Towards Optimized Discovery of Service Combinations without Collisions

Vaculín, Roman
2009

Institute of Computer Science
Academy of Sciences of the Czech Republic

Towards Optimized Discovery of Service Combinations without Collisions

Roman Vaculín, Roman Neruda, Katia Sycara

Technical report No. 1061, November 2009

Pod Vodárenskou věží 2, 182 07 Prague 8, phone: +420 266 051 111, fax: +420 286 585 789,
e-mail:ics@cs.cas.cz

Institute of Computer Science
Academy of Sciences of the Czech Republic

Towards Optimized Discovery of Service Combinations without Collisions [1]

Roman Vaculín[2], Roman Neruda, Katia Sycara[3]

Technical report No. 1061, November 2009

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 report, we address all the mentioned issues and present an optimized matchmaking algorithm for retrieval of the best top k collision-free service combinations satisfying a given query.

Keywords:
service discovery, service matchmaking, discovering combinations, semantic web services, optimized discovery

[1]This research was supported by the Czech Ministry of Education project ME08095. Katia Sycara was supported in part by Darpa contract FA865006C7606 and in part by funding from France Telecom.

[2]Roman Vaculín, Roman Neruda – Institute of Computer Science, Academy of Sciences of the Czech Republic ({vaculin,roman}@cs.cas.cz)

[3]Katia Sycara – The Robotics Institute, Carnegie Mellon University (katia@cs.cmu.edu)

1 Introduction

Traditionally, matchmaking algorithms [15, 11, 2] 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 [19, 1, 14] 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 report, 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 [15] 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 [3] 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 taken into account. 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 retrieval 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 $\mathrm{O}((m \log m) \cdot n)$ 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 NP-hard.

The contributions of this report 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, preconditions 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

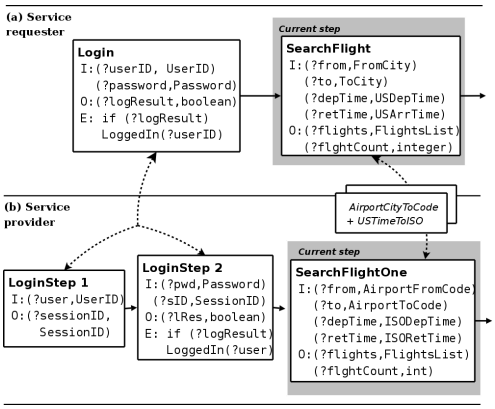

Figure 2.1: Process mediation problem demonstrating the need for a combined match

Concepts	UserID, PasswordCity, FromCity, ToCity, DateTime, USDateTime, USDepTime, USRetTime, ISODateTime, ISODepTime, ISORetTime, FlightsList, FltNr, ItineraryNr, AvailStatus, LoggedIn, AirportCode, AirportFromCode, AirportToCode
ISA relations	USDateTime $\sqsubseteq$ DateTime, USDepTime $\sqsubseteq$ USDateTime, USRetTime $\sqsubseteq$ USDateTime, ISODateTime $\sqsubseteq$ DateTime, ISODepTime $\sqsubseteq$ ISODateTime, ISORetTime $\sqsubseteq$ ISODateTime, FromCity $\sqsubseteq$ City, ToCity $\sqsubseteq$ City, AirportFromCode $\sqsubseteq$ AirportCode, AirportToCode $\sqsubseteq$ AirportCode

Figure 2.2: Fragment of the flights domain ontology with concepts and ISA-relations displayed only

(data) level matchmaking.

The rest of the report is structured as follows. In Section 2, we motivate the problem by an example from the process mediation domain. Section 3 introduces the basic discovery terminology which is followed by definitions of matching conditions of individual services in Section 4. In Section 5, we introduce formally a combined match, effects collisions and we discuss the problem of ranking combined matches. In Sections 6 and 7 we devise the naive and the top-k algorithms for retrieval of combined matches. Section 8 studies the time complexity of the combined matching problem and in Section 9 we introduce optimization techniques of the top-k algorithm. Finally, in Sections 10 and 11 we discuss the related work, conclusions and future directions.

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 2.1 presents an example of the mediation problem between a hypothetical requester (Figure 2.1a) and a provider (Figure 2.1b) from the flights booking domain.

The requester's process starts with the *Login* atomic process that has two inputs, $?userId$[4] 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 2.2.

The interoperability of both process models is hindered by incompatibilities or missing pieces of information. Dashed arrows between parts (a) and (b) of Figure 2.1 represent symbolically

[4]In our notation, variable names are distinguished by a question mark.

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 following inputs

$I_{SearchFlight}$ = { *(?from,FromCity), (?to,ToCity), (?depTime, USDepTime), (?retTime,USRetTime)* },

while the provider expects the inputs

$I_{SearchFlightOne}$ = { *(?from, AirportFromCode), (?to, AirportToCode), (?depTime, ISODepTime), (?retTime, ISORetTime)* }.

The mapping cannot be constructed directly, since the input types of processes do not match according to the ontology defined in Figure 2.2. For example, in the ontology, there is no relationship between the *FromCity* and *AirportFromCode* types, similarly there is no relationship between the *ToCity* and *AirportToCode* types; also no match can be found between *USDepTime* and *ISODepTime*, etc.

Differences between $I_{SearchFlight}$ and $I_{SearchFlightOne}$ 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 $I_{SearchFlight}$ as its inputs and produce $I_{SearchFlightOne}$ as its outputs. Clearly, it is extremely unlikely that there would ever exist one single service satisfying such a requirement. However, 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 2.1. 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.

An important fact to notice is that even in our relatively simple example the need for some kind of match allowing service combinations arises. Such a need is universal for almost any composition scenario which has to be realized in open changing environments.

3 Service Advertisements and Requests

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. In OWL-S a *Service Profile* is used to describe a service for discovery purposes — the Service Profile is used as an advertisement when publishing the service capabilities and as a service request when searching for a service. Most importantly, the Service Profile describes the service from the functional perspective by means of its inputs, outputs, precondition and effects (IOPEs).

For purposes of this report we consider only inputs, outputs, preconditions and effects in service advertisements and requests.

Definition 1 (Service Advertisement)**:** *A* service advertisement A *is a tuple* $A = \langle I, O, P, E \rangle$*, where* I *and* O *are sets of typed input and output parameters of the advertised service, i.e.,* $I = \{(?v, T) \,|\, ?v \in Var, T \in Types\}$*,* $O = \{(?v, T) \,|\, ?v \in Var, T \in Types\}$*, 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 DL concepts defined in some ontology).*

We use $Var_I = \{?v \,|\, (?v, .) \in I\}$ and $Var_O = \{?v \,|\, (?v, .) \in O\}$ to denote the set of input names and output names respectively. Var_{Free} is used to denote free variables which are not used in Var_I or Var_O, i.e. $Var_{Free} \cap (Var_I \cup Var_O) = \emptyset$.

Similarly, we define a *basic service request.*

Definition 2 (Basic Service Request)**:** *A* basic service request R *is a tuple* $R = \langle I, O, E \rangle$*, 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 XSD datatypes. An *expression* is a conjunction of atoms. An *atom* can be one of the following expressions: $C(s)$ (concept atom), $Po(s,t)$ (object property atom), $Pd(s,d)$ (datatype property atom), where C is an OWL class name, Po is an OWL object property, Pd is an OWL datatype property, s and t are variables or OWL individuals and d is a variable or an OWL data value. In preconditions variables from $Var_I \cup Var_{Free}$ can be used, while in effects variables from $Var_I \cup Var_O \cup Var_{Free}$ are allowed.

To better support matchmaking in contexts such as composition and process mediation, 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 matchmaker about the state of the world as seen by the requester. 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 $S = \langle F, D \rangle$, 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. Notice, that the requester's state does not need to describe the state of the requester entirely, on the contrary, typically it can contain only a fragment of the the entire state which the requester finds relevant for the discovery.

We define a *state enhanced service request* by adding the requester's state to the basic service request.

Definition 3 (State Enhanced Services Request)**:** *A* state enhanced service request *is a tuple* $R = \langle I, O, E, S \rangle$, where *where* I *and* O *are sets of typed input and output parameters of the requested service,* E *is a set of required effects, and* S is the requester's state, $S = \langle F, D \rangle$*, 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..*

Example 1: *Considering the situation of the requester in Figure 2.1 for the* SearchFlight *atomic process, the requester's state could look like as follows:*
$S_{SearchFlight} = \langle$
$F = \{LoggedIn(?userId)\}$,
$D = \{(?userId, UserID)$, $(?password, Password)$,
$(?logResult, boolean)$, $(?sessionID,\ SessionID)\}\rangle$

The requester's state $S_{SearchFlight}$ *captures that, as a result of some previous actions, the fact* $LoggedIn(?userId)$ *is valid, and that some more data, such as* $(?sessionID,\ SessionID)$*, is available to the requester. Such an information can be employed by the matchmaker to make a more informed match.*

4 Individual Services Matching

The matchmaking problem for a single service can be formulated as follows: given the service request R and a set of published service advertisements $Advertisements$, find the set $Match$, $Match \subseteq Advertisements$ such that each $A \in Match$ represents a service capable of satisfying the request R. The matching algorithm should be flexible, efficient and should minimize false positives and false negatives. In [15] the ability of 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. The following definition gives a precise characterization of match for a basic service request and the OWL-S service advertisement.

