A Formal Framework for Describing Information Providing Web Services

Andrey Bovykin
Department of Computer Science
University of Liverpool
`andrey@csc.liv.ac.uk`

Evgeny Zolin
School of Computer Science
University of Manchester
`ezolin@cs.man.ac.uk`

Abstract

In this paper, we introduce a formal framework for describing Semantic Web Services using Description Logic. Specifically, for *information providing* services, the notion of a *service description* is proposed. From syntactic point of view, it is an extension of the way services are described in OWL-S Service Profile (namely, of its part concerning description of inputs and outputs). However, from the semantical viewpoint, the definition of *service matching* introduced here (based on the proposed descriptions) appears to be more appropriate for service discovery purposes.

The extended notion of service description includes, besides the types of its inputs and outputs, a specification of relationships between inputs and outputs. This part has the form of a conjunctive query. We show that the reasoning problem of service matching for this kind of descriptions is reducible to checking subsumption between two conjunctive queries w.r.t. an ontology, which is a standard reasoning task.

1 Introduction

Semantic Web Services are programs available on the Web that can be executed by a user (a human or another program, so called a software agent). Their distinctive feature is that they are semantically marked up, i.e., supplemented with a *semantic annotation* – a formal description of functionality and other properties, which enables an automated *discovery*, *execution*, *composition*, and *execution monitoring* of services. These annotations (also called *service descriptions*) are formulated using terms (concepts and roles) whose semantics is well-defined in *ontologies* that are available on the Semantic Web.

Among others, one can distinguish *information providing* services. When such a service is executed, it accepts from a user an input data of a specified format ("typed data") and returns back to the user some information as an output. Most services of this kind are *stateless*, i.e., they only provide information about the current state of the world, but do not change that state.

Service descriptions can be divided into two logically different kinds (though they can share the same language). A service provider supplies a description of his service and puts it into a repository, so that the service could be found on the Web with the help of a search engine. This kind of description can be called a *service advertisement*. On the other side, a user specifies a description of a service that is to be found on the Web; this type of description will be referred to as a *service request*.

One of the most important reasoning task concerning services is so called *service discovery*. Given a service request Q, a search engine, using its reasoning system, compares the request with service advertisements stored in a repository, taking into account the background ontologies available on the Web (the request and the advertisements may refer to different ontologies). If an "appropriate" service S is found, the search engine returns to the user the location of S (its URL, port, etc.) together with a supplementary information (which inputs and outputs of Q correspond to input/outputs of S; does S match Q exactly, and if not, then how the service found is related to the user's request).

In this paper we present a formal framework for describing information providing (stateless) services. Let us point out the distinguishing features of this framework. Firstly, a description of a service is entirely based on standard background ontologies (in particular, we do not introduce any concepts or roles specially devoted to describing web services). The benefit of this is twofold: a) this enables to reuse, in service descriptions, the terminology from the existing and emerging ontologies available on the Web, thus making the descriptions succinct and equipped with well defined semantics; b) this allows to use, for service matching purposes, the semantics of the terms that occur in the descriptions, thus increasing the precision and recall of service discovery, as compared to the keyword-based search algorithm.

Secondly, we are concerned with (stateless) information providing services only. Consequently, service descriptions need not to contain pre- or post-conditions, since these services are always executable, and the state of the world after their execution is the same as before.

Thirdly, our primary aim is to provide a solution for the service discovery problem formulated above. We will formalise, for service descriptions introduced in our paper, the notion of service *matching*. We also show that the problem of matching services is decidable and is in fact reducible to the query subsumption w.r.t. an ontology, which is a standard reasoning task (cf. [3, 4, 5]). Hence, for deciding service matching, we can reuse the existing reasoning systems that are capable to decide query subsumption for corresponding Description Logics.

2 Motivating examples

Recall that, in OWL-S Service Profile, the functionality of an "information providing" (stateless) service is described by specifying (the types of) its inputs and outputs. Accordingly, when two services – a service advertisement S and a user's service request Q – are tested for matching, only information about the types of their inputs and outputs is taken into account (see, e.g., [10]). The typical matching condition is: each input of S must subsume at least one input of Q, and each output of Q must subsume at least one output of S. This notion of service matching is not satisfactory for discovery information providing services, as illustrated by the following examples.

Example 2.1. Our first example is adopted from a short discussion in [9]. Consider a service S with an input of type $\mathsf{GeoRegion}$ and an output of type Wine. This description may correspond to a service that, given a name of a geographical region (e.g., 'France'), retrieves the list of (names of) wines that are *produced* in this region. On the other hand, suppose that a user (or a software agent) wants to find a service Q that, given a name of a geographical region as an input, would return the list of wines that are *sold* in that region. Using OWL-S Service Profile, one cannot distinguish between these two services, and as a result, a search engine will return this service S to the user, even though it is irrelevant to his request Q.

The next example illustrates the opposite situation, when a search engine based on OWL-S descriptions rejects services that in fact match user's request.

Example 2.2. Let S be the above considered service (which takes a $\mathsf{GeoRegion}$ as input and returns the list of Wines produced in this region). Now suppose that a user is looking for a service Q that takes a $\mathsf{FrenchGeoRegion}$ as input and returns the list of only $\mathsf{FrenchWine}$s produced in this region as output. Observe that the service S returns wines that, in general, may not be $\mathsf{FrenchWine}$s, and hence S does not match Q according to a trivial matching algorithm. However, from a background ontology it follows that S returns only $\mathsf{FrenchWine}$s when called with a $\mathsf{FrenchGeoRegion}$, and thus should be matched to this request Q.

Remark 2.1. In the latter example, we were able to match these two services because we used some information from our background ontology. What kind of information did we need for that? First, it is natural to assume that the ontology contains the following concept definitions:

$$\begin{gathered}\mathsf{FrenchWine} \equiv \mathsf{Wine} \sqcap \exists\, \mathsf{producedIn}.\mathsf{France}\\ \mathsf{FrenchGeoRegion} \equiv \mathsf{GeoRegion} \sqcap \exists\, \mathsf{isPartOf}.\mathsf{France}\end{gathered}$$

Now recall that our inference was: "*If a wine is produced in a* $\mathsf{FrenchGeoRegion}$*, then it is produced in* France" (and hence is a $\mathsf{FrenchWine}$). This can be formalised as a *concept inclusion* axiom:

$$\exists\, \mathsf{producedIn}.\exists\, \mathsf{isPartOf}.\mathsf{France} \sqsubseteq \exists\, \mathsf{producedIn}.\mathsf{France}.$$

Moreover, a similar axiom holds for all other countries, not only for France. A general statement that covers all these cases can be formalised as a *complex role inclusion* axiom:

$$\mathsf{producedIn} \circ \mathsf{isPartOf} \sqsubseteq \mathsf{producedIn}.$$

To sum up, a matching algorithm that takes into account only the types of inputs and outputs of services does not yield a satisfactory solution to the service discovery problem. Our work is intended to overcome these deficiencies. As follows from the above discussion, in addition to the types of inputs and outputs, a service description must be enriched with a specification of *relationships* between inputs, outputs, and possibly some 'intermediate' objects. Analysing numerous examples of services (including those in bioinformatics), it was observed that a notion of *conjunctive query* can be adopted for these purposes of expressing these relationships. Moreover, the properly formulated notion of service matching appears to be reducible to query containment w.r.t. an ontology – a task whose decidability and complexity is extensively explored (see, e.g., [2, 3, 4]). Now we proceed to the formal presentation of our framework.

3 Services as queries

We assume the reader to be familiar with syntax and semantics of Description Logics (cf. [1]). When describing web services, we will assume the existence of a background ontology (or a TBox) $\mathcal{T}$, which includes the definitions of all terms (concepts and roles) involved in our service descriptions. In reality, different service providers and users may describe their services using different ontologies. We take this issue into account by assuming that $\mathcal{T}$ is the union of all those ontologies.

Furthermore, when a particular web service is executed by a user at a particular moment of time, it deals with a partial description of the world (or an ABox) $\mathcal{A}$, which contains an information about the current state of affairs (i.e., about the membership of individuals in some classes and relationships between individuals). All services considered here are assumed to be *stateless*, i.e., after their execution the ABox $\mathcal{A}$ is left unchanged, and the states of the world before and after the execution of a service are identical. As a consequence, there is no need to include pre- and post-conditions in a service description. Finally, we can assume that both a TBox and an ABox are formulated in some Description Logic, say $\mathcal{SHIQ}$, whose vocabulary contains the sets of concept names *Concepts*, role names *Roles* and individual names (or constants) *Const*. The union of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$ is called a *knowledge base* and denoted by $\mathcal{KB} = \langle \mathcal{T}, \mathcal{A} \rangle$.

3.1 Describing services

Since inputs and outputs of services are assumed to be typed (with types usually being some DL concepts), we introduce the following notion.

