

1 Slides Description

Problem addressed in our algorithm is: given a set of abstract services, a set of concrete services, a user query and a set of user quality preferences, derive a set of service compositions that answer the query and fulfill the quality preferences.

Our related work relies on two different domains: query rewriting algorithms using views (database domain) [?] and query algorithm in the service composition domain [?].

We assume that:

- The query expresses an abstract composition that describes the requirements of a user. It is expressed with respect to a catalogue of abstract services
- A concrete service is defined in terms of an abstract composition. It can be associated to a single abstract service or to a composition of abstract services
- Concrete services are tagged quality measures. Not all services are tagged with the same measures. Every measure is defined in a catalogue
- Each measure is written in the form: *constant* (measure name) *operation* (<, >, =, ≤, ≥) *value* (the static value associated to the measure)
- There are two types of measures: single and composite measures. The single measures is the simplest type. It is a static measure which has a name associated with an operation and a value. The composite measure is dynamically computed measure. It is defined as aggregations of single measures. All measures we are using are in a catalogue. Example: Single measure: availability, price per call, price per request, response time, location, provenance, etc. Composite measure: total price or cost which are computed by adding price per call and price per request values of all services included in the service composition. Total response time is computed by adding the response time of all services included in the service composition.

```
<measures>
<singlemeasure name='price per request' type='double' />
<singlemeasure name='price per call' type='double' />
<compositemeasure name='total price' type='double' action='sum'>
<singlemeasure name='price per request' type='double' />
<singlemeasure name='price per call' type='double' />
</compositemeasure>
</measures>
```

The parameter "action" in the composite measure specifies how the single measures should be aggregated (adding, average, product, etc).

The table 2 contains the abstract services that will be used in our example.

Let us suppose the following query. *The user wants to retrieve patient's personal and DNA information of patients who were infected by a disease «K» using services that have availability higher than 98%, price per call less than 0.2 dollars, and total cost less then 1 dollar.*

The query which express the example in terms of abstract services is specified below. The decorations ? and ! are used to specify input and output parameters, respectively.

$$Q(d?, patientinfo!, dna!) := DiseaseInfectedPatient(d?, p!), PatientPersonalInformation(p?, info!), PatientDNA(p?, dna!) \{p = "K"\} [availability > 98, price\ per\ call < 0.2, total\ cost < 1]$$

<i>Abstract Service</i>	<i>Description</i>
<i>DiseaseInfectedPatients</i> ($d?, p!$)	Given a disease d , a list of patients p infected by it is retrieved.
<i>DiseaseInfectedPatients</i> ($d?, p!, op!$)	Given a disease d , a list of patients p infected by it is retrieved, and op is an optional boolean output indicating if the operation proceeded well or not.
<i>PatientDNA</i> ($p?, dna!$)	Given a patient p , his DNA information dna is retrieved.
<i>PatientPersonalInformation</i> ($p?, info!$)	Given a patient p , his personal information $info$ is retrieved.

Table 1: Abstract services description

The query expressed in terms of abstract services including its constraints and preferences. In the current implementation braces ($\{\}$) and brackets ($[]$) are used to distinguish constraints from preferences.

The first step of the algorithm looks for concrete services that can be matched with the query. We have three matching problems associated to this step: abstract service matching, measure matching and concrete service matching.

- *abstract service matching*: a abstract service A can be matched with a abstract service B only if: (i) they have the same name. In this case we are assuming they perform the same function; and (ii) the number and type of variables should be compatible. This means that the number of input and output variables of A must be equal or higher than the number of input and output variables of B . Consider the example below.

1) $A1(x?, c!)$

2) $A1(a?, b!, c!)$

In this example, 2 can be matched to 1, because the number of input and output variables of 2 is higher than the number of input and output variables of 1. However 1 can not be matched to 2, because the number of input and output variables of 1 is less than the number of input and output variables of 2.

- *measures matching*: all single measures in the query must exist in the concrete service, and all of them can not violate the measures in the query.
- *concrete service matching*: one concrete service can be matched with the query if all its abstract services can be matched with the abstract service in the query (satisfying the *abstract service matching* problem) and all the single measures in the query can be matched with the concrete service measures (satisfying the *measures matching* problem). Consider the example below.