Definition 4 (IOs Matching Conditions for Individual Services)**:** *For a basic request* $R = \langle I_R, O_R, E_R \rangle$ *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. $\forall(?o_r, T_{o_r}) \in O_R \ \exists(?o_a, T_{o_a}) \in O_A$ *such that* $T_{o_r} \approx T_{o_a}$
2. $\forall(?i_a, T_{i_a}) \in I_A \ \exists(?i_r, T_{i_r}) \in I_R$ *such that* $T_{i_a} \approx T_{i_r}$

The matching conditions are defined by means of a relation $\approx$ which defines a match between too types.

Definition 5: *The types* $T_{o_r} \in Types$ *and* $T_{o_a} \in Types$ *are in a relation* $\approx$ *(written as* $T_{o_r} \approx T_{o_a}$*) if one of the following conditions hold:*

1. $T_{o_r} = T_{o_a}$ *or* T_{o_r} *subclassOf* T_{o_a} *(*exact match*)*
2. T_{o_a} *subsumes* T_{o_r} *(*plug in match*)*
3. T_{o_r} *subsumes* T_{o_a} *(*subsume match*)*

There is no match (or match failed) if none of the conditions holds for two types.

The three possible conditions of the $\approx$ relation can be seen as degrees of match with the decreasing quality. An *exact match* is the most preferable, followed by the *plug in* match and *subsume* match. In [15] the characterization of match from Definition 4 is used for computing the overall degree of match between the request R and an advertisement A and also for computing the set $Match$ ordered according to the degree of match.

The original work in [15] does not take effects and preconditions into account. Adding effects is actually quite easy. Naturally, in terms of effects, a service is able to satisfy a given request if it can produce such a set of effects that implies at least all requested effects.

Speaking of preconditions, the advertised service should be matched against the request, only when the requester is able to satisfy all preconditions of this services. Otherwise, it would not make a sense to match a service since the requester would not be able to execute it because of failed preconditions. The problem with preconditions evaluation in the matchmaker is that in order to evaluate the preconditions, the provider has to provide the matchmaker with its state or at least part of it (the requester's state). This might be a problematic issue in many scenarios which maybe explains why service preconditions are very often completely ignored during the discovery.

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

Definition 6 (IOPEs Matching Conditions for Individual Services)**:** *For a state enhanced request* $R = \langle I_R, O_R, E_R, S_R \rangle$*, 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. $\forall(?o_r, T_{o_r}) \in O_R \ \exists(?o_a, T_{o_a}) \in O_A$ *such that* $T_{o_r} \approx T_{o_a}$
2. $\forall e_r \in E_R \ \exists e_a \in E_A$ such that $S_R \models e_a \Rightarrow e_r$
3. $\forall(?i_a, T_{i_a}) \in I_A \ \exists(?i_r, T_{i_r}) \in (I_R \cup D_R)$ *such that* $T_{i_a} \approx T_{i_r}$
4. $\forall p_a \in P_A \ \ S_R \models p_a$

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 the condition 4 without knowing S_R, which is not very useful). For effects, the situation is quite different. In case of not knowing S_R, we can assume S_R to be empty in which case the condition 2 from Definition 6 transforms into the form

2'. $\forall e_r \in E_R \ \exists e_a \in E_A$ such that $\models e_a \Rightarrow e_r$

This form allows us to derive useful conclusions even without knowing the requester's state as we show in Section 6. For example, consider the *Login* atomic service and the *LoginStep2* in Figure 2.1. *Login* requires the $LoggedIn(?userID)$ effect and the *LoginStep2* produces $LoggedIn(?user)$. After variables unification condition 2' holds for these two services.

Knowing the S_R in the matchmaker also allows a slight modification of the condition 3 for inputs compared to the Definition 4 (condition 2). Specifically, the inputs required by the advertised service can either be taken from inputs specified in the I_R of the request (which is a preferable solutions) or to be taken from the set of additional data D_R.

In the following text we will often refer to the match defined in Definition 6 as to the *individual service match.*

5 Combined Match

In this section we formally define a combined match, possible effects collisions and a method for ranking combined matches. It is straightforward to extend matching conditions for an individual service so that sets of services can be considered as a valid match. The idea is to extend matching conditions for a single service so that sets of services can be considered as a valid match. Basically, 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. By allowing sets of advertisements, new issues need to be considered.

First of all, efficiency is a big concern. Discovery components generally, should avoid heavy computations so that a timely response to the requester can be guaranteed. To solve the problem of finding the right balance between the response time and the quality of results, we define the combined match (a less computationally expensive match).

Next, various collisions between effects produced by services contained in the match can make the match unacceptable. Finally, the ranking of sets of services has to be defined so that it is possible to retrieve only the best combined matches. In the following definitions, we define a combined match, while we consider possible collisions and service ranking in the following section.

We assume that the requester's state is available to the matchmaker. Such an assumption might be either unrealistic or undesirable in many scenarios. We consider also situation when the requester's state is not available, however, the definitions with the requester's state assumption and preconditions evaluation included are more general and can be easily modified for situations when only a basic service request is considered.

To simplify the notation a *combined match* definition we first define the notion of an effect implied by an advertisement.

Definition 7 (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 $\exists e_a \in E_A$ 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. In the combined match no service chaining is allowed. Thus, every advertisement in the combined match can use only inputs provided by the service requester. The same holds for preconditions.

Definition 8 (Combined Match)**:** *Let $R = \langle I_R, O_R, E_R, S_R \rangle$ be a state enhanced request, $S_R = \langle F_R, D_R \rangle$. We call a set of service advertisements $\mathsf{M} = \{A \mid A = \langle I_A, O_A, P_A, E_A \rangle\}$ a* combined match *satisfying R if the following conditions hold:*

1. $\forall (?o_r, T_{o_r}) \in O_R \;\; \exists A \in \mathsf{M} \;\; \exists (?o_a, T_{o_a}) \in O_A$ *such that* $T_{o_r} \approx T_{o_a}$

2. $\forall e_r \in E_R \;\; \exists A \in \mathsf{M}$ such that $S_R \models A \Rightarrow e_r$

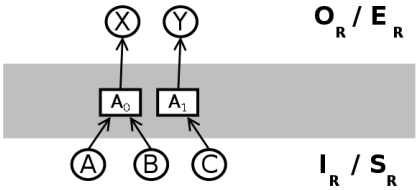

Figure 5.1: Combined Match

3. $\forall A \in \mathsf{M}\ \forall(?i_a, T_{i_a}) \in I_A\ \ \exists(?i_r, T_{i_r}) \in (I_R \cup D_R)$ *such that* $T_{i_a} \approx T_{i_r}$
4. $\forall A \in \mathsf{M}\ \ \forall p_a \in P_A\ \ S_R \models p_a$

All conditions 1–4 are basically just reformulations of conditions from Definition 6 in the context of the combined match.

Figure 5.1 depicts a symbolic example of service advertisements forming the combined match. Circles labeled A, B and C in the bottom part stand for inputs specified in the request R. Circles labeled X and Y in the top part stand for requested outputs and effects. The combined match in Figure 5.1 consists of two advertisements, A_0 consuming A, B and producing X, and A_1 consuming C and producing Y.

Let us mention relations between the combined match and the individual service match. The individual service match can be seen as a special case of the combined match with one advertisement only. This close relations allow us to reuse the theory of the individual service match to compute the combined match.

Finally, we introduce the notion of an incomplete combined match.

Definition 9 (Inclomplete Combined Match)**:** *Let* $R = \langle I_R, O_R, E_R, S_R\rangle$ *be a state enhanced request with* $S_R = \langle F_R, D_R\rangle$*. We call a set of service advertisements* $\mathsf{M} = \{A \mid A = \langle I_A, O_A, P_A, E_A\rangle\}$ *an* incomplete combined match *if all conditions of Definition 8 hold except for conditions 3 and 4, i.e., some inputs required by advertisements in* M *are not provided in the service request* R *or some preconditions are not guaranteed to hold. We define the* incompleteness degree of M *as the number of inputs and preconditions that are not provided or not valid.*

We use definitions of incomplete matches in matching algorithms in the next sections. Oftentimes even an incomplete match might be of use for the requester. This might be the case when the requester does not want to disclose its state. In such a situation it might be possible that the requester has more information available than those pieces of information explicitly specified in the request and thus even an incomplete match could be of use for the requester.

5.1 Effect Collisions

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.

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 services. 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 10 (Effect Duplication)**:** *Let* $R = \langle I_R, O_R, E_R, S_R \rangle$ *be a state enhanced request and* $\mathsf{M} = \{A \mid A = \langle I_A, O_A, P_A, E_A \rangle\}$ *a combined match satisfying* R*.* M *contains no effect duplications if the following holds:*

$\forall e_r \in E_R \;\; \forall A_i, A_j \in \mathsf{M} \quad ((S_R \models A_i \Rightarrow e_r) \wedge (S_R \models A_j \Rightarrow e_r)) \Rightarrow A_i = A_j$

The previous definition guarantees that each required effect is implied by only one advertisement in the combined match.