Definition 3.1. A *typed tuple* (of variables) is a tuple of variables together with concepts assigned to each variable: $\vec{x}:\vec{X} := \langle x_1: X_1, \ldots, x_n: X_n \rangle$, where x_i are variables and X_i are concepts. The length of a tuple $\vec{x}$ is denoted by $|\vec{x}|$, and if $|\vec{x}| = n$ then $\vec{x}$ is said to be an n-tuple.

Now let us formulate our basic notion of service description; see Section 6 for extended versions.

Definition 3.2. A *service description* (or a *service*, for short) is an expression of the form

$$S := \langle \vec{x}:\vec{X}; \quad \vec{y}:\vec{Y}; \quad \Phi(\vec{x}, \vec{y}) \rangle,$$

where $\vec{x}:\vec{X}$ is a typed tuple of *inputs*, $\vec{y}:\vec{Y}$ is a typed tuple of *outputs*, and $\Phi(\vec{x}, \vec{y})$ is a *conjunctive query*, i.e., an expression of the form

$$\exists \vec{z} \; \big(\mathit{term}_1(\vec{x}, \vec{y}, \vec{z}) \wedge \ldots \wedge \mathit{term}_k(\vec{x}, \vec{y}, \vec{z}) \big),$$

where each conjunct $\mathit{term}_i(\vec{x}, \vec{y}, \vec{z})$ is either an expression of the form $w{:}C$ with C being a concept, or wRw' with R being a role and w, w' variables from the lists $\vec{x}, \vec{y}, \vec{z}$, or individual names.

In other words, $\Phi(\vec{x}, \vec{y})$ is a conjunction of ABox assertions about inputs $\vec{x}$, outputs $\vec{y}$, known objects (individual names) and unknown objects $\vec{z}$ (the latter are existentially quantified). In practice, this multi-component description of a service can be written down with the help of additional roles `hasInput` and `hasOutput` that relate the service S with its inputs x_i and outputs y_j (and possibly, with a new role that links the service S to the specification Φ). We will not go into these details of implementation syntax here.

3.2 Intended meaning

The tuple $\vec{x}$ stands for inputs and $\vec{y}$ for outputs of the service S, and their "types" are declared to be $\vec{X}$ and $\vec{Y}$, respectively. The meaning of the whole expression is that, given an input $\vec{a}$ from $\vec{X}$, the service returns as its output the (unordered) set of all tuples of objects $\vec{b}$ that belong to $\vec{Y}$ and satisfy the condition $\Phi(\vec{a}, \vec{b})$.

More precisely, suppose that a service description S is formulated using a terminology (TBox) $\mathcal{T}$. A user who executes a particular service s (i.e., a software program that works with a data stored in a knowledge base), submits a tuple of *individual names* as input for s, and the service s returns as output some tuples of individual names. These individual names come from the current partial description of the world (i.e., an ABox) $\mathcal{A}$ (which is usually subject to permanent changes). It is natural to expect that if a service provider declares that S is a description of a particular service s (w.r.t. a background ontology $\mathcal{T}$), then this should hold for *any* data that can be stored in an ABox. To formalise this, let us give a definition of what it means for a service s to implement a description S.

Definition 3.3 (Service semantics). A service s *implements* a description S over a TBox $\mathcal{T}$ if, for any ABox $\mathcal{A}$ and any individuals $a_1, \dots, a_m$ in the knowledge base $\mathcal{KB} = \langle \mathcal{T}, \mathcal{A} \rangle$, if $\mathcal{KB} \models X_i(a_i)$ for all $1 \leqslant i \leqslant m$, then

1. s accepts $\vec{a} = \langle a_1, \dots, a_m \rangle$ as input (i.e., does not return an error) and
2. when executed with $\vec{a}$ as input, the service s returns the set of all those tuples of individuals $\vec{b} = \langle b_1, \dots, b_n \rangle$ from $\mathcal{KB}$ that satisfy the condition

$$\mathcal{KB} \models \vec{b} : \vec{Y} \ \wedge \ \Phi(\vec{a}, \vec{b}).$$

From this semantics it can be observed that the role of concepts $\vec{X}$ and $\vec{Y}$ in a service description is not symmetric: the concepts $\vec{X}$ define exactly the set of inputs of the service S (i.e., each tuple $\vec{a}$ from $\vec{X}$ must be accepted by the service), whereas $\vec{Y}$ specifies only a superset of all outputs (a tuple $\vec{b}$ from $\vec{Y}$ is in the answer set of the service, for a given input $\vec{a}$, only if it additionally satisfies $\Phi(\vec{a}, \vec{b})$). In fact, we could define, from the very beginning, a service description consisting of only $\vec{x} : \vec{X}$ and $\Phi(\vec{x}, \vec{y})$ and consider $\vec{y} : \vec{Y}$ as just a part of $\Phi(\vec{x}, \vec{y})$; then all the subsequent exposition should be modified in an obvious way (and even look a bit simpler). However, we decided to keep the types of outputs explicit, to make it evident that our descriptions are straightforward extensions of the standard ones (in particular, of OWL-S descriptions).

Let us illustrate how this definition works on the example discussed in Section 2. The two services that take a geographical region and return all wines that are produced (resp., sold) in this region can be described now as follows:

$$\begin{aligned} S = \langle\, & x\colon \mathsf{GeoRegion};\quad y\colon \mathsf{Wine}; \\ & \exists z\,(\, z\colon \mathsf{WineGrower} \ \wedge \ z\, \mathsf{isLocatedIn}\, x \ \wedge \ z\, \mathsf{produces}\, y\,)\,\rangle \\ Q = \langle\, & x\colon \mathsf{GeoRegion};\quad y\colon \mathsf{Wine}; \\ & \exists z\,(\, z\colon \mathsf{Shop} \ \wedge \ z\, \mathsf{isLocatedIn}\, x \ \wedge \ z\, \mathsf{sells}\, y\,)\,\rangle \end{aligned}$$

The first description can be read as: given a object x of the type GeoRegion, the service returns all objects y of the type Wine such that *there exists* an object z which is a WineGrower, isLocatedIn the region denoted by x and produces the wine denoted by y. Similarly, the service Q' that takes a FrenchGeoRegion as input and returns the list of only FrenchWines produced in this region as output can be described as follows:

$$\begin{aligned} Q' = \langle\, & x\colon \mathsf{FrenchGeoRegion};\quad y\colon \mathsf{FrenchWine}; \\ & \exists z\,(\, z\colon \mathsf{WineGrower} \ \wedge \ z\, \mathsf{isLocatedIn}\, x \ \wedge \ z\, \mathsf{produces}\, y\,)\,\rangle \end{aligned}$$

In what follows, we will define the notion of service matching in such a way that the service S will not match the request Q (even though they have identical types of input and output), but S will match Q' (even though, in general, the outputs of S are not contained in those of Q').

Let us give a few more examples of services that can be described in this framework.

Example 3.1. The database of politicians. Suppose we have a database of politicians with a lot of information about them, say full dossier: their biographies, all results of their voting in parliament, their being members of committees and panels. A typical service that a user may be interested in is "given names of two politicians, return the list of bills they both voted for", (inputs: pairs of politicians $\langle x_1, x_2 \rangle$, outputs y comprise the set of all bills they voted in the same way). Another service would be: "give the list of all politicians, whose children are classmates in King Edward School" (no inputs, outputs $\langle y_1, y_2 \rangle$, pairs of politicians). The first service is a combination of invocations of a basic service "given a politician, return his voting results", the second is a combination of invocations of a basic service "given a politician, return the exact affiliations of all of his children".

Example 3.2. The database of residents in a block of apartments. A typical service would be "return the list of all residents in this block of flats with their family relations indicated" (no inputs, output an ABox in the language of, say, three roles: {parentOf, livesInFlat, marriedTo}.

Example 3.3. The database of hospital records of people in the UK. A typical service would be: "given a town or city in the UK, give the list of children between 5 and 11 that had flu this year". However, here we need a more general version of service descriptions, namely the ones that can take as inputs and return as outputs some values from concrete domains.

3.3 Service matching and subsumption

Now we develop the means of comparing the service descriptions, i.e., a notion of one service matching another. To keep exposition as easy and readable as possible, we first confine ourselves to the services having only one input and one output. As a matter of fact, these are the most typical services one can encounter. After that we will explain how to deal with services that have multiple inputs and outputs.

Since we develop our framework in the context of the automated service discovery problem, in the definitions of service match given below we will call one service (denoted by S) just service and another one (denoted by Q) a request. This is only for the sake of convenience; all definitions are applicable to any two service descriptions. Our task is to formulate reasonable conditions when a service S can be considered as an "appropriate" candidate to be returned by a search engine to a user who specified a request Q. We assume that service descriptions are formulated using an ontology (TBox) $\mathcal{T}$.