$Q(x?, y!) = A1(x?, c!), A2(c?, y!)$ [availability > 98, price per call < 0.2, total price < 1]

$S1(a?, b!) = A1(a?, b!)$ [availability > 98] (It can not be matched. Price per call is missing.)

$S2(a?, b!) = A1(a?, b!)$ [availability > 98, price per call = 0.2] (It can not be matched. Price per call is violating the query preference.)

$S3(a?, b!) = A1(a?, c!), A3(c?, b!)$ [availability > 98, price per call = 0.1] (It can not be matched. A3 is not a abstract service in the query.)

$S4(a?, b!) = A1(a?, b!)$ [availability > 98, price per call = 0.1]

$S5(a?, b!) = A1(a?, c!), A2(c?, b!)$ [availability > 99, price per call = 0.1]

$S5(a?, b!) = A1(a?, c!), A2(c?, b!)$ [availability > 99, price per call = 0.1, location = \close"] (Even have more single measures than the query, it can be matched)

The principle of the solution for the first step of the algorithm is: given (i) a query and a set of user quality preferences; and (ii) a set of concrete services tagged with quality measures, looks for concrete services which respects the matching rules. The concrete service that matches the rules is a candidate concrete service. The result is a list of candidate concrete services which may be used in the rewriting process.

Related work on web service selection and composition considering QoS aspects [].

To illustrate the selection of candidate services, consider the query in the example presented before and the following concrete services.

```
S1(a?,b!) := DiseaseInfectedPatients(a?,b!)[availability > 99, price per call = 0.1]
S2(a?,b!) := DiseaseInfectedPatients(a?,b!)[availability > 98, price per call = 0.2]
S3(a?,b!,c!) := DiseaseInfectedPatients(a?,b!)[availability > 98, price per call = 0.1]
S4(a?,b!) := PatientDNA(a?,b!)[availability > 98, price per call = 0.1]
S5(a?,b!) := PatientDNA(a?,b!)[availability > 96, price per call = 0.1]
S6(a?,b!) := DiseaseInfectedPatients(a?,k!), PatientDNA(k?,b!)[availability > 93]
S7(a?,b!) := PatientPersonalInfo(a?,b!)[availability > 99, price per call = 0.1]
S8(a?,b!) := PatientPersonalInfo(a?,b!)[availability > 98, price per call = 0.1]
S9(a?,c!,b!) := PatientPersonalInfo(a?,b!),PatientDNA(a?,c!)[availability > 98, price per call = 0.1]
```

In this example, the services S2, S5 and S6 can not be selected as candidate concrete services since they violate user preferences

The second step of the algorithm tries to create *concrete services description* (CSD) to be used in the rewriting process. A CSD maps abstract services and variables of a concrete service to abstract services and variables of the query. A CSD is created according to the following variable mapping rules:

- *Rule 1*: head variables in the concrete service can be mapped to head or local variables in the query if they are from the same type
- *Rule 2*: local variables in the concrete service can be mapped to head variables in the query if they are from the same type
- *Rule 3*: local variable in the concrete service can be mapped to a local variable in the query if: (i) they are from the same type; and (ii) the concrete service cover all abstract service in the query that depends on this variable. Depends here means that this local variable is used as input in another abstract service.

The principle of the solution for the second step is: given a list of candidate concrete services, looks for a candidate concrete services that satisfies the the variable mapping rules. For the ones who satisfies the rules, a CSD is created. As result a list of CSDs is produced.

To illustrate the selection and creation of concrete service description, consider the concrete services selected in the previous step below.

```
S1(a?,b!) := DiseaseInfectedPatients(a?,b!)[availability > 99, price per call = 0.1]
S3(a?,b!,c!) := DiseaseInfectedPatients(a?,b!,c!)[availability > 98, price per call = 0.1]
S4(a?,b!) := PatientDNA(a?,b!)[availability > 98, price per call = 0.1]
S7(a?,b!) := PatientPersonalInfo(a?,b!)[availability > 99, price per call = 0.1]
S8(a?,b!) := PatientPersonalInfo(a?,b!)[availability > 98, price per call = 0.1]
S9(a?,c!,b!) := PatientPersonalInfo(a?,b!),PatientDNA(a?,c!)[availability > 98, price per call = 0.1]
```

In this example six CSDs are created once all concrete services satisfy the rules to create CSDs. The third step of the algorithm generates all combinations of CSDs. As result we have a list of list of CSDs.

