive the precise definition of a combined match from definition of individual service matchmaking.

The matchmaking problem for an individual service can be formulated as follows: given the service request \( R \) and a set of published service advertisements \( \text{Advertisements} \), find the set \( \text{Match} \subseteq \text{Advertisements} \) such that each \( A \in \text{Match} \) represents a service capable of satisfying the request \( R \). Usually (e.g., [11]) the ability of a service to satisfy a request is defined by means of relations between input and output types of advertisements and requests. A service can satisfy a request, if it produces at least all requested outputs and does not need any other inputs than those provided by the requester. Similarly, in terms of effects\(^2\), a service is able to satisfy a request if it can produce such a set of

\(^2\)The original work in [11] does not take effects and preconditions into account.
effects that implies at least all requested effects. Speaking of preconditions, the service should be matched against the request, only when the requester is able to satisfy all preconditions of this services.

The following definition summarizes our short discussion and gives the matching conditions for an individual service including preconditions and effects.

**Definition 1 (IOPEs Matching Conditions for Individual Services):** For a state enhanced request \( R = \langle l_R, o_R, e_R, s_R \rangle \), where \( s_R = \langle f_R, d_R \rangle \), and a service advertisement \( A = \langle i_A, o_A, p_A, e_A \rangle \) the following conditions need to hold for \( A \) to satisfy \( R \):

1. \( \forall (i_r, o_r) \in o_R \exists (i_{o_a}, o_{o_a}) \in o_A \text{ such that } T_{o_a} \approx T_{o_a} \)
2. \( \forall e_r \in e_R \exists e_{a} \in e_A \text{ such that } S_{R} \models e_{a} \Rightarrow e_r \)
3. \( \forall (i_r, T_{o_a}) \in i_A \exists (i_{r}, T_{i}) \in (i_R \cup d_R) \text{ such that } T_{i} \approx T_{i} \)
4. \( \forall p_a \in p_A \text{ } S_{R} \models p_a \)

Relation \( \approx \) defines a match between two types. There is a match between two types \( T_{o_a} \) and \( T_{o_a} \) if one of the following conditions hold:

- An **exact match** is the most preferable, followed by the **plug in match** and **subsume match**. In [11] the matching degree based on the \( \approx \) relation is used for computing the overall degree of match between the request \( R \) and an advertisement \( A \).

Preconditions and effects evaluation in the IOPEs matching conditions relies on knowing the requester’s state \( S_{R} \). In particular, preconditions cannot be properly evaluated without knowing \( S_{R} \) (only tautologies would satisfy condition 4 without knowing \( S_{R} \), which is not very useful). For effects, in case of not knowing \( S_{R} \), we can assume \( S_{R} \) to be empty in which case condition 2 transforms into the form

\[
\forall e_r \in e_R \exists e_{a} \in e_A \text{ such that } \models e_{a} \Rightarrow e_r
\]

This form allows us to derive useful conclusions even without knowing the requester’s state. For example, consider the **Login** atomic service and the **LoginStep2** in Figure 1. **Login** requires the **LoggedIN(\?userID)** effect and the **LoginStep2** produces **LoggedIN(\?user)**. After variables unification condition 2 holds for these two services.

It is straightforward to extend matching conditions for an individual service so that sets of services can be considered as a valid match. A set of advertisements satisfies a service request, if together all advertisements from the set are able to produce outputs and effects required by the requester, while using only inputs specified in the request and while the preconditions of all services hold.

To simplify the notation we first define the notion of an effect implied by an advertisement.

**Definition 2 (Implied Effect):** Let \( S_{R} \) be a requester’s state, \( e \) an effect and \( A = \langle i_A, o_A, p_A, e_A \rangle \) an advertisement. We say that \( e \) is implied by an advertisement \( A \) and write \( S_{R} \models A \Rightarrow e \) if \( \exists e_{a} \in e_A \text{ such that } S_{R} \models e_{a} \Rightarrow e \).