Definition 3.4 (Service matching; single input/output). Given two services with only one input and one output:

$$\begin{array}{rl} \text{Service:} & S = \langle x{:}\,X;\ \ y{:}\,Y;\ \ \Phi(x,y)\rangle, \\ \text{Request:} & Q = \langle z{:}\,Z;\ \ w{:}\,W;\ \ \Psi(z,w)\rangle, \end{array} \tag{1}$$

we say that the service S *matches* the request Q w.r.t. an ontology $\mathcal{T}$ if the following holds:

(i) Applicability: $\mathcal{T} \models X \sqsupseteq Z$. Intuitively, this means that all input data that the user intends to provide (i.e., the inputs of Q) are acceptable by the service S.

(ii) Coherence: *in any model of $\mathcal{T}$, the set of pairs $\langle e, d\rangle$ satisfying the conditions $e{:}\,Z$, $d{:}\,Y$, and $\Phi(e,d)$ coincides with the set of pairs $\langle e, d\rangle$ satisfying the conditions $e{:}\,Z$, $d{:}\,W$, and $\Psi(e,d)$.* In other words, the following two formulas with the free variables x and y:

$$\begin{array}{l} x{:}\,Z \ \wedge \ y{:}\,Y \ \wedge \ \Phi(x,y) \\ x{:}\,Z \ \wedge \ y{:}\,W \wedge \ \Psi(x,y) \end{array} \tag{2}$$

are equivalent w.r.t. $\mathcal{T}$, i.e., define the same binary relation in any model of $\mathcal{T}$. Note that the concept X does not occur here, since it is already used in condition **(i)**.

Intuitively, this means that, on any input that conforms to the user's request Q, the services S and Q return the same answers.

Condition **(i)** is quite standard; for example, it can be found in definitions for matching of OWL-S services (cf. [10]). In contrast, condition **(ii)** is–to the best of our knowledge–new, and it is not expressible in terms of OWL-S service profiles. As we will see in Section 4, this condition is in fact reducible to checking subsumption between two conjunctive queries w.r.t. a TBox, which is a standard reasoning task.

Now we generalise the notion of service matching to the case of multiple inputs and outputs. Here we will give a definition for services S and Q that have equal number of inputs and equal number of outputs. Other possibilities are considered in Section 5.

Definition 3.5 (Service matching; multiple inputs/outputs). Given two services:

$$\begin{array}{rl} \text{Service:} & S = \langle \vec{x} : \vec{X};\ \ \vec{y} : \vec{Y};\ \ \Phi(\vec{x}, \vec{y}) \rangle, \\ \text{Request:} & Q = \langle \vec{z} : \vec{Z};\ \ \vec{w} : \vec{W};\ \ \Psi(\vec{z}, \vec{w}) \rangle, \end{array} \tag{3}$$

with $|\vec{x}\,| = m = |\vec{z}\,|$ and $|\vec{y}\,| = n = |\vec{w}\,|$, we say that the service S *matches* the request Q w.r.t. the ontology $\mathcal{T}$ (in symbols: $\mathcal{T} \models S \colon Q$) if there exist two permutations

$$\begin{array}{l} \tau \colon \{1, \ldots, m\} \to \{1, \ldots, m\} \\ \sigma \colon \{1, \ldots, n\}\ \ \to \{1, \ldots, n\} \end{array}$$

such that the following two conditions hold:

(i) **Applicability:** $\mathcal{T} \models X_i \sqsupseteq Z_{\tau(i)}$, for all $i \leqslant m$, i.e., the type of x_i subsumes the type of $z_{\tau(i)}$ w.r.t. the ontology $\mathcal{T}$.

Intuitively, this means that one can map the inputs of S to inputs of Q so that all input data that the user intends to provide will be acceptable by S.

(ii) **Coherence:** for any model $\mathcal{I}$ of $\mathcal{T}$ and any tuples of elements[1] $\vec{e}, \vec{d}$ in $\mathcal{I}$ with $|\vec{e}\,| = m$ and $|\vec{d}\,| = n$, if $\mathcal{I} \models \vec{e} \colon \vec{Z}$ then the following equivalence holds:

$$\mathcal{I} \models \sigma(\vec{d}) \colon \vec{Y}\ \wedge\ \Phi(\tau(\vec{e}), \sigma(\vec{d})) \qquad \text{iff} \qquad \mathcal{I} \models \vec{d} \colon \vec{W}\ \wedge\ \Psi(\vec{e}, \vec{d}).$$

In other words, the following two formulas with free variables $\langle \vec{z}, \vec{w}\,\rangle$:

$$\begin{array}{l} \vec{z} \colon \vec{Z}\ \wedge\ \sigma(\vec{w}) \colon \vec{Y}\ \ \wedge\ \Phi(\tau(\vec{z}), \sigma(\vec{w})) \\ \vec{z} \colon \vec{Z}\ \wedge\ \ \ \ \ \vec{w} \colon \vec{W}\ \ \ \wedge\ \Psi(\vec{z}, \vec{w}) \end{array} \tag{4}$$

are equivalent w.r.t. $\mathcal{T}$, i.e., define the same relation in any model of $\mathcal{T}$. Again, the concepts X_i do not occur here, since they are already used in condition **(i)**.

Intuitively, condition **(ii)** means that, modulo some re-arrangement of the input and output vectors, the services Q and S return the same answers on any input that conforms to the user's request Q.

Some remarks are in order here. The need to permute inputs and outputs of a service S to "fit" the ones of Q (so that their types match accordingly) is by no means new—it is present in any reasonable definition of service matching. Thus, in order to check whether S matches Q, a reasoning system must "guess" two appropriate permutations τ and σ or exhaustively explore all possible ones.

The notion "S matches Q" is not symmetic (since we restrict the service S to inputs that conform to the request Q, not vice versa). This is justified by the fact that it is natural to count a service S as matching a request Q even if S can perform a more general task, but on inputs provided by a user it works exactly as user desired.

One would argue that this definition of service matching does not reflect the intuition behind it (and does not comply with the semantics of service description introduced in Definition 3.3). Indeed, a real service does not operate with arbitrary elements $\vec{e}$ of interpretations (i.e., models of an ontology). Instead, it accepts as inputs and returns as outputs only individual names, i.e.,

[1]The vector $\vec{z}^{\,\mathcal{I}} := \vec{e}$ plays the role of the input, and $\vec{w}^{\,\mathcal{I}} := \vec{d}$ the output of the requested service Q. Since $\vec{x} = \tau(\vec{z})$ and $\vec{y} = \sigma(\vec{w})$, the vector $\vec{x}^{\,\mathcal{I}} = \tau(\vec{e})$ can be considered as the input and $\vec{y}^{\,\mathcal{I}} = \sigma(\vec{d})$ as the output of the service S. Here $\tau(\vec{z})$ is the permutation of the vector $\vec{z}$, i.e., $\langle z_{\tau(1)}, \ldots, z_{\tau(m)} \rangle$.

named objects. Therefore, it would be more natural to formulate the service matching criterion that takes into account only the behaviour of services over individual names, not over arbitrary elements of a model. This is done in the definition given below. However, this definition is in fact equivalent to the one given above, as will be shown in Section 4.

Definition 3.6 (Service matching; ABox-based definition). Given two services:

$$\begin{array}{rl} \text{Service:} & S = \langle \vec{x} : \vec{X};\ \ \vec{y} : \vec{Y};\ \ \Phi(\vec{x}, \vec{y}) \rangle, \\ \text{Request:} & Q = \langle \vec{z} : \vec{Z};\ \ \vec{w} : \vec{W};\ \ \Psi(\vec{z}, \vec{w}) \rangle, \end{array} \tag{5}$$

with $|\vec{x}\,| = m = |\vec{z}\,|$ and $|\vec{y}\,| = n = |\vec{w}\,|$, we say that the service S *matches* the request Q w.r.t. the ontology $\mathcal{T}$ (in symbols: $\mathcal{T} \models S\colon Q$) if there exist two permutations

$$\begin{array}{l} \tau\colon \{1, \ldots, m\} \to \{1, \ldots, m\} \\ \sigma\colon \{1, \ldots, n\}\ \to \{1, \ldots, n\} \end{array}$$

such that the following two conditions hold:

(i) Applicability: $\mathcal{T} \models X_i \sqsupseteq Z_{\tau(i)}$, for all $i \leqslant m$. This condition is the same as in Definition 3.5.

(ii) Coherence: for any ABox $\mathcal{A}$ and any tuples of individuals $\vec{a}, \vec{b}$ in the knowledge base $\mathcal{KB} = \langle \mathcal{T}, \mathcal{A} \rangle$ with $|\vec{a}\,| = m$ and $|\vec{b}\,| = n$, if $\mathcal{KB} \models \vec{a} : \vec{Z}$, then the following equivalence holds:

$$\mathcal{KB} \models \sigma(\vec{b}) : \vec{Y}\ \wedge\ \Phi(\tau(\vec{a}\,), \sigma(\vec{b}\,)) \qquad \text{iff} \qquad \mathcal{KB} \models \vec{b} : \vec{W}\ \wedge\ \Psi(\vec{a}, \vec{b}\,).$$

Again, this means that, after some permutation of inputs and outputs, the services S and Q return the same answers on any input that conforms to the request Q.