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

In other words, a combined match does not produce any unwanted side-effects only if all effects produced by advertisements in the match imply some of the requested effects.

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 $\neg LoggedIn$ effects). Clearly, this is not acceptable, especially when the particular effect is part of requested effects.

Definition 12 (Contradictory Effects)**:** *Let* $R = \langle I_R, O_R, E_R, S_R \rangle$ *be a state enhanced request and* $\mathsf{M} = \{A \mid A = \langle I_A, O_A, P_A, E_A \rangle\}$ *a combined match satisfying* R*.* M *contains no contradictory effects if* $\forall A \in \mathsf{M} \;\; \forall e \in E_R \;\; S_R \not\models A \Rightarrow \neg e$*.*

The previous definitions are independent on each other and it depends on the requester if all of the defined collisions are unacceptable or if only some collision needs to be avoided. We discuss in subsequent sections the implications of avoiding each introduced collision in the retrieval algorithm.

5.2 Combined Matches Ranking

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. [15] 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, $+$, $-$, etc., and set operators such as *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 $\langle bestMatch, worstMatch \rangle$ 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 mInputs, mOutputs, mPreconditions, mEffects 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 $\mathsf{M} = \{A \mid A = \langle I_A, O_A, P_A, E_A \rangle\}$ a set mOutputs $= \{\langle out_1^R, out_1^A \rangle, \ldots, \langle out_n^R, out_n^A \rangle\}$ such that $out_i^R \in O_R$, $out_i^A \in O_A$, $A \in \mathsf{M}$ and out_i^A is the best match for out_i^R $(i = 1, \ldots, n)$. The other sets are defined in the similar fashion.

To compute an overall ranking for a combined match we use an *aggregate ranking function* $\mathcal{AG}$ which aggregates the values of elementary rankings for pairs from mInputs, mOutputs, mPreconditions, mEffects. The only requirement we have for the $\mathcal{AG}$ function is that it is monotonic (i.e., non-decreasing or non-increasing) and monotonic in each parameter (i.e., in each pair from mInputs, mOutputs, mPreconditions, mEffects). This constraint allows us to retrieve the 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, avg, etc. For example, the overall matching degree of the work of Paolucci et. al. [15] can be modeled as a minimum over the values of elementary matching function for pairs from the set mOutputs.

6 Naive Algorithm for a Combined Match Retrieval

Definitions of a combined match from the previous section present conditions that service advertisements need to fulfill in order to match a given request. In this section we use these conditions to define the matchmaking algorithms which are used in the discovery service. We defined the combined match with efficiency in mind, however, still relatively heavy computations might hinder the efficient processing. Compared to matching individual services, the main source of additional complexity is the fact that possible combinations have to be considered during the retrieval. 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 advertisements $A \in Advertisements$ 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 can exploit the fact that typically the requester is not interested in retrieving all possible matches, but instead wants to retrieve the best match or the best k matches (where k is specified by the requester). Thus instead of retrieving all possible combined matches we can focus on an efficient retrieval of top k combined matches with respect to an aggregate ranking function $\mathcal{AG}$. In this section we present a naive retrieval algorithm which is derived from the basic logic of the algorithm for the retrieval of individual services, and in the following section we present an algorithm specifically designed for an efficient retrieval of top k combined matches.

The matchmaking algorithm works in two phases. In the registration phase, a service advertisement is registered with the matchmaker. During this phase the advertisement is saved in the main data structure, $Advertisements$, 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. In addition to input and output types, the advertisements can be also 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 $Advertisements$ structure is used for finding the set $Match$ for a given service request R. Algorithm 1 is used for finding the combined match. Since the main computation burden is carried out during the registration phase, the retrieval of matching individual advertisements can be done relatively efficiently.

Algorithm 1 $combinedMatch(R = \langle I_R, O_R, E_R, S_R \rangle)$

1. **for each** $(?o_r, T_{o_r}) \in O_R$
   - 1.1 retrieve all $A = \langle I_A, O_A, P_A, E_A \rangle$ from *Advertisements* for which $\exists (?o_a, T_{o_a}) \in O_A$ such that $T_{o_r} \approx T_{o_a}$ (Def. 8, C. 1)
   - 1.2 add all such A to $PreMatch$
2. **for each** $e_r \in E_R$
   - 2.1 retrieve all $A = \langle I_A, O_A, P_A, E_A \rangle$ from *Advertisements* for which $S_R \models A \Rightarrow e_r$ (Def. 8, C. 2)
   - 2.2 add all such A to $PreMatch$
3. **for each** $A \in PreMatch$
   - 3.1 **if** A is a complete single service match (Def. 6, C. 1–4) **then** add A to $Match$; delete A from $PreMatch$; continue;
   - 3.2 **if** $\exists (?i_a, T_{i_a}) \in I_A \ \ \forall (?i_r, T_{i_r}) \in (I_R \cup D_R)$ such that $T_{i_a} \not\approx T_{i_r}$ (Def. 6, C. 3 fails) **then** mark A as incomplete
   - 3.3 **if** $\exists p_a \in P_A \ \ S_R \not\models p_a$ (Def. 6, C. 4 fails) **then** mark A as incomplete
4. add to $Match$ results of $generateCombined(PreMatch)$
5. **return** $sort(Match)$

When answering a combined match query (Algorithm 1), the discovery service first finds a set of services that together produce the required outputs (step 1) and effects (step 2) (i.e., any service producing some of required outputs or effects is a good candidate). In the next step, out of these candidates, those advertisements that constitute a complete single service match (as defined in Definition 6) are added to the final results set $Match$ (step 3.1). In steps 3.2 and 3.3, the remaining advertisements are possibly labeled as incomplete if some of their inputs or preconditions are missing. As the next step (step 4), all possible combined matches are generated out of the matched candidate advertisements ($PreMatch$) by employing the *generateCombined* procedure. The *generateCombined* procedure is basically quite a simple procedure which uses a brute force algorithm to generate all possible advertisement combinations compliant with the combined match definition. Finally, the resulting matches are sorted according to their degree of match.

The overall degree of match is computed based on the discussion in Section 5.2 which is essentially a conservative extension of the single service matching degree (see [15] for details). The main difference is implied by the fact that the *combinedMatch* can return also incomplete matches. An incomplete match is considered worse than an exact, plugin and subsume match and is penalized accordingly. Still, the incomplete match can be useful. Either it can be used directly by the requester who might try to get required pieces of information from some other source, or it can serve as a starting point to compute the composed match (i.e., a match based on a full fledged service composition).

An interesting thing, is the fact that implementation of the *combinedMatch* procedure required relatively minimal modifications to the single service matching procedure. The most visible modification was addition of the *generateCombined* procedure, which also has the possible biggest impact on the time complexity. We will discuss this issue in more detail in the next section.

If requested, the *combinedMatch* procedure generates only collision free matches. The problem of duplicate effects is tackled in the *generateCombinedMatch* procedure by checking if an advertisement that is being added to a newly generated combined match collides with adver-

tisements already presented in the match. This can be done very efficiently by pre-computing the list of implied effects for each matched advertisement and by testing if any effect of the considered advertisement is not already generated by the constructed combined match. For avoiding 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.

With respect to properties of the naive algorithm, the algorithm always terminates and always returns the list of combined matches ordered by their aggregate ranking. This is a direct consequence of the fact, that the algorithm simply generates all possible combined matches and that the set of combined matches is finite. However, the time complexity is exponential because of the exponential number of all combined matches.

7 Top-k Retrieval Algorithm

Quite clearly the main drawback of the naive algorithm is the fact that to be able to retrieve top k combined matches, all combinations have to be generated and ranked first. Considering that the number of combinations grows exponentially with the number of effects and outputs in requests, such approach can become unacceptable for service requests with many effects and outputs. The top-k algorithm presented in this section does not attempt to generate all possible combined matches, but instead attempts to generate the possible combined matches directly in the ascending order, starting with the best combined match and continuing until top k matches are found. In order to allow such an approach we defined the aggregate ranking function as a monotonic and monotonic in all its parameters which allows the matching algorithm to search the search space starting from the potentially lowest point with the best ranking and continuing towards matches with a worse ranking.

The basic idea of the algorithm is to retrieve advertisements for each requested output or effect separately and for each such an output/effect sort the retrieved advertisements based on the matching degree of the advertisement with respect to the requested output/effect. Having the basic relevant advertisements retrieved and sorted, in the next phase the search space of combined matches built out of these retrieved advertisements is traversed in such a way that guarantees the increasing order of generated combined matches. The following paragraphs describe the algorithm in detail.