In essence, the combined match is just a set of service advertisements which are able to produce required effects and outputs. Since no service chaining is allowed in the combined match, every advertisement in the combined match can use only inputs provided by the service requester. The same holds for preconditions.

**Definition 3 (Combined Match):** Let \( R = \langle l_R, o_R, e_R, s_R \rangle \) be a state enhanced request, \( S_{R} = \langle f_R, d_R \rangle \). We call a set of service advertisements \( M = \{ A \mid A = \langle i_A, o_A, p_A, e_A \rangle \} \) a combined match satisfying \( R \) if the following conditions hold:

1. \( \forall (i_r, o_r) \in o_R \exists A \in M \exists (i_{o_a}, o_{o_a}) \in o_A \text{ such that } T_{o_a} \approx T_{o_a} \)
2. \( \forall e_r \in e_R \exists A \in M \text{ such that } S_{R} \models A \Rightarrow e_r \)
3. \( \forall A \in M \forall (i_r, T_{o_a}) \in i_A \exists (i_{r}, T_{i}) \in (i_R \cup d_R) \text{ such that } T_{i} \approx T_{i} \)
4. \( \forall A \in M \forall p_a \in p_A \text{ } S_{R} \models p_a \)

Figure 3 depicts a symbolic example of service advertisements forming the combined match. Circles \( A, B \) and \( C \) stand for inputs of the request \( R \). Circles \( X \) and \( Y \) stand for requested outputs and effects. The combined match consists of two advertisements, \( A_0 \) consuming \( A, B \) and producing \( X \), and \( A_1 \), consuming \( C \) and producing \( Y \).

**5 Combined Match: Collisions and Ranking**

By allowing sets of advertisements as a valid match, new issues need to be considered which are typically ignored in case of individual service matching. Specifically, since a combined match can contain several advertisements, effect duplications, contradictory and unwanted side-effects must be taken into account. In this section we formally define the collisions and in the next section we show how the collisions definitions can be employed in the matchmaking algorithm.

**By duplicate effects** we mean a situation when two or more advertised services in a combined match produce the same effect. Such a situation might need to be avoided, since an effect means a change in the real world and we assume that the requester wants to achieve a given effect only once. As an example, imagine a situation when a flight ticket would be booked twice with two different providers.
For outputs, this is generally not such a big problem, since outputs present only new information and the duplication should not matter in most cases. Of course, there might be situations when even outputs duplication is a problem — for example when the requester has to pay for use of each service. Essentially, the problem of duplicate outputs can be solved in a similar fashion as for duplicate effects. The following definition introduces an effects duplicate free combined match.

**Definition 4 (Effects Duplication):** Let \( R = \langle I_R, O_R, E_R, S_R \rangle \) be a state enhanced request and \( M = \{ A \mid A = \langle I_A, O_A, P_A, E_A \rangle \} \) a combined match satisfying \( R \). \( M \) contains no effect duplications if the following holds:

\[
\forall e_r \in E_R \forall A_i, A_j \in M \ ( (S_R \models A_i \Rightarrow e_r) \land (S_R \models A_j \Rightarrow e_r)) \Rightarrow A_i = A_j
\]

Another problem is presented by unwanted side-effects. An unwanted side-effect is such an effect produced by some advertisement which was not required to be produced by the service requester. An example can be a service providing both a flight and a hotel reservation, while the requester wants only a flight reservation.

**Definition 5 (Unwanted Side-Effects):** Let \( R = \langle I_R, O_R, E_R, S_R \rangle \) be a state enhanced request and \( M = \{ A \mid A = \langle I_A, O_A, P_A, E_A \rangle \} \) a combined match satisfying \( R \). \( M \) produces no unwanted side-effects if \( \forall A \in M \forall e_a \in E_A \exists e_r \in E_R \) such that \( S_R \models e_a \Rightarrow e_r \).