The fifth and final step of the algorithm verifies/identifies which combination of CSDs is a valid rewriting of the user query. A combination of CSDs is a valid rewriting if:

- The number of abstract services in the query is equal to the result of adding the number of abstract services of each CSD
- There is no duplicity/redundancy of abstract services in the list of CSD
- All head variables in the query must be mapped to a variable in one of the concrete services in the list of CSDs

- If the query has a composite measure, this measure is updated for each rewriting produced, and this measure can not be violated

As result of this step we have a list of rewriting of the query. To illustrate let us consider the example used before. In our query we have a preference which is associated to the rewritings (composite measure) and not to a single service. Considering this preference, we have to update its value while producing the rewritings. The value of total cost in this example is updated by aggregating the value of price per call of each service. The rewritings produced that can satisfy the user preference while aggregating these values are below. Note that more than three rewritings can be produced that composite measure did not exists. The rewritngs are listed in the lexicographical order considering the concrete services .

$Q(\text{disease?}, \text{info!}, \text{dna!}) := S1(\text{disease?}, p!) S7(p?, \text{info!}) S4(p?, \text{dna!})$
 $Q(\text{disease?}, \text{info!}, \text{dna!}) := S3(\text{disease?}, p!, _) S7(p?, \text{info!}) S4(p?, \text{dna!})$
 $Q(\text{disease?}, \text{info!}, \text{dna!}) := S1(\text{disease?}, p!) S8(p?, \text{info!}) S4(p?, \text{dna!})$

2 Formal definition of the Rhone service based query rewriting algorithm

The basic input for the Rhone algorithm is: (1) a query; (2) a list of concrete services.

Definition 1 (Query): A query Q is defined as a set of *abstract services*, a set of *constraints*, and a set of *user preferences* in accordance with the grammar:

$$Q(\bar{I}, \bar{O}) := A_1(\bar{I}, \bar{O}), A_2(\bar{I}, \bar{O}), \dots, A_n(\bar{I}, \bar{O}), C_1, C_2, \dots, C_m[P_1, P_2, \dots, P_k]$$

The left side of the definition is called the *head* of the query; and the right side is called the *body*. \bar{I} and \bar{O} are a set of *input* and *output* parameters, respectively. Input parameters in both sides of the definition are called *head variables*. In contrast, input parameters only in the query body are called *local variables*. Abstract services (A_1, A_2, \dots, A_n) describes a set of basic service capabilities. C_1, C_2, \dots, C_m are constraints over the *input* and/or *output* parameters. The user preferences (over the services) are signed in P_1, P_2, \dots, P_k . C and P are in the form $x \otimes \text{constant}$ such that $\otimes \in \{\geq, \leq, =, \neq, <, >\}$.

To illustrate the definition, let us suppose the set of abstract services in Table 2 and the Example 1.

<i>Abstract Service</i>	<i>Description</i>
<i>DiseaseInfectedPatients(d?, p!)</i>	Given a disease d , a list of patients p infected by it is retrieved.
<i>PatientDNA(p?, dna!)</i>	Given a patient p , his DNA information dna is retrieved.
<i>PatientPersonalInformation(p?, info!)</i>	Given a patient p , his personal information $info$ is retrieved.

Table 2: Abstract services description

Example 1: The user wants to retrieve the DNA information from patients infected by the disease ‘K’ using services that have availability higher than 99%, price per call less than 0.2 dollars, and the total cost less then 1 dollar.

The query which express the Example 1 according to the Definition 1 and the abstract services in Table 2 is specified below. The decorations ? and ! are used to specify input and output parameters, respectively.

$$Q(d?, dna!) := DiseaseInfectedPatients(d?, p!), PatientDNA(p?, dna!), \\ d = "K"[availability > 99\%, price\ per\ call < 0.2\$, total\ cost < 1\$]$$

Analyzing the query, it is possible to note that the parameters “d?” and “dna!” appear in both sides of the definition. Due to that they are *head* variables. On the other hand, “p!” and “p?” are *local* variables considering that they appear only in the body definition. Additionally, note that the local variables “p!” and “p?” have the same name. Intuitively, this fact indicates a dependency between the abstract services which use these variables (in that case *DiseaseInfectedPatients* and *PatientDNA*).