Finally, we give a definition of service subsumption; since we have done much work above, this definition can be formulated easily. It can be based on either Definition 3.5 or Definition 3.6.

Definition 3.7 (Service subsumption). A service S *subsumes* (or is *more general than*) a request Q w.r.t. an ontology $\mathcal{T}$ (in symbols: $\mathcal{T} \models S \sqsupseteq Q$) if the following conditions hold: take condition **(i)** exactly as in Definition 3.5, and in condition **(ii)**, replace the equivalence by implication, namely, the second formula in (4) must imply the first one.

If we assert in (4) that the first formula implies the second one, then we say that S is *less general than* (or *more specific than*, or *subsumed by*) the request Q (in symbols: $\mathcal{T} \models S \sqsubseteq Q$).

Intuitively, S subsumes Q iff S accepts all the input data that Q does and on any such an input, the answers of S contain those of Q (and possibly some additional answers).

Clearly, for many users, a more general service may be equally relevant as exactly matching, and users would be interested to discover services that subsume their request. For example, after finding a service S that subsumes his request Q, a user may want to find another service that would extract the answers he needs from those returned by S.

The usefulness of the notion of service subsumption is also justified by the fact that it is unrealistic to assume that all web services are supplied with a precise description of their functionality. Rather, it will be typically the case that, from time to time, a service provider will supplement his service description with more and more information (e.g., he can add conjuncts into the formula Φ or modify the existing conjuncts) in order to increase the accuracy of the description. From this perspective, it is reasonable to count a service S as matching a request Q even if, according to the currently available description of S, it is more general than Q.

3.4 Automatic description of service composition

In this section, we will show that, from the service descriptions $S_1, \ldots, S_n$ that comply with Definition 3.2, we can automatically construct a description of the sequence of services $S_1 \circ \ldots \circ S_n$ that again complies with Definition 3.2. This is another important advantage of our approach, since it decreases the annotation workload for the web service provider. For the beginning, suppose that we are given two service descriptions in a repository, both having only one input and one output:

$$\begin{array}{l} S\ = \langle x\colon X;\quad y\colon Y;\quad \Phi(x, y) \rangle \\ S' = \langle x'\colon X';\ y'\colon Y';\ \Phi'(x', y') \rangle \end{array}$$

Our task is to formulate reasonable conditions when the composition of services $S \circ S'$ (to be read as 'first S runs, then S' runs on the output produced by S') is meaningful (i.e., when these services are compatible) and when it matches a user's request

$$Q = \langle z\colon Z;\ w\colon W;\ \Psi(z, w)\rangle.$$

First, let us give a definition in the style of Definition 3.5, i.e., as if services deal with elements of a model.

Definition 3.8 (Service composition; model-based definition). A composition of services $S \circ S'$ *matches* a request Q w.r.t. a TBox $\mathcal{T}$ if the following conditions hold:

(a) **Applicability:** $\mathcal{T} \models X \sqsupseteq Z$. This ensures that S accepts all inputs described in the request Q. The concept X will not occur in the remaining part of the definition.

(b) **Compatibility:** for any model $\mathcal{I}$ of $\mathcal{T}$ and any elements e, d in $\mathcal{I}$,

$$\text{if} \qquad \mathcal{I} \models e\colon Z \wedge \Phi(e, d) \wedge d\colon Y \qquad \text{then} \qquad \mathcal{I} \models d\colon X'.$$

This ensures that, if S runs on user's inputs, then its outputs are accepted by S'. The concept X' will not occur in the remaining part of the definition.

(c) **Coherence:** for any model $\mathcal{I}$ of $\mathcal{T}$ and any elements e, d in $\mathcal{I}$, if $\mathcal{I} \models e\colon Z$, then

$$\mathcal{I} \models \Psi(e, d) \wedge d\colon W \quad \Longleftrightarrow \quad \mathcal{I} \models \exists y\, \big(\, \Phi(e, y) \wedge y\colon Y \wedge \Phi'(y, d) \,\big) \wedge d\colon Y'.$$

This means that, on the user's inputs, the application of S and then S' yields the same answers as Q.

Conditions **(b)** and **(c)** can be rewritten without explicit referring to models:

(b) $\quad \mathcal{T} \models \forall x, y\, \big(\, x\colon Z \wedge \Phi(x, y) \wedge y\colon Y \ \to\ y\colon X' \,\big),$
(c) $\quad \mathcal{T} \models \forall x, w\, \big(\Psi(x, w) \wedge w\colon W \ \leftrightarrow\ \exists y\, \big(\, \Phi(x, y) \wedge y\colon Y \wedge \Phi'(y, w) \,\big) \wedge w\colon Y'\big).$

Observe that the notion of service composition is *request dependent* in the sense that the composition of services $S \circ S'$ is built for a particular request Q. This is so because, in general, the services S and S' may not be compatible—S may return outputs that S' cannot accept—yet on inputs that a user intends to provide, S returns only outputs acceptable by S'. Moreover, the definition given above suggests to assign to the composition of services the following description:

$$S \circ S' = \langle\, x\colon X;\ y'\colon Y';\ \exists t\, (\Phi(x, t) \wedge t\colon Y \wedge \Phi'(t, y'))\,\rangle,$$

provided that we have checked beforehand the **compatibility** of the services S and S' on the inputs of the service Q (i.e., condition **(b)** above). Indeed, conditions **(a)** and **(c)** in this case are precisely the same as conditions **(i)** and **(ii)** from Definition 3.4.

Now we give an analogue of this definition in the style of Definition 3.6, i.e., for services that deal with individuals from an ABox.

Definition 3.9 (Service composition; ABox-based definition). A composition of services $S \circ S'$ *matches* a request Q w.r.t. a TBox $\mathcal{T}$ if the following conditions hold:

(a) **Applicability:** $\mathcal{T} \models X \sqsupseteq Z$. This ensures that S accepts all inputs described in the request Q. The concept X will not occur in the remaining part of the definition.

(b) **Compatibility:** for any ABox $\mathcal{A}$ and any individuals a, b in $\mathcal{KB} = \langle \mathcal{T}, \mathcal{A}\rangle$,

$$\text{if} \qquad \mathcal{KB} \models a\colon Z \wedge \Phi(a, b) \wedge b\colon Y \qquad \text{then} \qquad \mathcal{KB} \models b\colon X'.$$

This ensures that, if S runs on user's inputs, then its outputs are accepted by S'. The concept X' will not occur in the remaining part of the definition.

Given two services:

$$S = \langle x{:}\,X;\quad y{:}\,Y;\quad \Phi(x,y)\rangle,$$
$$S' = \langle x'{:}\,X';\ y'{:}\,Y';\ \Phi'(x',y')\rangle,$$

in order to describe their composition $S \circ S'$ for matching to a request Q:

$$Q = \langle z{:}\,Z;\ w{:}\,W;\ \Psi(z,w)\rangle,$$

first check their compatibility on the inputs of Q:

$$\mathcal{T} \models \forall x\, \forall y\, \big(x{:}\,Z \wedge \Phi(x,y) \wedge y{:}\,Y \;\longrightarrow\; y{:}\,X'\big)$$

and then annotate the composition $S \circ S'$ as follows:

$$S \circ S' = \langle\, x{:}\,X;\ y'{:}\,Y';\ \hat{\exists} t\, (\Phi(x,t) \wedge t{:}\,Y \wedge \Phi'(t,y'))\,\rangle.$$

Figure 1: Automatic description of service composition.

(c) Coherence: for any ABox $\mathcal{A}$ and any individuals a, c in $\mathcal{KB} = \langle \mathcal{T}, \mathcal{A}\rangle$, if $\mathcal{KB} \models a{:}\,Z$, then

$$\begin{aligned}&\mathcal{KB} \models \Psi(a,c) \wedge c{:}\,W \quad \Longleftrightarrow\\ &\mathcal{KB} \models \Phi(a,b) \wedge b{:}\,Y \wedge \Phi'(b,c) \wedge c{:}\,Y', \quad \text{for some individual } b \text{ in } \mathcal{KB}.\end{aligned}$$

This means that, on the user's inputs, the application of S and then S' yields the same answers as Q.

In Section 4 we will show that although these two definitions of service composition are not equivalent to each other, they are both reducible to standard reasoning problems. The ABox-based definition suggests the way of describing a composition of services $S \circ S'$, as summarised in Figure 1. Therein, the symbol '$\hat{\exists} t$' denotes (informally) 'there exists an individual t'; how to define such an expression formally, and how to reason about it is discussed in [8]. Therein, it is shown that reasoning with this kind of quantifiers is reducible to query answering and subsumption, and thus decidable.