Finally, by contradictory effects we refer to a situation when a combined match produces both some effect and its negation as well (e.g. LoggedIn and \( \lnot \)LoggedIn effects). Clearly, this is not acceptable, especially when the particular effect is part of requested effects.

**Definition 6 (Contradictory Effects):** Let \( R = \langle I_R, O_R, E_R, S_R \rangle \) be a state enhanced request and \( M = \{ A \mid A = \langle I_A, O_A, P_A, E_A \rangle \} \) a combined match satisfying \( R \). \( M \) contains no contradictory effects if \( \forall A_i, A_j \in M \forall e \in E_A \exists e_a \in E_A, S_R \not\models A_j \Rightarrow \lnot e_a \).

A ranking of combined matches presents another important topic. Generally speaking, the ranking of matching advertisements has to express how well does a given match satisfy a given request. For example, Paolucci et. al. [11] use the matching degree (as expressed by the \( \approx \) relation in the Definition 1) between individual parameters of an advertisement and a request as a basic criterion. The overall match of the whole advertisement is computed as the worst match of all matching outputs while matching degrees between inputs are used as a secondary matching criterion. A different approach presented more recently by Binder et. al. in [5] proposes a ranking based on a numeric expression provided as part of the request. Essentially, the expressions support a combination of arithmetic operators such as \( \min, \max, +, \cdot, \), etc., and set operators such as \( \text{union}, \) intersection, etc. operating on sets of inputs and outputs of the advertisement and the request. While the approach of Binder et. al. is very flexible and allows a requester to specify a specific type of ranking, the expressions are somewhat non-intuitive.

In our approach, we propose a compromise solution. We use the matching degree between individual parameters of an advertisement and a request as an elementary ranking function. For the elementary ranking function we assume that it returns a value from the numeric interval \( \langle \text{bestMatch, worstMatch} \rangle \) with bestMatch standing for the best match (i.e., an exact match) and the worstMatch standing for the match failed. An elementary ranking function is always applied to the matched pair of parameters (inputs and outputs), effects or preconditions. For inputs and outputs we use the \( \approx \) relation from the Definition 1 as an elementary ranking function. For preconditions and effects we use a simple elementary ranking function returning bestMatch if the effect is satisfied and worstMatch if it cannot be satisfied. We use sets \( m\text{Inputs}, m\text{Outputs}, m\text{Preconditions}, m\text{Effects} \) for sets of pairs of matched inputs, outputs, preconditions and effects respectively. For example, for a request \( R = \langle I_R, O_R, E_R, S_R \rangle \) and a combined match \( M = \{ A \mid A = \langle I_A, O_A, P_A, E_A \rangle \} \) a set \( m\text{Outputs} = \{ \langle \text{out}_R^1, \text{out}_A^1 \rangle, \ldots, \langle \text{out}_R^n, \text{out}_A^n \rangle \} \) such that \( \text{out}_R^i \in O_R, \text{out}_A^i \in O_A, A \in M \) and \( \text{out}_R^i \) is the best match for \( \text{out}_A^i \) (\( i = 1, \ldots, n \)). The other sets are defined in the similar fashion.

To compute an overall ranking for a combined match we use an aggregate ranking function \( \mathcal{A} \mathcal{G} \) which aggregates the values of elementary rankings for pairs from \( m\text{Inputs}, m\text{Outputs}, m\text{Preconditions}, m\text{Effects} \). The only requirement we have for the \( \mathcal{A} \mathcal{G} \) function is that it is monotonic (i.e., non-decreasing or non-increasing) and monotonic in each parameter (i.e., in each pair from \( m\text{Inputs}, m\text{Outputs}, m\text{Preconditions}, m\text{Effects} \)). This constraint allows us to retrieve the best top-k combined matches efficiently as is presented in the next section. Examples of suitable aggregate matching functions include functions such as \( \min, \max, \sum, \text{avg}, \text{etc.} \) For example, the overall matching degree of the work of Paolucci et. al. [11] can be modeled as a minimum over the values of elementary matching function for pairs from the set \( m\text{Outputs} \).