Similarly to the naive algorithm, the top-k retrieval algorithm first finds a set of services that can together produce the required outputs and effects (i.e., any service producing some of required outputs or effects is a good candidate). The candidate service advertisements are stored in a structure $PreMatch$. Compared to the naive version where the $PreMatch$ structure was just a list of matching advertisements, in this case we represent the $PreMatch$ structure as an array indexed by effects/outputs of the request. Each field in the $PreMatch$ array contains advertisements that are able to produce the given effect/output ordered by the elementary degree of match for the given effect/output. The $PreMatch$ structure can be formally defined as follows.

Definition 13 (Ordered $PreMatch$)**:** *Let $R = \langle I_R, O_R, E_R, S_R \rangle$ be a state enhanced request, $n = |O_R \cup E_R|$, $oe_i \in O_R \cup E_R$, $i, \ldots, n$. An ordered $PreMatch$ is a tuple $PreMatch = \langle E_1, \ldots, E_n \rangle$, $E_i = \langle A_{i_1}, \ldots, A_{i_{m_i}} \rangle$, such that every advertisement in E_i implies oe_i, i.e. $\forall A_j \in E_i$ $S_R \models A_j \Rightarrow oe_i$, and advertisements in E_i are ordered by the elementary degree of match for the given effect/output oe_i for all $i = 1, \ldots, n$.*

O_R/E_R	e_1	e_2	e_3
1.	A^5	F^8	E^6
2.	B^6	B^{10}	C^9
3.	F^8	C^{20}	D^{21}

Figure 7.1: Example $PreMatch$ structure

Algorithm 2 *populatePreMatch*(oe_r, $PreMatch$), $oe_r \in O_R \cup E_R$

1. **if** $oe_r \in O_R$, $oe_r = (?o_r, T_{o_r})$ **then**
   retrieve advertisements $A = \langle I_A, O_A, P_A, E_A \rangle$ from *Advertisements* for which $\exists (?o_a, T_{o_a}) \in O_A$ such that $T_{o_r} \approx T_{o_a}$ (Def. 8, C. 1)
2. **if** $oe_r \in E_R$ **then** retrieve advertisements $A = \langle I_A, O_A, P_A, E_A \rangle$ from *Advertisements* for which $S_R \models A \Rightarrow oe_r$ (Def. 8, C. 2)
3. add all retrieved A to $PreMatch[oe_r]$
4. **for each** $A \in PreMatch$ **do**
   - 4.1 **if** A is a complete individual service match (Def. 6, C. 1–4) **then** add A to $Match$; delete A from $PreMatch$; **continue**
   - 4.2 **if** $\exists (?i_a, T_{i_a}) \in I_A \;\; \forall (?i_r, T_{i_r}) \in (I_R \cup D_R)$ such that $T_{i_a} \not\approx T_{i_r}$ (Def. 6, C. 3 fails) **then** delete A from $PreMatch$; **continue**
   - 4.3 **if** $\exists p_a \in P_A \;\; S_R \not\models p_a$ (Def. 6, C. 4 fails) **then** delete A from $PreMatch$; **continue**
5. **return** $PreMatch$

Example 2: *Figure 7.1 shows an example 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 ranking function for the given advertisement and the corresponding effect/output*[5]*. Thus, for example, A^5 in the column of e_1 means that the advertisement A matches the effect e_1 with the matching degree 5. By analyzing the 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 $\{F, F, E\}$ is the best combined match with respect to any aggregate function $\mathcal{AG}$ 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 $\{F, F, E\}$ will have the overall ranking equal to 22 $(8 + 8 + 6)$.*

The $PreMatch$ structure is computed by the procedure *populatePreMatch* presented in Algorithm 2. Essentially, the *populatePreMatch* procedure simply retrieves advertisements for each required output (step 1) and effect (step 2) and adds them into appropriate part of the $PreMatch$ structure. In the following steps, out of these candidates, those advertisements that constitute a complete individual service match (as defined in Definition 6) are added to the final results set $Match$ (step 4.1), and advertisements with some of their inputs or preconditions missing are deleted from $PreMatch$ (steps 4.2 and 4.3). Optionally, such matches can be labeled as incomplete in the same fashion as in the *combinedMatch* procedure. The *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 $PreMatch$ structure.

[5]For the sake of clarity we use an imaginary elementary ranking function that assigns positive integers as matching degrees in the example.

With the *PreMatch* structure computed we need to generate combined matches in an increasing order. Also, if requested by the requester, only matches without collisions need to be generated. 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 *PreMatch* structure as with the best possible candidate for a combined match (there cannot be any other combined match with a better overall ranking). The subsequent candidates will be generated by traversing the *PreMatch* in the downwards direction (as showed in Figure 7.1). During the traversing we need to test if each candidate match is collision free. To define the notion of traversing the *PreMatch* structure formally, we have to introduce a definition of a *direct successor* of a combined match with respect to the *PreMatch* structure and a definition of a *index* function of a combined match with respect to the *PreMatch* structure.

Definition 14 (Combined Match Index)**:** *Let* $R = \langle I_R, O_R, E_R, S_R \rangle$ *be a state enhanced request,* $n = |O_R \cup E_R|$*, and* $\mathsf{M} = \{A_i,\ i = 1, \ldots, n\}$ *a combined match satisfying R, and let* $PreMatch = \langle E_1, \ldots, E_n \rangle$*,* $E_i = \langle A_{i_1}, \ldots, A_{i_{m_i}} \rangle$*,* $i = 1, \ldots, n$*.*

We define a function index of a combined match M *in the structure PreMatch as follows:* $index(\mathsf{M}, PreMatch) = \langle k_1, \ldots, k_n \rangle$*, such that* $A_i = A_{i_{k_i}}$*,* $A_i \in \mathsf{M}$*,* $A_{i_{k_i}} \in E_i$*,* $E_i \in PreMatch$ *for every* $i = 1, \ldots, n$*.*

The *index* function of a combined match simply returns the position of the match in *PreMatch* as a tuple of coordinates. For example, given the *PreMatch* structure in Figure 7.1 the index of the match $\{A,\ F,\ E\}$ is $\langle 1, 1, 1 \rangle$, since all advertisements A, F, E are in the position 1 in their corresponding columns. The index of the best collision free combined match $\{F, F, E\}$ is $\langle 3, 1, 1 \rangle$.

Given the *index* function, we can compare combined matches with respect to their indexes in *PreMatch*.

Definition 15 ($<_{PreMatch}$)**:** *Let* $\mathsf{M_r}$ *and* $\mathsf{M_s}$ *be combined matches based on an ordered PreMatch structure for some state enhanced request R, and let index values for* $\mathsf{M_r}$ *and* $\mathsf{M_s}$ *be*

$$index(\mathsf{M_r}, PreMatch) = \langle r_1, \ldots, r_n \rangle$$

$$index(\mathsf{M_s}, PreMatch) = \langle s_1, \ldots, s_n \rangle$$

We write $\mathsf{M_r} <_{preMatch} \mathsf{M_s}$ *if and only if* $\exists i \in \langle 1, n \rangle$ *such that* $r_i < s_i$ *and* $\forall j \in \langle 1, n \rangle\ r_j \leq s_j$*.*

Definition 16 (Direct Successor)**:** *A combined match* $\mathsf{M_s}$ *is a* direct successor *of a combined match* $\mathsf{M_r}$ *with respect to the PreMatch structure if* $\mathsf{M_r} <_{preMatch} \mathsf{M_s}$ and there exists no combined match $\mathsf{M_p}$ such that $\mathsf{M_r} <_{preMatch} \mathsf{M_p} <_{preMatch} \mathsf{M_s}$.

In our example in Figure 7.1 members of the set $\{\{B^6, F^8, E^6\}, \{A^5, B^{10}, E^6\}, \{A^5, F^8, C^9\}\}$ are the only direct successors of the match $\{A^5, F^8, E^6\}$. Notice, that none of these matches is collision free. Similarly to Definition 16 a *collision free direct successor* can be defined by simply requiring the successor to be collision free.

We introduced the notion of a direct successor because of its straight-forward relation to the ordering of combined matches with respect to the aggregate matching function $\mathcal{AG}$. Let us assume that we have found some combined match (e.g., the match $\{A^5, F^8, E^6\}$) and that we are searching for another combined match that would follow right behind the already found one in the totally ordered sequence of matches ordered by the value of $\mathcal{AG}$. Quite clearly, because each column in the *PreMatch* structure is ordered and because the $\mathcal{AG}$ is monotonic and monotonic in all its parameters, the next match has to be one of the direct followers of the already found match. I.e., for the match $\{A^5, F^8, E^6\}$ with the overall ranking equal to 19, the next match is $\{B^6, F^8, E^6\}$ with the overall ranking equal to 20. Thus we can use the direct successors as means of traversing the search space of combined matches in the ascending order.