More generally, if we are given several services $S_1, \ldots, S_r$, each with one input and one output:

$$S_i \;=\; \langle\, x{:}\,X_i;\ y{:}\,Y_i;\ \Phi_i(x,y)\,\rangle,$$

then we can easily modify Definition 3.9 accordingly. Condition **(b)** will say that the outputs of S_1 (on user's inputs) are accepted by S_2, the outputs of S_2 (on inputs coming from S_1) are accepted by S_3, etc. Condition **(c)** will look as follows: for any ABox $\mathcal{A}$ and any individuals a, c in $\mathcal{KB} = \langle \mathcal{T}, \mathcal{A}\rangle$, if $\mathcal{KB} \models a{:}\,Z$, then $\mathcal{KB} \models \Psi(a,c) \wedge c{:}\,W$ iff, for some individuals $b_0, \ldots, b_r$ in $\mathcal{KB}$, where $b_0 := a$ and $b_r := c$, we have

$$\mathcal{KB} \models \bigwedge\nolimits_{i=1}^{r} \Phi_i(b_{i-1}, b_i) \wedge b_i{:}\,Y_i\,.$$

Finally, this notion of composition of services can be further generalised to the case of several services having multiple inputs and outputs. No fundamental difficulties arise here, but the notation becomes cumbersome due to permutations of variables.

4 Decidability of reasoning about services

Here we show that the problems of service matching and subsumption are reducible to the standard reasoning problem, namely to conjunctive query *subsumption* (also called query *containment*). We also briefly overview the state-of-the-art results on decidability and complexity of the query subsumption problem for different Description Logics.

4.1 Query subsumption

First we recall briefly the notions of a conjunctive query and query subsumption (see [7] for details and extensions).

Definition 4.1. A *conjunctive query* is an expression of the form

$$q(\vec{x}\,) \;\leftarrow\; \exists \vec{y}\, \big(t_1(\vec{x}\,,\vec{y}\,) \wedge \ldots \wedge t_k(\vec{x}\,,\vec{y}\,) \big),$$

where $\vec{x}$ and $\vec{y}$ are tuples of (*distinguished*, resp., *non-distinguished*) variables, and each term $t_i(\vec{x}\,,\vec{y}\,)$ is either $w{:}\,C$ or wRz, where C is a concept, R a role, and w, z are either are either variables from the lists $\vec{x}\,,\vec{y}$ or individual names. In other words, a conjunctive query is an existentially quantified conjunction of ABox assertions about individual names and variables.

In literature, one can find two different, but equivalent, definitions of query subsumption w.r.t. a TBox. The first one (adopted in [3, 4]) is based on the notion of evaluation of a query. Given an interpretation $\mathcal{I} = \langle \Delta, \cdot^{\mathcal{I}} \rangle$, a query $q(x_1, \ldots, x_m)$ is evaluated in $\mathcal{I}$ as the following subset of Δ^m:

$$q^{\mathcal{I}} := \{ \vec{e} \in \Delta^m \;\mid\; \mathcal{I} \models \exists \vec{y}\; \big(t_1(\vec{e}\,,\vec{y}\,) \wedge \ldots \wedge t_n(\vec{e}\,,\vec{y}\,) \big) \}.$$

Definition 4.2. A query $p(\vec{x}\,)$ *subsumes* a query $q(\vec{x}\,)$ w.r.t. a TBox $\mathcal{T}$ (written as $\mathcal{T} \models p \sqsupseteq q$) if, for any model $\mathcal{I}$ of $\mathcal{T}$, the inclusion $p^{\mathcal{I}} \supseteq q^{\mathcal{I}}$ holds. Two queries $p(\vec{x}\,)$ and $q(\vec{x}\,)$ are *equivalent* w.r.t. a TBox $\mathcal{T}$ (written as $\mathcal{T} \models p \doteq q$) if they subsume each other.

The second definition (cf. [1, p. 481]) is based on the idea that queries are used to retrieve information from knowledge bases, and so it is natural to say that a query p subsumes a query q if the answers of p *always* contain the answers of q (and possibly some additional ones). Here the word "always" is understood as "for any KB based on the given TBox". This definition involves the notion of the answer set of a query, let us recall it.

Definition 4.3. The *answer set* of a query $q(\vec{x}\,)$ over a knowledge base $\mathcal{KB}$ is denoted by $q(\mathcal{KB})$ and defined as the set of all tuples of individuals names that satisfy the query in all models of $\mathcal{KB}$:

$$q(\mathcal{KB}) \;:=\; \{\; \vec{a} \in \mathit{Const} \;\mid\; \mathcal{KB} \models q(\vec{a}\,) \;\}.$$

Definition 4.4. A query $p(\vec{x}\,)$ *subsumes* a query $q(\vec{x}\,)$ w.r.t. a TBox $\mathcal{T}$ (written as $\mathcal{T} \models p \sqsupseteq q$) if, for any ABox $\mathcal{A}$, the inclusion $p(\mathcal{T}, \mathcal{A}) \supseteq q(\mathcal{T}, \mathcal{A})$ holds.

Lemma 4.5. *Definitions 4.2 and 4.4 of query subsumption are equivalent.*

Proof. Suppose we are given a TBox $\mathcal{T}$ and two queries $p(x)$ and $q(x)$.
(Def.4.2 $\Rightarrow$ Def.4.4) Assume that $\mathcal{T} \models p \sqsupseteq q$ according to Definition 4.2. Take any ABox $\mathcal{A}$ and show that $p(\mathcal{T}, \mathcal{A}) \supseteq q(\mathcal{T}, \mathcal{A})$. Let $\vec{a} \in q(\mathcal{T}, \mathcal{A})$, and in order to see that $\vec{a} \in p(\mathcal{T}, \mathcal{A})$, take an arbitrary model $\mathcal{I} \models \mathcal{T} + \mathcal{A}$. By Definition 4.2, the inclusion $p^{\mathcal{I}} \supseteq q^{\mathcal{I}}$ holds. Since $\vec{a} \in q(\mathcal{T}, \mathcal{A})$, we have $\vec{a}^{\,\mathcal{I}} \in q^{\mathcal{I}}$, hence $\vec{a}^{\,\mathcal{I}} \in p^{\mathcal{I}}$, i.e., $\mathcal{I} \models p(\vec{a}\,)$. This holds for arbitrary model $\mathcal{I}$ of $\mathcal{T} + \mathcal{A}$, so we conclude that $\mathcal{T} + \mathcal{A} \models p(\vec{a}\,)$, i.e., $\vec{a} \in p(\mathcal{T}, \mathcal{A})$ and we are done.
(Def.4.2 $\Leftarrow$ Def.4.4) Assume that $\mathcal{T} \models p \sqsupseteq q$ according to Definition 4.4. Recall that $q(\vec{x}\,)$ has the form $\exists \vec{y} \bigwedge_{i=1}^{k} t_i(\vec{x}\,,\vec{y}\,)$. Now we introduce fresh individuals $\vec{a}$ and $\vec{b}$ with $|\vec{a}\,| = |\vec{x}\,|$ and $|\vec{b}\,| = |\vec{y}\,|$ and define the *canonical ABox* $\mathcal{A}_q$ for $q(\vec{x}\,)$ as

$$\mathcal{A}_q \;:=\; \{\; t_i(\vec{a}\,,\vec{b}\,) \;\mid\; 1 \leqslant i \leqslant k \;\}.$$

Obviously $\vec{a} \in q(\mathcal{T}, \mathcal{A}_q)$, since $\mathcal{T} + \mathcal{A}_q \models t_i(\vec{a}\,,\vec{b}\,)$ for all i. Hence $\vec{a} \in p(\mathcal{T}, \mathcal{A}_q)$.

To show that $\mathcal{T} \models p \sqsupseteq q$ holds according to Definition 4.2, consider an arbitrary model $\mathcal{I} \models \mathcal{T}$ and show that $p^{\mathcal{I}} \supseteq q^{\mathcal{I}}$. Take any tuple of elements $\vec{e} \in q^{\mathcal{I}}$, i.e., $\mathcal{I} \models q(\vec{e}\,)$, then there exists a tuple of elements $\vec{d}$ in $\mathcal{I}$ such that $\mathcal{I} \models t_i(\vec{e}\,,\vec{d}\,)$, for all i. Now extend the interpretation $\mathcal{I}$ to the constants $\vec{a}$ and $\vec{b}$ by putting $\vec{a}^{\,\mathcal{I}} := \vec{e}$ and $\vec{b}^{\,\mathcal{I}} := \vec{d}$. Then $\mathcal{I} \models t_i(\vec{a}\,,\vec{b}\,)$, for all i, and so $\mathcal{I} \models \mathcal{T} + \mathcal{A}_q$. Above we have shown that $\vec{a} \in p(\mathcal{T}, \mathcal{A}_q)$, so we conclude that $\mathcal{I} \models p(\vec{a}\,)$, or equivalently, $\vec{e} = \vec{a}^{\,\mathcal{I}} \in p^{\mathcal{I}}$. $\dashv$

Observe that the '$\Rightarrow$' implication in this lemma holds for any first-order formulas $p(\vec{x}\,)$ and $q(\vec{x}\,)$, whereas the converse implication is specific for conjunctive queries.