### 6 Matching Algorithm and its Properties

We defined the combined match with efficiency in mind, however, still relatively heavy computations might hinder the efficient processing. A quick analysis reveals that in the worst case the time complexity of retrieval of all combined matches can be exponential in number of effects and outputs of the service request. Let \( R = \langle I_R, O_R, E_R, S_R \rangle \) be a request, assuming \( |O_R \cup E_R| = n \), and let us assume that for each effect or output \( oe \in O_R \cup E_R \) there are \( m \) different ad-
A \in \textit{Advertisements} \text{ that are able to produce } oe. Then there exist \( m^n \) combined matches which are potentially able to satisfy the request \( R \). To avoid such an exponential complexity we exploit the fact that typically the requester wants to retrieve the best match or the best top \( k \) matches (where \( k \) is specified by the requester). Thus instead of retrieving all possible combined matches we focus on an efficient retrieval of top \( k \) combined matches with respect to an aggregate ranking function \( \sigma/\eta \).

The matchmaking algorithm works in two phases: the registration phase and the look-up phase. In the registration phase, a service advertisement is registered with the matchmaker. During this phase the advertisement is saved in the main data structure, \textit{Advertisements}, which is basically a look-up table (an inverted index) in which advertisements together with some auxiliary data structures are stored. Stored advertisements are indexed by classes (types) to support a fast retrieval of advertisements which produce or consume a given class. The advertisements can also be retrieved by using atoms appearing in advertisements effects and preconditions. Advertisements are stored together with precomputed degree of match for each relevant class. In the look-up phase, the \textit{Advertisements} structure is used for finding the set \textit{Match} for a given service request \( R \).

When answering a combined match query (Algorithm 1), the discovery service first finds a set of services that can together produce the required outputs and effects (step 1) (i.e., any service producing some of required outputs or effects is a good candidate). The candidate service advertisements are stored in a structure \textit{PreMatch} which is an array indexed by effects/outputs of the request. Each field in the \textit{PreMatch} array contains top \( k \) advertisements that are able to produce the given effect/output ordered by the elementary degree of match for the given effect/output. The \textit{PreMatch} structure is computed by the procedure \textit{populatePreMatch} presented in Algorithm 2. Essentially, the \textit{populatePreMatch} procedure simply retrieves advertisements for each required output (step 1) and effect (step 2) and adds them into appropriate part of the \textit{PreMatch} structure. In the following steps, out of these candidates, those advertisements that constitute a complete individual service match (as defined in Definition 1) are added to the final results set \textit{Match} (step 4.1), and advertisements with some of their inputs or preconditions missing are deleted from \textit{PreMatch} (steps 4.2 and 4.3). The \textit{populatePreMatch} procedure performs almost the same steps which have to be performed for an individual service match and thus does not introduce any complexity other than maintaining the \textit{PreMatch} structure.

Figure 4 shows an example \textit{PreMatch} structure for a request with three effects/outputs \( e_1, e_2, e_3 \). In each column advertisements producing a corresponding effect are showed. For example, the effect \( e_1 \) can be produced by either of the advertisements \( A, B, F \). The top index at each advertisement stands for the matching degree computed by the elementary matching function for the given advertisement and the corresponding effect/output. Thus, for example, \( A^5 \) in the column of \( e_1 \) means that the advertisement \( A \) matches the effect \( e_1 \) with the degree 5. By analyzing the \textit{PreMatch} structure we can for example derive that combined match consisting of advertisements in the first row, i.e. advertisements \( A, F, E \) is not collision free, since the effect \( e_1 \) is produced twice (by \( A \) and \( F \)), while the combined match consisting only of advertisements \( F \) and \( E \) does not produce duplicate effects and produces all required effects (\( F \) produces \( e_1 \) and \( e_2 \) while \( E \) produces \( e_3 \)). Also the combined match \( \{ F, E \} \) is the best combined match with respect to any aggregate function \( \sigma/\eta \) assuming it is monotonic (non-decreasing in this case) in each parameter. If for example sum is used as an aggregate ranking function the combined match \( \{ F, E \} \) will have the overall ranking equal to 22 \( (8 + 8 + 6) \).