In the example, *DiseaseInfectedPatients* and *PatientDNA* are abstract services that specify basic service functions which are combined to answer the query. The constraint ($d = "K"$) over the input parameter ‘d’ will be further used while executing the query over a database (the where clause). *Availability*, *price per call* and *total cost* are the user preferences over the services.

Definition 2 (Concrete service): A concrete service (S):

$$S(\bar{I}, \bar{O}) := A_1(\bar{I}, \bar{O}), A_2(\bar{I}, \bar{O}), \dots, A_n(\bar{I}, \bar{O})[P_1, P_2, \dots, P_k]$$

A concrete service (S) is defined as a set of abstract services (A), and by its quality constraints P . These quality constraints associated to the service represent the service level agreement exported by the concrete service.

Example 2: Considering the query (see Example 1) and the abstract services (see Table 2), the concrete services below are examples in accordance with the Definition 2.

$$\begin{aligned} S1(a?, b!) &:= DiseaseInfectedPatients(a?, b!)[availability > 99\%, price\ per\ call = 0.2\$] \\ S2(a?, b!) &:= DiseaseInfectedPatients(a?, b!)[availability > 99\%, price\ per\ call = 0.1\$] \\ S3(a?, b!, c!) &:= DiseaseInfectedPatients(a?, b!, c!)[availability > 98\%, price\ per\ call = 0.1\$] \\ S4(a?, b!) &:= PatientDNA(a?, b!)[availability > 99.5\%, price\ per\ call = 0.1\$] \\ S5(a?, b!) &:= PatientDNA(a?, b!)[availability > 99.7\%, price\ per\ call = 0.1\$] \\ S6(a?, b!) &:= PatientPersonalInformation(a?, b!)[availability > 99.7\%, price\ per\ call = 0.1\$] \\ S7(a?, b!) &:= PatientDNA(a?, c!), PatientPersonalInformation(c?, b!)[availability > 99.7\%, price\ per\ call = 0.1\$] \end{aligned}$$

Given the query and a list of concrete services as input, the algorithm looks for candidate concrete services. Candidate concrete service is a concrete service that probably can be used in the rewriting process. It contains only abstract services which are also query abstract services, and with the same signature (same name and number of input/output variables). The candidate concrete services are chosen while searching for matches between abstract services in S and abstract service in Q .

Definition 3 (abstract service equivalence): A match between abstract services occurs when an abstract service A_i is equivalent to A_j , denoted $A_i = A_j$. Given two abstract services A_i and A_j , $A_i = A_j$ iff: (1) A_i and A_j have the same abstract function name; (2) the number of *input* parameters of A_i is equal to A_j ; and (3) the number of *output* parameters of A_i is equal to A_j . For example, looking to the concrete services in the Example 2, the abstract service *DiseaseInfectedPatients* in $S1$ and $S2$ are equivalent to the abstract service *DiseaseInfectedPatients* in the query Q (Example 1) once they have the same name and number of input/output parameters. On the other hand, the abstract service *DiseaseInfectedPatients* in $S3$ is not equivalent to the abstract service *DiseaseInfectedPatients* in the query because the number of parameters are different.

Based on the assumptions that: (a) a concrete service can represent a service composition in which the abstract services involved may be able not only to retrieve data, but also to execute business rules that may impact the entire system; and (b) the execution of a concrete service

consists in executing all its abstract services. A concrete service (S) is selected as *candidate* to the rewriting process if for each abstract service in S there is an equivalent in Q ; there is no abstract service in S that does not exist in Q ; and the quality constraints in Q must be guaranteed in S .

Definition 4 (candidate service): Given a query Q and a concrete service S , S is a *candidate* service iff: (1) $\nexists A_i$ s.t. $A_i \in S$ and $A_i \notin Q$; and (2) the quality constraints in S does not violate the user preferences in Q .

For example, considering the query in the Example 1 and the concrete services in the Example 2, it is possible to see that: (1) $S1$ is not a candidate service because it violates a user preference (*price per call*); (2) $S3$ and $S7$ are not a candidate service because they have abstract services that are not in Q ; and (3) $S2$, $S4$ and $S5$ are candidate services once: all their abstract services have an equivalent in Q and there is no violation in the user preference.