4.2 Deciding the service matching problem

Now we are ready to reduce the reasoning about services to the query subsumption problem.

Theorem 4.6 (Reduction). *The service matching (resp., service subsumption) problems w.r.t. an ontology are reducible to equivalence (resp., subsumption) of conjunctive queries w.r.t. a TBox.*

Proof. We consider only service matching; for service subsumption the proof is carried out analogously. Condition **(i)** in Definition 3.5 and 3.6 is the standard concept subsumption (which is a special case of query subsumption). Now consider condition **(ii)** in Definition 3.5. Recall that $\Phi(\vec{x}, \vec{y})$ in (4) has the form of a conjunctive query. If we write the existential quantifiers explicitly, then $\Phi(\vec{x}, \vec{y})$ has the form $\exists \vec{u}\; F(\vec{x}, \vec{y}, \vec{u})$. Similarly, $\Psi(\vec{z}, \vec{w})$ has the form $\exists \vec{v}\; G(\vec{z}, \vec{w}, \vec{v})$. Substituting these expressions into (4) and pushing all quantifiers outside, we obtain two conjunctive queries with $m + n$ distinguished variables. Therefore, the equivalence of the formulas from (4) in all models of $\mathcal{T}$ is the same as equivalence of these two queries w.r.t. $\mathcal{T}$ (according to Definitoin 4.2).

Similarly, condition **(ii)** in Definition 3.6 states precisely the equivalence of the same two queries w.r.t. $\mathcal{T}$ (according to Definitoin 4.4). Thus, the desired reduction consists of guessing the assignment (of variables $\vec{x}$ to some of $\vec{z}$ and $\vec{w}$ to some of $\vec{y}$) and then checking concept subsumptions in condition **(i)** and a query subsumption in condition **(ii)**. $\dashv$

In theory, this guessing step increases the complexity of service matching in comparison to that of query subsumption by the factor $m^m \times n^n$. However, in practice, the vast majority of services has small number of inputs and outputs. Besides that, this process is amenable to optimization. For example, when consecutively trying all possible permutations, if one observes that the type of x_1 does not subsume the type of z_1 (i.e., the subsumption $X_1 \sqsupseteq Z_1$ does not hold w.r.t. $\mathcal{T}$), then one can skip all other permutations that map x_1 to z_1.

4.3 Reasoning about service composition

Our aim in this section is to show that the matching problem for composite services introduced in Definitions 3.8 and 3.9 is reducible to the query subsumption and query answering problem. To this end, we need some preliminaries on reasoning about queries. Namely, we introduce a notion of a composition of conjunctive queries, which again comes in two versions – for queries that "deal" with elements in models or with individuals in ABoxes. Although these two definitions are not equivalent, we will show that the problem of checking the subsumption relationship between a composition of queries (for both definitions of a composition) and other queries is reducible to query subsumption or answering. Finally, we observe that Definitions 3.8 and 3.9 correspond special cases of the two definitions of query composition, and hence they are equivalent and both reducible to query subsumption or answering problem. This section is based on the technical results that are obtained in [8].

Suppose that we want to define the composition of conjunctive queries $q_1(x, y)$ and $q_2(y, z)$ with two distinguished variables. If we consider the queries as formulas that define, in each interpretation $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$, a binary relation $q_i^{\mathcal{I}}$ on the set $\Delta^{\mathcal{I}}$ (cf. Definition 4.2), then it is natural to define their composition as a formula that is interpreted in $\mathcal{I}$ as the composition of those binary relations. Clearly, the following definition achieves this goal.

Definition 4.7 (Query composition; model-based definition). The *composition* of conjunctive queries $q_1(x, y)$ and $q_2(y, z)$ is the following conjunctive query:

$$(q_1 \circ q_2)(x, z) \;:=\; \exists y\, (\, q_1(x, y) \wedge q_2(y, z)\,).$$

According to another approach, queries are used to retrieve individuals that satisfy certain conditions in (all models of) a knowledge base. In this setting, the composition must be defined in such a way that, given a knowledge base, $q_1 \circ q_2$ retrieves a pair of individuals $\langle a, c\rangle$ if and only if there is an *individual* b in the knowledge base such that $\langle a, b\rangle$ satisfies q_1 and $\langle b, c\rangle$ satisfies q_2. This gives rise to the following definition.

Definition 4.8 (Query composition; ABox-based definition). The answer set of the *composition* of conjunctive queries $q_1(x, y)$ and $q_2(y, z)$ over a knowledge base $\mathcal{KB}$ is defined as follows:

$$(q_1 \mathbin{\hat{\circ}} q_2)(\mathcal{KB}) := \{\langle a, c\rangle \mid \text{ there is an individual } b \text{ in } \mathcal{KB} \text{ such that } \mathcal{KB} \models q_1(a, b) \wedge q_2(b, c)\}.$$

The two definitions just given are not equivalent: the inclusion $(q_1 \hat{\circ}\, q_2) \sqsubseteq (q_1 \circ q_2)$ always holds, but the converse inclusion does not hold in general, as follows from the results obtained in [8] (see item (0) of Theorem 4.2 there). However, what we are interested in is not the composition of queries *per se*, but rather in deciding the subsumption relationship between the composition and other queries. The following theorem, which is a corollary of the above cited theorem, give a complete answer to these issues.

Theorem 4.9 (Subsumption of composite queries). *Let $q_1(x, y)$, $q_2(y, z)$, and $p(x, z)$ be conjunctive queries and $\mathcal{T}$ a TBox. Then the following conditions hold:*

(1) $\mathcal{T} \models q_1 \hat{\circ}\, q_2 \sqsubseteq p$ iff $\mathcal{T} \models q_1 \circ q_2 \sqsubseteq p$.

(2) An analogous statement does not hold for the converse subsumption '$\sqsupseteq$'. But the problem $\mathcal{T} \models p \sqsubseteq q_1 \hat{\circ}\, q_2$ is reducible to query answering.

Proof. Take $q(x, y, z) := q_1(x, y) \wedge q_2(y, z)$. Then $q_1 \circ q_2 \equiv \exists y\, q(x, y, z)$ and $q_1 \hat{\circ}\, q_2 \equiv \hat{\exists} y\, q(x, y, z)$. Now apply Theorem 4.2 from [8]. ⊣

Now we can apply the results obtained. Items **(a)** in both Definitions 3.8 and 3.9 are concept subsumption, items **(b)** in both definitions have the form of query subsumption, Finally, items **(c)** in both definitions have the form "$q_1 \circ q_2 \doteq p$" and "$q_1 \hat{\circ}\, q_2 \doteq p$", resp., for the following queries:

$$\begin{array}{rcl} p(x,z) & := & x{:}\, Z \wedge \Psi(x,z) \wedge z{:}\, W, \\ q_1(x,y) & := & x{:}\, Z \wedge \Phi(x,y) \wedge y{:}\, Y, \\ q_2(y,z) & := & \Phi'(y,z) \wedge z{:}\, Y', \end{array}$$

and thus reducible to query subsumption or answering, by Theorem 4.9.

4.4 An overview of the complexity of query subsumption

Here we will briefly discuss results on the decidability and complexity of query containment w.r.t. ontologies. In general, query containment is at least as hard as concept subsumption or satisfiability. Hence, the lower bound for its complexity immediately follows from the lower complexity bound of a Description Logic itself.

As for upper bounds, there are several results in this direction. In [3] the query containment problem for the Description Logic $\mathcal{DLR}_{reg}$ was explored. This is a logic with n-ary relations and boolean operations on them, and regular expressions for binary relations. The logic itself is ExpTime-complete, whereas the upper bound for query containment is shown to be exponential in size of TBox and double exponential in size of queries. To be more exact, the problem of checking that $\mathcal{T} \models q \sqsubseteq p$ is shown to have the complexity $\exp(\text{polynom}(|\mathcal{T}| \times |q|^{|p|}))$. In that paper, however, only so-called simple properties are allowed in query terms. Hence it does not completely suite our setting: most of our example services involve a transitive and thus non-simple property `hasPart`. It is also proved there that if we allow inequality in queries, then the problem becomes undecidable.

In [4], the query containment problem for the logic $\mathcal{DLR}$ was reduced to checking ABox satisfiability for the same logic, which, in turn, was reduced to knowledge base satisfiability for the DL $\mathcal{SHIQ}$. The latter problem is successfully solved using an implemented highly optimised reasoner FaCT. However, it suffers from the same restriction to simple properties in queries.