Algorithm 3 *combinedMatchTopK*($R = \langle I_R, O_R, E_R, S_R \rangle$, k, $\mathcal{AG}$)

1. **for each** $oe_r \in O_R \cup E_R$ **do**
   $populatePreMatch(oe_r, PreMatch)$
2. $candidatesQueue = \emptyset$
3. $firstCandidate = generateFirstCandidate(PreMatch)$
4. $candidatesQueue.add(firstCandidate, \mathcal{AG})$
5. **while** $candidatesQueue \neq \emptyset$ **and** $|Match| < k$ **do**
   5.1 $bestCandidate = candidatesQueue.removeFirst()$
   5.2 **if** $collisionFree(bestCandidate, PreMatch)$ **then** add $bestCandidate$ to $Match$
   5.3 $newCandidates = directSuccessors(bestCandidate, PreMatch)$
   5.4 **for each** $candidate \in newCandidates$ **do**
   if $candidate \notin candidatesQueue$ **then** $candidatesQueue.add(candidate, \mathcal{AG})$
6. **return** $Match$

Algorithm 3 presents the *combinedMatchTopK* procedure. First, it populates the $PreMatch$ structure by calling the *populatePreMatch* in step 1. Next, it traverses the $PreMatch$ structure and maintains the combined match candidates in a priority queue *candidatesQueue* which stores possible combined matches ordered by their overall ranking computed by the $\mathcal{AG}$ function. The queue is initialized with the first possible candidate (step 4) — in case of our example in Figure 7.1 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 collision free it is added to the final results set $Match$. In step 5.3 the direct successors of the current candidate are generated by the *directSuccessors* 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 collision free. Alternatively, the direct collision free successors can be generated (for clarity we present a basic version of our algorithm that generates successors which might contain collisions). Finally, in step 5.4 only new candidates are added to the queue.

The top-k algorithm always terminates since the *directSuccessors* method can generate altogether only a finite number of successors. If the aggregate ranking function $\mathcal{AG}$ is monotonic and monotonic in each parameter, Algorithm 3 is guaranteed to return the best k collision free combined matches.

The implementation of the method *collisionFree* for testing the collisions is rather straightforward and can be done very efficiently. For avoiding duplicate effects we use the $PreMatch$ structure — only one advertisement out of each column can be present in the combined match. Avoiding unwanted side-effects and contradictory effects is addressed in the same fashion as in the naive version of our algorithm.

8 Complexity of the Top-k Algorithm

In this section we discuss properties of the top-k algorithm with the main focus on its time complexity. 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. 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 output or effect $oe \in O_R \cup E_R$ there are at most m different advertisements $A \in Advertisements$ that are able to produce oe. In the following discussion we distinguish two different cases depending on weather duplicate effects (Definition 10) are allowed in the collision free matches or not. We show that if duplicate effects are allowed the complexity overhead is $O((m \log m) \cdot n)$ or $O(m \cdot n^2)$ depending on the specific request, while when the duplicate effects need to be

avoided in combined matches, the problem becomes NP-complete.

We start the discussion with considering the overhead of testing for collisions. For avoiding *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. Therefore for each matched advertisement at most n tests with the constant price have to be performed to guarantee that it has no side-effects. If an advertisement has side-effects it does not have to be considered any more and it does not have to be added into the $PreMatch$ structure. Thus the overall complexity of avoiding side-effects is $\mathrm{O}(m \cdot n^2)$. Notice that to make sure that even a individual service match does not produce side-effects, exactly the same tests have to be performed. Thus, this step does not add any additional time complexity compared to the individual service matching. As a result of the discussion we get the following lemma.

Lemma 1: *Let $R = \langle I_R, O_R, E_R, S_R \rangle$ be a request, $|O_R \cup E_R| = n$, and let m be the maximum number of advertisements $A \in Advertisements$ that are able to produce oe for each $oe \in O_R \cup E_R$. The time complexity of avoiding side-effects in combined matches and in individual service matching is* $\mathrm{O}(m \cdot n^2)$.

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 which can be precomputed in the registration phase. Again, this allows us to perform the test simply by accessing the hash-set once for each effect in the service request, i.e., for each matched advertisement at most n tests with the constant price have to be performed, leading to an overall complexity $\mathrm{O}(m \cdot n^2)$ as in the case of side-effects. If we assume that each registered advertisement is consistent, by which we mean that it does not produce any contradictory effect on its own (i.e., no single advertisement implies both e and $\neg e$ for any effect), the test for contradictory effects does not have to be performed in case of individual service matching. That means the time overhead imposed by avoiding contradictory effects in combined matches is $\mathrm{O}(m \cdot n^2)$. On the on the other hand, if inconsistent advertisements can be registered with the matchmaker, there would be no time overhead since the same testing would have to be performed as well.

Lemma 2: *Let $R = \langle I_R, O_R, E_R, S_R \rangle$ be a request, $|O_R \cup E_R| = n$, and let m be the maximum number of advertisements $A \in Advertisements$ that are able to produce oe for each $oe \in O_R \cup E_R$. The time complexity of avoiding contradictory effects in combined matches is* $\mathrm{O}(m \cdot n^2)$.

In order to generate top-k combined matches, we need to populate and traverse the $PreMatch$ structure. Populating the $PreMatch$ structure involves sorting of each vector of individual advertisements for each effect/output of the request, thus the time complexity overhead is $\mathrm{O}((m \log m) \cdot n)$.

Finally, there is an overhead for traversing the $PreMatch$ structure and generating combined matches. As we pointed out in previous paragraphs, the $PreMatch$ structure does not contain advertisements that are producing contradicting effects and side-effects, if this is required by the requester. This means that while generating combined matches only possible collisions caused by effect duplications need to be taken into account. Let us start with the case when effect duplications are allowed in combined matches. In such a case the *directSuccessors* method always generates at most n direct successors for a given base combined match. Since effect duplications are allowed, each new direct successor is obtained from the base combined match by simply increasing one coordinate of the index of the base combined match. For example, for the base combined match $\{A^5, F^8, E^6\}$ with the index $\langle 1, 1, 1 \rangle$ the combined matches with indexes $\langle 2, 1, 1 \rangle$, $\langle 1, 2, 1 \rangle$, $\langle 1, 1, 2 \rangle$ (i.e., matches $\{B^6, F^8, E^6\}$, $\{A^5, B^{10}, E^6\}$, $\{A^5, F^8, C^9\}$) are the direct successors. Since the *directSuccessors* procedure has to be called exactly k times, the overall time complexity is $\mathrm{O}(k \cdot n)$.

Lemma 3: *Let $R = \langle I_R, O_R, E_R, S_R \rangle$ be a request, $|O_R \cup E_R| = n$, and let m be the maximum number of advertisements $A \in Advertisements$ that are able to produce oe for each oe $\in O_R \cup E_R$. The time overhead of generating combined matches in which effect duplications are allowed is* $\mathrm{O}((m \log m) \cdot n) + \mathrm{O}(k \cdot n)$.

Surprisingly, however, for the case when only combined matches without effect duplications need to be retrieved the problem becomes NP-complete. This can be showed by reducing the NP-complete *monotone one-in-three satisfiability* problem (1-in-3 SAT) [17] to finding a combined match without duplications (CMD problem). We describe the reduction and thus we show that even a problem of finding *any* combined match without duplicate effects is NP-complete. This means that our problem of retrieving top-k instances with respect to the $\mathcal{AG}$ ranking function is also NP-complete.

The monotone 1-in-3 SAT is a variant of a well known 3-SAT problem. In the 1-in-3 SAT problem a formula is a conjunction of clauses with exactly three literals, every literal in a clause is simply a variable (i.e., no negations are allowed in clauses) and in each clause exactly one literal has to be true to make the whole formula satisfied. A solution of a 1-in-3 SAT instance is an assignment of values 1 or 0 to each variable of the formula that makes the formula satisfied.

Definition 17 (1-in-3 SAT [17])**:** *Let $R(x, y, z)$ be a 3-place logical relation with the truth-table being $\{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$, and let $F = C_1 \wedge \cdots \wedge C_l$ be a conjunction of clauses with each clause $C_i = R(x, y, z)$, x,y,z boolean variables, $i = 1, \ldots, l$. The* monotone one-in-three satisfiability problem *is a problem of deciding weather a formula F is satisfiable.*

The following formula presents an example instance of the 1-in-3 SAT problem with three clauses and variables x,y,z,u [17]:

$$R(x, y, z) \wedge R(x, y, u) \wedge R(u, u, y) \tag{8.1}$$

This formula can be satisfied by assigning 1 value to y variable and 0 to the remaining variables and thus this particular instance has a solution.

Theorem 1: *Let $R = \langle I_R, O_R, E_R, S_R \rangle$ be a request, $|O_R \cup E_R| = n$, and let m be the maximum number of advertisements $A \in Advertisements$ that are able to produce oe for each oe $\in O_R \cup E_R$. The problems of finding a combined match without duplications (CMD problem) satisfying a request R is NP-complete.*

Proof. The proof is by reducing the 1-in-3 SAT instance into a CMD instance. 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 C_i is mapped to a unique effect e_i. 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. The table in Figure 8.1 is the result of the transformation of the example formula 8.1 into a corresponding CMD instance:

E_R	e_1	e_2	e_3
	A_x	A_x	A_u
	A_y	A_y	A_u
	A_z	A_u	A_y

Figure 8.1: A CMD instance corresponding to the 1-in-3 SAT instance for formula 8.1

Clearly, a combined match without duplications (e.g., $\{A_y, A_y, A_y\}$) corresponds to a truth assignment to corresponding variables that satisfies the original 1-in-3 SAT instance, i.e., 1 will be assigned to variables whose corresponding advertisements are present in the combined match.