With the \textit{PreMatch} structure computed we need to generate combined matches that do not contain collisions. Since we have constrained the aggregate ranking functions to be monotonic in each parameter we can simply start with the first item in each column of the \textit{PreMatch} structure as the best possible candidate for a combined match and generate the other candidates by traversing the \textit{PreMatch} in the downwards direction (as showed in Figure 4). During the traversing we need to test if each candidate match is collisions free. Algorithm 1 traverses the \textit{PreMatch} structure and maintains the combined match candidates in a priority queue \textit{candidatesQueue} which stores possible combined matches ordered by their overall ranking computed by the \( \sigma/\eta \) function. The queue is initialized with the first possible candidate (step 4) – in case of our example it is the combination \( \{ A^5, F^8, E^6 \} \). In the following steps, the best current candidate is tested for collisions (step 5.2) – \( \{ A^5, F^8, E^6 \} \) produces duplicate effect (\( e_1 \)) – and if it is collisions free it is added to the the final results set \textit{Match}. In step 5.3 the direct successors of the current candidate are generated by the \textit{generateSuccessors} method. In our example the set \( \{ \{ B^6, F^8, E^6 \}, \{ A^5, B^{10}, E^6 \}, \{ A^5, F^8, C^9 \} \} \) is generated for the initial candidate \( \{ A^5, F^8, E^6 \} \). As it can be seen, none of these newly generated candidates is collisions free. Alternatively, the direct collision-free successors can be generated (for the sake of simplicity we present a basic version

<table>
<thead>
<tr>
<th>( O_{R/E_R} )</th>
<th>( e_1 )</th>
<th>( e_2 )</th>
<th>( e_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( A^5 )</td>
<td>( F^8 )</td>
<td>( E^6 )</td>
</tr>
<tr>
<td>2.</td>
<td>( B^6 )</td>
<td>( B^{10} )</td>
<td>( C^9 )</td>
</tr>
<tr>
<td>3.</td>
<td>( F^8 )</td>
<td>( C^{20} )</td>
<td>( D^{21} )</td>
</tr>
</tbody>
</table>

**Figure 4. Example PreMatch structure**
of our algorithm). Finally, in step 5.4 only new candidates are added to the queue.

The implementation of the method `collisionsFree` for testing the collisions is rather straightforward and can be done very efficiently. For avoiding duplicate effects we use the `PreMatch` structure. For testing of unwanted side-effects we are using the pre-classified taxonomy of effect types (classes) which allows us to determine presence of side-effects by simply accessing a hash-set once for each effect in an advertisement. Similarly, to test the presence of contradictory effects in a combined match we maintain a set of positive effects implied by each advertisement and a set of implied negative effects. Again, this allows us to perform the test simply by accessing the hash-set once for each effect in the advertisement.

The presented algorithm always terminates since the

generateSuccessors can generate altogether only a finite number of successors. If the aggregate ranking function $\mathcal{A}$ is monotonic and monotonic in each parameter, Algorithm 1 is guaranteed to return the best top $k$ collision-free combined matches. With respect to the time complexity, we are most interested in the overhead of computing the top $k$ combined matches compared to the top $k$ individual services retrieval. For a request $R = (I_R, O_R, E_R, S_R)$, let $|O_R \cup E_R| = n$, and let us assume that for each $oe \in O_R \cup E_R$ there are at most $m$ advertisements able to produce $oe$. There is an overhead of maintaining the `PreMatch` structure which involves sorting of each vector of individual advertisements for each effect/output of the request, thus the time complexity overhead compared to the individual services retrieval is $O((m \log m) \cdot n)$. Next, the overhead of testing for collisions is linear in both the number of matched advertisements $m$ and in the number of outputs and effects of the request, thus the overhead is $O(m \cdot n)$. Finally, there is an overhead for traversing the `PreMatch` structure and generating the combined matches. If effect duplications are allowed in combined matches the time complexity overhead of the traversal for matches without side-effects and contradictory effects is $O(k \cdot n)$ since no backtracking is necessary to obtain new direct successors and the `generateSuccessors` procedure has to be called exactly $k$ times, each time generating at most $n$ direct successors.