In a recent paper [2], the complexity of query answering and query containment was explored. For many fragments of expressive Description Logics, the *data complexity* was studied, i.e., the complexity of the query answering in size of ABox. Numerous results (from LogSpace to coNP) were presented for queries with possibly non-simple properties.

Continuing this line of research, in [11] the upper bound for data complexity of query answering is established for the DL $\mathcal{SHIQ}$, namely, it is shown that this problem is coNP in the size of the ABox. As for the complexity of query containment, the results obtained in [11] for the logic $\mathcal{SHIQ}$ state the following: if the underlying knowledge base has no transitive relations, then the complexity is 3coNExpTime in size of the knowledge base; otherwise there are some difficulties in obtaining the complexity estimates for query answering.

There is ongoing work on obtaining tight upper complexity bounds for various Description Logics. In particular, for the DL $\mathcal{SHOIQ}$, which is the underlying logic of the W3C recommended web ontology language OWL-DL, the upper complexity bounds are investigated, as well as tableaux algorithms are been devised.

Summing up, service matching w.r.t. OWL ontologies is known to be decidable, and decision procedures for this problem are available. Yet, to the best of our knowledge, neither tight complexity bounds nor an implementation are currently available.

5 Generalisations of service matching

First of all, we give a more general definition of service matching, which is applicable to the case when some inputs in the user's request Q and/or some outputs of the service S are "redundant". We assume that, when invoking a service, a user will not need to "convert" inputs and outputs, or build a complex input from simpler ones, or extract a simpler output from a complex one. In other words, inputs and outputs are passed between a user and a service "as is". This justifies the assumption, in definitions below, that the number of inputs of the service S is not greater than the number of inputs of the service Q, whereas the number of outputs of the service S is not less than the number of inputs of the service Q. How to deal with cases when these inequalities do not hold is discussed later in this section.

Definition 5.1 (Service matching, redundant inputs/outputs). Given two services:

$$\begin{array}{rl} \text{Service:} & S = \langle \vec{x} : \vec{X};\ \ \vec{y} : \vec{Y};\ \ \Phi(\vec{x}, \vec{y})\rangle, \\ \text{Request:} & Q = \langle \vec{z} : \vec{Z};\ \ \vec{w} : \vec{W};\ \ \Psi(\vec{z}, \vec{w})\rangle, \end{array} \tag{6}$$

with $|\vec{x}\,| \leqslant |\vec{z}\,|$ and $|\vec{y}\,| \geqslant |\vec{w}\,|$, we say that the service S *matches* the request Q w.r.t. an ontology $\mathcal{T}$ (in symbols: $\mathcal{T} \models S\colon Q$) if there exist two injective mappings

$$\begin{array}{l} \tau\colon \{1, \dots, |\vec{x}\,|\} \to \{1, \dots, |\vec{z}\,|\} \\ \sigma\colon \{1, \dots, |\vec{w}|\} \to \{1, \dots, |\vec{y}\,|\} \end{array}$$

such that if we denote

$$\begin{array}{ll} \vec{z}\,' = \langle z_{\tau(1)}, \dots, z_{\tau(m)}\rangle, & \vec{z}\,'' = \vec{z} \setminus \vec{z}\,', \\ \vec{y}\,' = \langle z_{\sigma(1)}, \dots, z_{\sigma(n)}\rangle, & \vec{y}\,'' = \vec{y} \setminus \vec{y}\,', \end{array}$$

then the following two conditions hold:

(i) Applicability; $\mathcal{T} \models X_i \sqsupseteq Z_{\tau(i)}$, for all $i \leqslant |\vec{x}\,|$, i.e., the type of x_i subsumes the type of $z_{\tau(i)}$ w.r.t. the ontology $\mathcal{T}$.

This means that one can assign each input x_i of S to some input $z_{\tau(i)}$ of Q in such a way that all input data that the user intends to provide will be acceptable by S. Since $|\vec{x}\,| \leqslant |\vec{z}\,|$, such an assignment τ is possible combinatorically. In particular, here one determines those inputs $\vec{y}\,'$ in the request Q that will be submitted to the service S, and the "redundant" ones $\vec{y}\,''$.

(ii) Coherence: The following two formulas with free variables $\langle \vec{x}, \vec{y}\,'\rangle$ and $\langle \vec{z}\,', \vec{w}\,\rangle$, resp.:

$$\begin{array}{l} \exists \vec{y}\,''\,(\,\vec{x} : \vec{Z}' \ \wedge\ \Phi(\vec{x}, \vec{y}\,)\ \wedge\ \vec{y} : \vec{Y}\,) \\ \exists \vec{z}\,''\,(\,\vec{z} : \vec{Z}\ \wedge\ \Psi(\vec{z}, \vec{w}\,) \wedge\ \vec{w} : \vec{W}) \end{array} \tag{7}$$

are equivalent w.r.t. $\mathcal{T}$, i.e., represent the same relation in any model of $\mathcal{T}$, if we identify the vector of variables $\vec{x} = \langle x_1, \dots, x_m\rangle$ with $\vec{z}\,' = \langle z_{\tau(1)}, \dots, z_{\tau(m)}\rangle$ and $\vec{y}\,' = \langle z_{\sigma(1)}, \dots, z_{\sigma(n)}\rangle$ with $\vec{w} = \langle w_1, \dots, w_n\rangle$. Again, the concepts X_i do not occur here, since they are already used in condition **(i)**.

Intuitively, condition **(ii)** means that, after dropping "redundant" inputs of the request Q and "redundant" outputs of the service S and permuting the remaining inputs and outputs, the services Q and S will return the same answers on any input that conforms to the request Q. Note that we get rid of the "redundant" inputs and outputs by existentially quantifying over them, as can be seen from formulas (7).

Now we consider the remaining cases for the numbers of inputs and outputs. Suppose that $|\vec{x}\,| > |\vec{z}\,|$, i.e., the service S requires more input arguments than the user Q is able to provide. Then a possible solution is to "instantiate" some inputs of S with constants (individual names) and thus decrease the number of inputs required by S. This discussion leads to the following notion.

Definition 5.2 (Service instance). Given a service $S := \langle \vec{x} : \vec{X}; \vec{y} : \vec{Y}; \Phi(\vec{x}, \vec{y}) \rangle$, a partition of its inputs[2] $\vec{x} = \vec{x}\,'\vec{x}\,''$, and a tuple of individual names $\vec{a}$ of the length $|\vec{a}| = |\vec{x}\,'|$ such that $\mathcal{KB} \models \vec{a} : \vec{X}'$ ("type matching" condition), the following service is called an *instance* of the service S:

$$S[\vec{a}/\vec{x}\,'] := \langle \vec{x}\,'' : \vec{X}''; \quad \vec{y} : \vec{Y}; \quad \Phi(\vec{a}, \vec{x}\,'', \vec{y}) \rangle.$$

If a user submits a service request Q to a search engine (SE), and the SE finds in repository a service S that has more inputs than Q, then the SE can try to instantiate S with some individual names. Afterwards, the SE verifies whether the service instance obtained in this way matches (or subsumes or is subsumed by) the service request Q.

Example 5.1. Suppose that the service S takes as inputs the name of a UK citizen and a city of his/her residence, and returns the names of all his/her children. The description of this service is:

$$\begin{aligned} S = \langle\, & x_1 : \mathsf{Person},\ x_2 : \mathsf{City} \sqcap \exists\, \mathsf{isLocatedIn.UK}; \\ & y : \mathsf{Person};\ x_1\ \mathsf{livesIn}\ x_2\ \wedge\ x_1\ \mathsf{hasChild}\ y \,\rangle. \end{aligned}$$

A user requests for a service Q that retrieves the names of children of any women living in Liverpool, and he describes his service as follows (the user did not specify the type of output, and the system takes $\top$ as a default type):

$$Q = \langle\, z : \mathsf{Woman} \sqcap \exists\, \mathsf{livesIn.Liverpool}; \quad w : \top; \quad z\ \mathsf{hasChild}\ w \,\rangle.$$

Obviously, the service S is suitable for user's purposes, even though it has more inputs than the user is able to provide. We will show now that notion of service instance we introduced works properly in this situation.