Since the 1-in-3 SAT is NP-complete and since the described transformation is polynomial the CMD problem is NP-complete as well. □

Table in Figure 8.2 summarizes the complexity discussion of the top-k algorithm for retrieving collision free combined matches. We assume an individual services matchmaking algorithm without support for top-k retrieval and for collision avoidance as a base-line. The table shows time overheads of all discussed collision avoidance procedures compared to this base-line in the context of individual service matching and in the context of combined matching. It distinguishes four procedures (each in a separate line): (1) matching with all collisions allowed, (2) unwanted side-effects avoidance procedure, (3) contradictory effects avoidance procedure, and (4) matching when effect duplications are not allowed. The first column of the table summarizes overheads of top-k retrieval of individual services compared to the base-line algorithm (without support for top-k retrieval). For example, the overhead in the case when all collisions are allowed is $\mathrm{O}(m \log m)$ which is caused by the need to order the retrieved service advertisements. When 0 appears in some cell, it means that the corresponding procedure does not have to be performed in the given configuration.

	Individual service matching	**Combined matching**
top-k without collision avoidance	$\mathrm{O}(m \log m)$	$\mathrm{O}((m \log m) \cdot n) + \mathrm{O}(k \cdot n)$
side-effects avoidance	$\mathrm{O}(m \cdot n^2)$	$\mathrm{O}(m \cdot n^2)$
contradictory effects avoidance	0	$\mathrm{O}(m \cdot n^2)$
effect duplications avoidance	0	NP-complete

Figure 8.2: Overview of time complexities of the top-k algorithm for collision free combined matches, compared to individual service matching (for request $R = \langle I_R, O_R, E_R, S_R \rangle$, $|O_R \cup E_R| = n$, for each $oe \in O_R \cup E_R$ at most m advertisements A that imply oe)

9 Dealing with Complexity

In the previous section we analyzed the complexity of the top-k combined match retrieval algorithm. We observed that while in many cases the algorithm is tractable, i.e., polynomial in both the number of outputs/effects in the request and in the number of advertisements, still there is a case of avoiding duplicate effects when the algorithm becomes exponential (NP-complete). In essence, this means that potentially we need to deal with a large search space of possible combined matches and that to find the best k combined matches this search space has to be searched exhaustively to guarantee completeness. In this section we describe the search strategy and the domain independent and domain specific optimization techniques. The optimizations are based on branch and bound pruning and constraints propagation known from constraint satisfaction problem [16, Ch. 5].

Since the problem of finding a combined match without duplicate effects is NP-hard, the search space has to be searched exhaustively. The procedure *combinedMatchTopK* in Algorithm 3 performs an exhaustive search under the assumption that the *directSuccessors* procedure finds *all* direct successors for a given base combined match. In our implementation we use a variation of this method called *directCollisionFreeSuccessors* showed in Algorithm 4 which generates all collision free direct successors of a given base combined match $\mathsf{M}_{\mathsf{base}}$. The *directCollisionFreeSuccessors* procedure uses a depth-first search backtracking as a basic search method which guarantees completeness, i.e. it returns *all* direct collision-free matches for

Algorithm 4 $directCollisionFreeSuccessors(\mathsf{M}_{\mathsf{base}}, PreMatch, Succ, \mathsf{M}_{\mathsf{new}}, depth)$

Inputs: $\mathsf{M}_{\mathsf{base}} = \{A_1, \ldots, A_n\}$ a combined match, $\langle k_1, \ldots, k_n \rangle = index(\mathsf{M}_{\mathsf{base}}, PreMatch)$
$PreMatch = \langle E_1, \ldots, E_n \rangle$, $E_i = \langle A_{i_1}, \ldots, A_{i_{m_i}} \rangle$
$Successors$ a set of found direct successors, initially $Successors = \emptyset$
$\mathsf{M}_{\mathsf{new}}$ a constructed combined match, initially $\mathsf{M}_{\mathsf{new}} = \emptyset$
$depth$ a position (index) in required outputs/effects and in $PreMatch$

1. **if** $depth = n + 1$ **and** $isDirectSuccessor(\mathsf{M}_{\mathsf{new}}, \mathsf{M}_{\mathsf{base}}, Successors, PreMatch)$ **then**
   1.1 add $\mathsf{M}_{\mathsf{new}}$ to $Successors$
   1.2 **return** $Successors$
2. **foreach** $A_j \in E_{depth}$ **such that** $j \geq k_{depth}$ *// index is bigger than base index*
   2.1 **if** $collisionFree(\mathsf{M}_{\mathsf{new}} \cup A_j, PreMatch)$ **then**
   $Successors =$
   $directCollisionFreeSuccessors(\mathsf{M}_{\mathsf{base}}, PreMatch, Successors, \mathsf{M}_{\mathsf{new}} \cup A_j, depth + 1)$
3. **return** $Successors$

a given base combined match $\mathsf{M}_{\mathsf{base}}$. The listing in Algorithm 4 presents only a basic skeleton of the procedure without showing technical details or any optimisation discussed in the further text. The main idea of the procedure is to recurse over all columns in the $PreMatch$ structure and in each recursion level ($depth$) to iterate over advertisements of the current column (E_{depth}, step 2) and to try to add the advertisement to the newly created combined match $\mathsf{M}_{\mathsf{new}}$ if no collision is introduced. After some new combined match $\mathsf{M}_{\mathsf{new}}$ is completed (step 1) it is added to the $Successors$ set only if it is a direct successor of the $\mathsf{M}_{\mathsf{base}}$ match.

To avoid unnecessary bactracking in the $directCollisionFreeSuccessors$ procedure we exploit the following ideas:

- Many branches in the search space do not lead to a result because a particular selection of advertisements producing a subset of required outputs/effects does not allow any other advertisements to be added to the combined match without introducing any effect duplication. To reduce the number of such branches *forward checking* and *constraints propagation* can be used to eliminate such advertisements in the $PreMatch$ which necessarily lead to a failure given a current choice of advertisement in the constructed combined match.
- In Algorithm 4, the columns (E_{depth}) of the $PreMatch$ structure are ordered arbitrarily. Instead it proves to be better to start always with a column which contains the least number of advertisements (i.e., the fewest "legal" values) and thus has the smallest branching factor. This heuristic is known as a *minimum remaining values* heuristic (MRV) or a *fail-first* heuristic [16, Ch. 5].
- During the search process, some branches can be pruned because every possible newly found combined match in that branch would
  - have a too bad overall ranking with respect to the best k matches found so far (*pruning by overall matching degree best estimate*)
  - be a *non-direct* successor of the base combined match $\mathsf{M}_{\mathsf{base}}$ — the matches in the $Successors$ set are used for pruning such branches (*pruning by already found direct successors*)
  - produce a duplicate effect since some previous branch with the same set of available advertisements has failed (*pruning by local caching of failed branches*)

In the following subsections we discuss the introduced optimization techniques.

9.1 Forward Checking and Constraint Propagation

In order to reduce unnecessary bactracking, some advertisements can be eliminated temporarily (or "disabled") given the current state of the search, since they cannot be added to the match without introducing an effect duplication. The problem of finding collision free combined matches can be formulated in terms of a *constraint satisfaction* problem. In constraint satisfaction problems a set of variables each with a domain of possible values is given together with a set of constraints that need to be satisfied. A solution of the problem is such an assignment of values to each variable that satisfies all the constraints. We can see each requested output/effect as a variable, and advertisements able to produce this output/effect as possible values that can be assigned to this variable. For the example problem in Figure 7.1, there are three variables e_1, e_2, e_3 with the domains $\{A, B, F\}$, $\{F, B, C\}$ and $\{E, C, D\}$ in the respective order.

Forward checking is a basic technique that employs constraints during search in order to delete those values from variable domains that are not consistent with constraints relating the current variable with other unassigned variables. Consider for example the $PreMatch$ table in Figure 9.1a. Let us assume that the $direcCollisionFreeSuccessors$ procedure started with the effect e_1 and selected the advertisement A (i.e., A was assigned to e_1). To avoid duplication of the effect e_1, all remaining advertisements in the domain of e_1 can be removed from domains of remaining unassigned variables. Thus, the advertisements B and F can be removed from the domain of e_2 (showed with the gray background in Figure 9.1a). For the same reason in Figure 9.1b the advertisement F can be removed from the domain of e_2 after B is assigned to e_1 in the next iteration. Additionally, the advertisement C can be removed from the domain of e_2 (and subsequently from the domain of e_3 as well) since the assignment of B to e_1 necessitates its assignment to e_2 as well, which leads to removal of C.