A *candidate service description* (CSD) describes how a *candidate* concrete service can be used in the query rewriting process. It is a complex data structure which includes: mappings from variables in a concrete service to variables in the query; mappings from variables on the head of a concrete service to variables on its body; a set of abstract services that represents partially or fully the abstract services in the query; and a set of quality constraints associated to the concrete service. Intuitively, a rewriting is a set of *candidate service descriptions* that fully covers the original query, and do not violates the user preferences.

Definition 5 (candidate service description): A CSD is represented by an n-tuple:

$$\langle S, h, \varphi, G, P \rangle$$

where S is a concrete service. h are mappings between variables in the head of S to variables in the body of S . φ are mapping between variables in the concrete service to variables in the query. G is a set of abstract services covered by S . P is a set quality constraints associated to the service S .

The CSD for a given service will be created following rules: (1) for all head variables in S , there is a mapping for a head variable in Q ; and (2) if x is a local variable in S mapped to a local variable in Q , then S must cover all abstract services in Q which uses x or cover only one abstract service that uses x .

Example 3: To illustrate the rules above consider the following example. *The user wants to retrieve the personal information and the DNA information from patients infected by disease “K”.* Supposing we have the query Q and the concrete services $S1$, $S2$, $S3$ and $S4$:

$$\begin{aligned} Q(d?, info!, dna!) &:= \\ DiseaseInfectedPatients(d?, p!), PatientDNA(p?, dna!), PatientPersonalInformation(p?, info!) \\ S1(a?, b!) &:= DiseaseInfectedPatients(a?, c!), PatientDNA(c?, b!) \\ S2(a?, b!) &:= PatientPersonalInformation(a?, b!) \\ S3(a?, b!) &:= DiseaseInfectedPatients(a?, b!) \\ S4(a?, b!) &:= PatientDNA(a?, b!) \end{aligned}$$

In the query Q it is possible to note that “ $p!$ ” is a *local* variable which is used as input (“ $p?$ ”) for the abstract services $A2$ and $A3$. Looking to the concrete service $S1$ no CSD will be created for it because the *local* variable $c!$ is mapped to the local variable $p!$, but $S1$ does not cover all abstract services which expects that variable. On the other hand, CSDs are constructed to the services $S2$, $S3$ and $S4$ once even existing the mapping from a local variable in the concrete service to a local variable in the query, all of them only cover one abstract service which uses that *local* variable. To be more clear about these rules, consider the rewriting below in which the CSDs for the services $S2$, $S3$ and $S4$ are used:

$$Q(d?, info!, dna!) := S3(d?, p!), S4(p?, dna!), S2(p?, info!)$$

The rewriting above is the only one possible for the query. However, let us suppose that a CSD for $S1$ was created violating the rule number two, consequently the wrong rewriting below would be created:

$$Q(d?, info!, dna!) := S1(d?, info!), S4(p?, dna!)$$

The problem here is regarding the *local* variable $p?$ which appears in $S4$, and it apparently should come from $S1$, but we can not guarantee that the same *local* variable internally used in $S1$ is the one expected by $S4$. That is the reason the rule two exists.

3 Query rewriting approaches

A query rewriting algorithm which processes queries on data provider services is proposed in [?]. A query is defined using SPARQL and data services are modeled as RDF views. Both can be seen as a graph. Given a query and a set of data services, the algorithm searches for relevant services and creates a mapping table for them. This table shows different ways of using a data service to cover part of the query. Then, based on the mapping table, the algorithm generates different combinations of data services to answer the query. A valid combination (a rewriting answer) is a service composition in which the set of data service graphs fully satisfy the query graph.

A service composition framework to answer preference queries is proposed in [?]. The concept of preferences is included to SPARQL queries, and fuzzy constraints to services. Services and service compositions are ranked according to a fuzzy dominance relationship and fuzzy scores. Two algorithms based on [?] to generate the rewriting compositions are presented: (1) the first produces all possible rewritings before computing their scores, and then return the best ones; (2) the second uses a quality metric that combines diversity and accuracy to, incrementally, rank services and to build the best rewritings.