Surprisingly, however, for combined matches without duplicate effects the problem becomes NP-hard. This can be showed by reducing the NP-complete monotone one-in-three satisfiability problem (1-in-3 SAT) [12] to finding a combined match without duplications. The monotone 1-in-3 SAT is a variant of a well known 3-SAT problem where every literal is simply a variable (i.e., no negations) and in each clause exactly one literal has to be true to make the whole formula satisfied. The reduction is straight-forward. Each variable $v$ in the 1-in-3 SAT instance is mapped to a unique service advertisement $A_v$, and each clause is mapped to a unique effect. To each advertisement $A_v$ exactly those effects are added that correspond to clauses in which the variable $v$ appears, i.e., every clause is mapped to one column of the `PreMatch` table. Clearly, a combined match without duplications represents a truth assignment to corresponding variables that satisfy the original 1-in-3 SAT instance.

7 Related Work

Traditionally, the web services discovery field does not primarily focus on matching sets of advertisements. We extend the work of Paolucci et. al. [11]. Recently, Bellur et. al. [3] analyzes the correctness of [11] and suggests an improved algorithm based on bipartite graph matching.

Benatallah et. al. in [4] propose an approach based on
request rewriting that allows a combination of several services to satisfy the service request. The hypergraph theory is used in order to find a combination of web services that best match the given request. The optimality criteria is derived from the notion of covering as much as possible the outputs of the request and requiring as little as possible of inputs that are not provided in the description of the request. The approach of Benatallah et al. is different from our approach in the sense that in the combination match in [4] authors insist on finding an optimal coverage which leads to an NP-hard time complexity.

An approach similar to ours was developed by Küster et al. in [9]. In [9] the authors propose an integrated approach to service matchmaking and composition in the context of the DIANE Service Description language. In principle, the authors extend the matching work of [6, 7] so that combinations of services are possible in order to satisfy requests with multiple effects. There are several differences between our work and [9]. First of all, the DIANE language is quite different from OWL-S, and also the basic matching algorithm differs substantially. For example, the DIANE matching is based on graphs matching and fuzzy sets comparisons while, OWL-S relies mainly on description logic reasoning. In terms of combination based matching, our notion of a combined match is very similar in nature to the concept of multiple effects matching introduced in [9]. Another difference is that [9] et al. consider also matchmaking on the instance level. On the other hand, in our approach we consider possible collisions between service advertisements and we focus on the top \( k \) retrieval.

8 Conclusions and Further Work

In this paper we described an efficient mechanism for matchmaking of sets of services. We defined a combined match and a set of collisions which might make the match problematic for a requester. Our approach presents an extension to the individual service based matching. Such an extension finds its use mostly in dynamic environments in contexts such as service composition or process mediation, in which it is not realistic to assume the perfect knowledge of the environment. Therefore, discovery services need to be used as part of the composition or mediation processes and be able to respond to needs identified within these processes. We argued that introducing matches derived from service combinations increases substantially the likelihood of a successful match. We devised a matchmaking algorithm for a combined match allowing to retrieve top \( k \) collision-free matches. In our future work we will explore possible optimisation techniques for the case when combined matches without duplicate effects need to be retrieved. The problem of the combined match is the fact, that it does not consider proper service compositions. We plan to explore extensions of our work that will allow service chaining and thus support discovery of proper compositions.

References