O_R/E_R	e_1	e_2	e_3
1.	$\underline{A}$	F	E
2.	B	B	C
3.	F	C	D

(a) A selected for e_1

O_R/E_R	e_1	e_2	e_3
1.	A	F	E
2.	$\underline{B}$	B	C
3.	F	C	D

(b) B selected for e_1

Figure 9.1: Example $PreMatch$ structure after forward checking

At each level of recursion, the forward checking process removes only values from those variable domains that are directly connected to the currently assigned variable (in our examples it is the variable e_1). Although forward checking is efficient and effective, still many values can be inconsistent with some constraints that are not directly related to the current variable. This problem can be addressed by *constraint propagation* techniques which consider also constraints than are not directly related to the current variable. Consider again the example in Figure 9.1a where A is assigned to e_1. During forward checking, no value was removed from the domain of e_3. However, if we take additional constraints into account, e.g., constraints between e_3 and e_2, we figure out that advertisements E and D can be removed from the domain of e_3 (as showed in Figure 9.2 by cells with the dark gray background). The reason is straight-forward. If E is assigned to e_3, the advertisement C needs to be removed from the domain of e_2 (to avoid duplication of e_3). However, after removal of C, the domain of e_2 becomes empty and thus there would be no advertisement available for e_2. Therefore E can be safely removed from the domain of e_3 since it leads to emptying of the e_2 domain. The same logic applies to the advertisement D.

In our implementation we use the arc consistency constraint propagation algorithm (a variation of the AC-3 algorithm [16, Ch. 5.2]). Essentially, the algorithm removes any value from the domain of some unassigned variable x if there exists no value v in the domain of some

O_R/E_R	e_1	e_2	e_3
1.	$\underline{A}$	F	E
2.	B	B	C
3.	F	C	D

Figure 9.2: Example $PreMatch$ structure after forward checking and constraint propagation, A selected for e_1

other unassigned variable y such that the constraint between x, y is satisfied. In the case of the $PreMatch$ structure, a value can be safely removed from some column if after applying forward checking no advertisement will be left in some other column.

Constraint propagation is more powerful than forward checking, however, still it does not remove every inconsistent value and thus there is a need for other optimizations. Since both forward checking and constraint propagation dynamically reduce the number of available advertisements in each column during the search process, it is beneficial to apply the minimum remaining values heuristic (MRV) for selection of the next unassigned variable (column) at each recursion level. The MRV heuristics suggests to select a column which contains the least number of advertisements (i.e., the fewest "legal" values) which constrains the most the available choices and possibly leads to a failure as soon as possible.

9.2 Pruning

There are several reasons why pruning can help to reduce the search space. First, we noticed that even after constraint propagation, there can be branches that do not lead to any combined match without effect duplications. Next, we are not interested in *any* successor of a given base combined match in a particular run of the *directCollisionFreeSuccessors* procedure, but we want to find only *direct* successors. Therefore, every branch that cannot contain direct successors can be pruned. Finally, since we want to retrieve the best k combined matches, branches that cannot improve the already found results can be pruned.

Pruning by local caching of failed branches

To reduce the need for searching in branches that do not contain any solutions, caching can be used. Specifically, if some branch was explored without a success and later the *directCollisionFreeSuccessors* procedure is about to explore a different branch, we can first test if this new branch has same characteristics as the branch that failed before. If the new branch is found to be "equivalent" to some of the failed branches it can be pruned. We are using a local caching instead of caching all failed branches which would require a lot of memory. Each time the *directCollisionFreeSuccessors* is called, the new local cache is created. Of course, this approach is less effective than maintaining a global cache, but it has a benefit of a good memory control.

To allow the caching to work, we need to define the meaning of equivalence of two branches with respect to the failure of the search. As a matter of fact we are not interested in an equivalence but in an implication: having a branch in which the search failed we need to identify a situation when this branch implies that a different branch will necessarily fail as well. This can be achieved by comparing the sets of advertisements that were removed from the $PreMatch$ structure before exploring the particular branch.

Consider the $PreMatch$ table in Figure 9.2 and let us assume for a moment that there was no solution found for the branch starting with A assigned to e_1. In this branch the advertisements $R_1 = \{F, B, E, D\}$ were removed from $PreMatch$ (i.e., were greyed out in the figure). Now, imagine that in the next iteration B is assigned to e_1 and forward checking is called which removes some other advertisements. Let R_2 be the set of advertisements removed

from $PreMatch$ for the second branch for which we are not sure if we should explore it or not. Since any solution can consist only of advertisements that were not removed, we can conclude that if every advertisement in R_1 is also contained in R_2 (i.e., in the second branch at least the same advertisements were removed as in the first branch), the second branch cannot contain any solution, because if there was some, it would have been found in the first branch as well. This insight allows us to skip a branch if its set of removed advertisements is a superset of removed advertisements of some cached branch that failed.

Pruning by already found direct successors

The purpose of the *directCollisionFreeSuccessors* procedure is to find all direct successors of the base combined match $\mathsf{M}_{\mathsf{base}}$. During the search process the found direct matches are stored in the *Successors* set. The already found matches in the *Successors* set can be used to prune those branches in which every new match will necessarily be a successor of some match $\mathsf{M} \in Successors$ (because every new match $\mathsf{M}_{\mathsf{new}}$ which is a successor of M cannot be a direct successor of $\mathsf{M}_{\mathsf{base}}$). To test whether some branch can be pruned we simply compare the indexes of matches $\mathsf{M} \in Successors$ with the smallest possible index of the newly constructed match $\mathsf{M}_{\mathsf{newSmallest}}$ considering the branch is used. If $\mathsf{M} <_{preMatch} \mathsf{M}_{\mathsf{newSmallest}}$ for some $\mathsf{M} \in Successors$, the branch can be pruned safely.

Pruning by the best estimate of an overall matching degree

Finally, since only the best k combined matches need to be retrieved, an information about the already found matches and their overall matching degrees can be used to prune those branches which cannot lead to any improvement of the solution. Let M_{k} be the k-th best match found so far in the *combinedMatchTopK* procedure (Algorithm 3). Notice, that M_{k} might not be the *overall* best k-th solution since some matches with a better overall matching degree than the one of M_{k} can have direct successors which will also have a better matching degree. However, the overall matching degree of M_{k}, $\mathcal{AG}(\mathsf{M}_{\mathsf{k}})$, can be used to prune some branches as follows. Let B be a branch we are considering to be explored, and let $minMatchDegree_B$ be a lower bound estimate of all overall matching degrees for matches that can be obtained by using the branch B. The branch B can be pruned if $\mathcal{AG}(\mathsf{M}_{\mathsf{k}}) < minMatchDegree_B$ since no match found by exploring B can have an overall matching degree better than the one of M_{k}. Thus, the only problem that remains is to calculate the estimate $minMatchDegree_B$ for a given branch B. This can be actually done quite easily. Let us assume that we are considering the branch B at the depth *depth* in the *directCollisionFreeSuccessors* procedure (Algorithm 4), and let $\mathsf{M}_{\mathsf{new}}$ be a combined match that is being constructed by this procedure. The match $\mathsf{M}_{\mathsf{new}}$ already consists of *depth* advertisements (each one was added at previous levels of recursion) and there is still $n-depth$ advertisements to be specified in $\mathsf{M}_{\mathsf{new}}$ to guarantee that all requested n outputs/effects are produced by the combined match. The remaining $n - depth$ advertisements have to be taken from the branch B, which actually coincides with unused columns in the $PreMatch$ structure. We construct a combined match $\mathsf{M}_{\mathsf{minMatchDegree_B}}$ simply by using all advertisements from $\mathsf{M}_{\mathsf{new}}$ and by adding an advertisement with the minimal index from each unused column $E_d \in PreMatch$, for $d \in (depth, n\rangle$.[6] The combined match $\mathsf{M}_{\mathsf{minMatchDegree_B}}$ constructed in this fashion produces all n requested outputs/effects and for every match M_{B} that can be found in the branch B the following holds $\mathcal{AG}(\mathsf{M}_{\mathsf{minMatchDegree_B}}) \leqq \mathcal{AG}(\mathsf{M})$ since $\mathcal{AG}$ is monotonic and monotonic in all its parameters and since the advertisements with the minimal possible indexes were added into $\mathsf{M}_{\mathsf{minMatchDegree_B}}$. Thus $minMatchDegree_B = \mathcal{AG}(\mathsf{M}_{\mathsf{minMatchDegree_B}})$ can be used as a lower bound for the branch B.

This concludes our discussion on optimizations of combined match retrieval.

[6] I.e., if $\mathsf{M}_{\mathsf{base}}$ is the base combined match with the index $\langle k_1, \ldots, k_n\rangle = index(\mathsf{M}_{\mathsf{base}}, PreMatch)$, for each $d \in (depth, n\rangle$, A_{k_d} from E_d is added to $\mathsf{M}_{\mathsf{minMatchDegree_B}}$.

10 Related Work

Traditionally, the Web services discovery field focuses on discovery of individual services, while our work is targeting the problem of discovering sets of service advertisements. The body of work in individual service discovery is rather big and therefore we only review some representative approaches and then we focus on research that is most relevant to discovery of sets of services or service compositions.

Discovery of individual services

In essence, our work extends the seminal work of Paolucci et. al. [15] which defines the matchmaking conditions for Semantic Web services, discusses the requirements for effective and efficient discovery, and introduces the matchmaking algorithm. The matchmaking is focused on semantic similarities between the service request and service advertisements, based on matching degrees between service inputs and outputs (i.e., the functional characteristics of the service), and it also considers other criteria such as service categorisation. The work of Paolucci et. al. has its roots in discovery research in agents community. Specifically, some concepts and ideas originate in the LARKS notation [18] developed for discovery of agents based on their capabilities. The specific matching conditions defined by Paolucci et. al. for Semantic Web services are derived from matching conditions for generic software components introduced by Zaremski et. al. [21].

The work of Paolucci et. al. was recently extended in several directions. For example, Klusch et. al. [12] address the problem of flexible matchmaking by using a hybrid matchmaker which combines the semantic approach with approaches from the information retrieval field. Recently Kaufer and Klusch [8] developed a WSMO-MX Matchmaker, which is a hybrid matchmaker for WSMO services that employs the graph-matching approach combined with syntactic similarity measurement. Bellur et. al. [2] analyzes the correctness of [15] and suggests an improved algorithm based on bipartite graph matching.

Another approach to service discovery was developed by Bernstein et al. [4]. The authors propose to use process ontologies to describe the behaviour of services and then to query such ontologies using a Process Query Language (PQL).

Discovery of services combinations

The approach to service discovery developed as part of the METEOR-S framework by Verma et. al. [20] focuses on dynamic binding of services in service workflows represented as composite service templates. As such the approach can be seen as a step towards dealing with service combinations or compositions. However, the authors do not attempt to dynamically discover or synthesize service compositions. Rather, the focus is on selecting services that can instantiate an abstract workflow while optimizing some global criteria such as an overall cost.

Benatallah et. al. in [3] 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 many outputs of the request as possible and requiring as little inputs that are not provided by the requester as possible. Although the approach of Benatallah et. al. is similar to our work, there are several important differences. First, the matching criteria is different. While Benatallah et. al. allow some outputs or inputs to be missing, in our case we require all outputs to be produced while using only provided inputs. While the approach of Benatallah might be more flexible in some situations, our matching conditions are giving stronger guarantees. Additionally, we consider the problem of effect collisions, which is not addressed in [3].

Another approach similar to our work was developed by Kster et. al. in [13]. 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 basic discovery approach based on the DIANE language [9, 10] so as to allow combinations of services to be discovered in order to satisfy requests with multiple effects. In terms of combination based matching, our notion of a combined match is very similar in nature to the concept of multiple effects matching, however there are several substantial differences between our work and [13]. First of all, the DIANE language is quite different from OWL-S, and also the basic matching algorithm differs substantially. The DIANE matching is based on graphs matching and fuzzy sets comparisons, while OWL-S relies primarily on description logic reasoning. Another signifficant difference is that while we only consider service matching based on types of IOPEs of the service, Kster et. al. additionally consider so called instance level service retrieval, which means that services are matched also based on particular values (instances) that they are able to produce. On the other hand, in our approach we consider possible collisions between service advertisements and we also focus on the top k retrieval algorithm.

Another thread in the discovery research focuses on discovery of proper service compositions and on considering behavioral aspects of composite services as a possible matchmaking criterion. For example, Brogi et. al. in [7] consider both, discovery based on information specified in service interfaces, and also discovery based on behavioral aspects of service in the form of supported interaction protocols. The OWL-S workflow of discovered services is tranformed into Petri net representation which is used during discovery for checking properties such as a deadlock freedom. In the later work, Bonchi et. al. [6] define a behavioural equivalence of Web services based on bisimilarity. Although the authors claim that the proposed equivalence can be used for behaviour-aware discovery purposes, it is not very clear whether such an approach would be tractable and whether it is possible to develop an efficient discovery algorithm.

With respect to ranking of service compositions, an approach similar to our work was developed recently by Binder et. al. in [5]. The authors propose a ranking based on a numeric expression provided as part of the request, and a directory service which supports such user-defined selection and ranking expression. The expressions support a combination of arithmetic operators such as min, max, $+$, $-$, and set operators such as $union$, $intersection$, etc., which operate 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 define a specific type of ranking, the expressions are somewhat non-intuitive.

11 Conclusions and Future Work

In this report 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. We developed an efficient matchmaking algorithm for retrieval of top k combined matches without undesired and contradictory effects and we showed that the algorithm has similar time complexity as retrieval of individual services. On the other hand, we proved that retrieval of combined matches containing no effect duplications is an NP-hard problem and we developed a top-k retrieval algorithm employing optimization techniques known from constraint satisfaction problem solving that substantially reduce the search space. The focus on avoiding collisions and on the efficient retrieval of top k matches distinguishes our work from other research that considers discovery of sets of services.

Our approach to discovery of service combinations presents an important extension to matchmaking methods for discovering individual services. Such an extension finds its use mostly in dynamic environments in contexts such as service composition or process mediation, where it is not realistic to assume the perfect knowledge of the environment. We argued that introducing matches derived from service combinations increases substantially the likelihood of a successful match. However, at the same time, it turned out that by allowing several services to be a solu-

tion for a particular request also a careful attention has to be paid to possible collisions between services that are constituting the resulting match.

The main 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 service compositions. In terms of efficiency and complexity of discovering full fledged service composition, expectations are constrained by a known fact that the problem of service composition is NP-hard. Still, it is meaningful to try to develop for example an any-time approximation algorithm. Another open research question is related to aggregate ranking functions of combined or composed matches. As opposed to matching of individual services, ranking of service combinations is a rather new and open question. Similarly, also questions of evaluation of non-functional parameters such as reliability or security of combined or composed matches needs to be addressed.

Bibliography

[1] A. Atif, E. Mohamed, and S. A. El. Classification of the state-of-the-art dynamic web services composition techniques. *International Journal of Web and Grid Services*, 2:148 – 166, 2006.

[2] U. Bellur and R. Kulkarni. Improved matchmaking algorithm for semantic web services based on bipartite graph matching. In *IEEE International Conference on Web Services*, pages 86–93. IEEE Computer Society, 2007.

[3] B. Benatallah, M.-S. Hacid, C. Rey, and F. Toumani. Request rewriting-based web service discovery. In *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 2003.

[4] A. Bernstein and M. Klein. Discovering services: Towards high-precision service retrieval. In C. Bussler, R. Hull, S. A. McIlraith, M. E. Orlowska, B. Pernici, and J. Yang, editors, *WES*, volume 2512 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2002.

[5] W. Binder, I. Constantinescu, and B. Faltings. A flexible directory query language for the efficient processing of service composition queries. *Int. J. Web Service Res*, 4(1):59–79, 2007.

[6] F. Bonchi, A. Brogi, S. Corfini, and F. Gadducci. A behavioural congruence for web services. In F. Arbab and M. Sirjani, editors, *FSEN*, volume 4767 of *Lecture Notes in Computer Science*, pages 240–256. Springer, 2007.

[7] A. Brogi and S. Corfini. Behaviour-aware discovery of web service compositions. *Int. J. Web Service Res*, 4(3):1–25, 2007.

[8] F. Kaufer and M. Klusch. WSMO-MX: A logic programming based hybrid service matchmaker. In *ECOWS*, pages 161–170. IEEE Computer Society, 2006.

[9] M. Klein and B. König-Ries. Coupled signature and specification matching for automatic service binding. In L.-J. Zhang, editor, *2004 European Conference on Web Services*, volume 3250 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2004.

[10] M. Klein, B. König-Ries, and M. Mussig. What is needed for semantic service descriptions? A proposal for suitable language constructs. *International Journal of Web and Grid Services*, 1(3/4):328–364, 2005.

[11] M. Klusch, B. Fries, and K. P. Sycara. Automated semantic web service discovery with OWLS-MX. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 915–922. ACM, 2006.

[12] M. Klusch, B. Fries, and K. P. Sycara. Automated semantic web service discovery with OWLS-MX. In H. Nakashima, M. P. Wellman, G. Weiss, and P. Stone, editors, *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 915–922. ACM, 2006.

[13] U. Küster, B. König-Ries, M. Klein, and M. Stern. Diane - a matchmaking-centered framework for automated service discovery, composition, binding and invocation. In *Proceedings of the 16th International World Wide Web Conference (WWW2007)*, Banff, Alberta, Canada, May 2007.

[14] S. McIlraith and T. C. San. Adapting Golog for composition of semantic web services. In *KR2002: Principles of Knowledge Representation and Reasoning*, pages 482–493. Morgan Kaufmann, San Francisco, California, 2002.

[15] 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.

[16] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002.

[17] T. J. Schaefer. The complexity of satisfiability problems. In *ACM Symposium on Theory of Computing (STOC '78)*, pages 216–226, New York, May 1978. ACM Press.

[18] K. P. Sycara, M. Klusch, S. Widoff, and J. Lu. Dynamic service matchmaking among agents in open information environments. *SIGMOD Record*, 28(1):47–53, 1999.

[19] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 2004.

[20] K. Verma, K. Gomadam, A. Sheth, J. 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.

[21] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(4):333–369, 1997.