We assume that, in the background ontology $\mathcal{KB}$, the concept Woman is defined as $\mathsf{Person} \sqcap \mathsf{Female}$, there is an axiom $\mathsf{Person} \sqsubseteq \forall\, \mathsf{hasChild.Person}$, and the assertions $\mathsf{Liverpool} : \mathsf{City}$ and $\mathsf{Liverpool}$ $\mathsf{isLocatedIn}$ UK are stored in the ABox of the $\mathcal{KB}$. We also need a concept inclusion axiom

$$\exists \mathsf{livesIn}.\exists \mathsf{isLocatedIn.UK} \sqsubseteq \exists \mathsf{livesIn.UK}$$

or a more general *complex role inclusion* axiom:

$$\mathsf{livesIn}\ \circ\ \mathsf{isLocatedIn}\ \sqsubseteq\ \mathsf{livesIn}.$$

A search engine, using its reasoning system, infers from these facts that $\mathcal{KB} \models \mathsf{Liverpool} : X_2$. Then the SE "instantiates" the variable x_2 in S with the constant $\mathsf{Liverpool}$ and obtains a service $S' := S[x_2/\mathsf{Liverpool}]$ having only one input x_1. Afterwards, the SE verifies that the service obtained in this way indeed matches the request Q, namely:

(i) $\mathcal{KB} \models X_1 \sqsupseteq Z$, i.e. that Person subsumes $\mathsf{Woman} \sqcap \exists\, \mathsf{livesIn.Liverpool}$;

(ii) if we substitute $\mathsf{Liverpool}$ for x_2 in S, then the equivalence of the corresponding formulas in (7) becomes almost trivial.

After all these conditions are verified, a SE returns S as an "appropriate" service to a user, together with informing him which inputs and outputs of S correspond to inputs and outputs of Q, as well as how to instantiate extra inputs of S.

Another option (in the same case when $|\vec{x}| > |\vec{z}|$) is to merge some inputs of S, provided that their types are compatible. The inputs x_i and x_j have *compatible* types if the concept $X_i \sqcap X_j$ is satisfiable w.r.t. the ontology $\mathcal{T}$. The corresponding definition is rather easy to formulate, and we leave the details to the reader.

Next, suppose that $|\vec{y}| < |\vec{w}|$, i.e. the service S has less outputs than a user desired. In this situation, it is reasonable that a SE does not reject such a service S, but tries to find out whether S produces at least part of outputs suitable for user. To be more exact, SE can verify whether S matches (or subsumes) a service Q' obtained from Q by dropping some its outputs (but still preserving their relation to inputs and remaining outputs). In this case we will say that a service S *partially* matches (resp., subsumes) a query Q. These notions are based on the following one.

[2]Although this notation does not reflect the order of variables (the variables $\vec{x}\,'$ can alternate with $\vec{x}\,''$ in the list $\vec{x}$), we use it for simplicity.

Definition 5.3 (Subservice). Given a service $Q = \langle \vec{z} : \vec{Z}; \vec{w} : \vec{W}; \Psi(\vec{z}, \vec{w}) \rangle$, and a partition of its outputs $\vec{w} = \vec{w}'\vec{w}''$, the following service is called a *subservice* of Q:

$$Q[\exists/\vec{w}''] := \langle \vec{z} : \vec{Z}; \quad \vec{w}' : \vec{W}'; \quad \exists \vec{w}''(\vec{w}'' : \vec{W}'' \wedge \Psi(\vec{z}, \vec{w})) \rangle.$$

All the definitions given in this section do not introduce any new reasoning problem. Rather, they are reduced to service matching and subsumption defined earlier. The reduction consists in guessing of a subtuple of inputs of S that should be replaced by constants (and finding suitable constants), or in finding a subtuple of outputs of Q that can be produced by a service S. A priori, we need to look through an exponential number of subtuples of a tuple of inputs (or outputs). However, since we are concerned with *typed* inputs and outputs, this can decrease an algorithm complexity drastically, for, in general, only a small number of inputs or outputs of two services can correspond to each other w.r.t. their types.

6 Extentions to service description

In the subsequent sections, we consider several extensions to the service description framework introduced above. Although these extensions are considered separately, this is done only for simplicity purposes, and one can readily combine various extensions together. This is mainly due to the fact that reasoning about these extra items in service descriptions is usually carried out independently from each other.

6.1 Services with structured outputs

Recall that, according to Definition 3.3, a service returns a *set* of outputs satisfying some conditions. However, a realistic web service does not behave like this; instead, it returns output values (tuples) successively one by one, or returns a file (or table, etc.) containing the output values in some order. This order can be either arbitrary (depending on some occasional circumstances, like order of the data stored in a database, or the order in which other invoked services have replied, etc.) or a predefined order w.r.t. some (linearly ordering) relation (say, a list of names in alphabetic order, or a list of goods from cheapest to more expensive, etc.). There can be imagined non-linearly ordered data types like trees and multi-dimensional arrays. Here we generalize the notion of service description to (partially) capture this expressivity.

Definition 6.1 (Service with structured output). A *service with structured output* is a formal expression of the form

$$S := \langle \vec{x} : \vec{X}; \; \vec{y} : \vec{Y}; \; \Phi(\vec{x}, \vec{y}); \; \vec{A}; \; \vec{R} \rangle,$$

where, in addition to ordinary service (cf. Definition 3.2), we have finite lists of concept names $\vec{A} = \langle A_1, \ldots, A_k \rangle$ and role names $\vec{R} = \langle R_1, \ldots, R_\ell \rangle$.

The semantics of this description is that, given a tuple of individuals $\vec{a}$ of the type $\vec{X}$, the service returns as its output the set of all tuples $\vec{b}$ of individuals that satisfy the condition $\vec{b} : \vec{Y} \wedge \Phi(\vec{a}, \vec{b})$ and, additionally, the service returns the *full atomic diagram* of the set Obj in the language $\{\vec{A}, \vec{R}\}$, where Obj is the set of all individuals occurring among the returned tuples $\vec{b}$. In other words, the service provides the answers, for each individual in Obj, whether it is an instance of A_i (for every i) and, for each pair of those individuals, whether they are in R_j relation (for every j).

Example 6.1. A service returns a set of (names of) all children of an employee of a certain company, and $\vec{R}$ consists of the only one role name $\mathsf{isYoungerThan}$ (or, $\mathsf{AlphabeticallyPreceds}$). We know that this relation is a linear order (and we assume that the background ontology contains the corresponding axioms). Therefore we can say that a service returns a set of (names of) children together with the linear order on them according to their age (resp., alphabetical order). Given this kind of output, a user (or a shim-service) can readily reorder the output data of the service into a list or array or any other data type suitable for a user in accordance with that linear order. From this perspective, we do not need to explicitly consider these numerous data types and translations between them, and leave it to an implementation level.

The notion of service matching is naturally extended to this kind of services as follows.

Definition 6.2 (Structured service matching). Given two services with structured outputs:

$$\begin{aligned} S &= \langle \vec{x} : \vec{X};\ \ \vec{y} : \vec{Y};\ \ \Phi(\vec{x}, \vec{y});\ \ \vec{A};\ \ \vec{R} \rangle, \\ Q &= \langle \vec{z} : \vec{Z};\ \ \vec{w} : \vec{W};\ \Psi(\vec{z}, \vec{w});\ \ \vec{B};\ \ \vec{P} \rangle, \end{aligned} \tag{8}$$

with $|\vec{x}\,| \leqslant |\vec{z}\,|$ and $|\vec{y}\,| \geqslant |\vec{w}\,|$, we say that the service S *matches* the request Q w.r.t. an ontology $\mathcal{T}$ if conditions **(i)**, **(ii)** from Definition 5.1 hold together with the following condition:

(iii) there exist two injections

$$\begin{aligned} \alpha &: \{1, \dots, |\vec{B}|\} \to \{1, \dots, |\vec{A}|\} \\ \beta &: \{1, \dots, |\vec{P}|\} \to \{1, \dots, |\vec{R}|\} \end{aligned}$$

such that $\mathcal{T} \models B_i \doteq A_{\alpha(i)}$ and $\mathcal{T} \models P_j \doteq R_{\beta(j)}$, for all $i \leqslant |\vec{B}|$ and $j \leqslant |\vec{P}|$.

Intuitively: each additional concept name B_i and role name P_j in Q is equivalent to some concept name and role name from $\vec{A}$ and $\vec{R}$ in S. In particular, we have that $|\vec{A}| \geqslant |\vec{B}|$ and $|\vec{R}| \geqslant |\vec{P}|$, i.e., we allow the output of the service S to be "more structured" than it was required in the user's request Q.

6.2 Services with boolean outputs

Sometimes it can be too resource consuming for a service to return the whole set of output values satisfying some conditions. Instead, for a given input, a service may evaluate a certain formula and return a boolean value. An important class of services of this kind is the family of services, in which these formulas are conjunctive queries (see Definition 4.1).

Definition 6.3 (Boolean service). A *service with boolean outputs* (or *boolean service*, for short) is a formal expression of the form

$$S := \langle \vec{x} : \vec{X};\ \ \vec{q}\,(\vec{x}\,) \rangle,$$

where $\vec{x} : \vec{X}$ is a typed tuple of input variables, and $\vec{q}\,(\vec{x}\,) = \langle q_1(\vec{x}\,), \dots, q_k(\vec{x}\,) \rangle$ is a list with each $q_j(\vec{x}\,)$ being a conjunctive query with the distinguished variables among $\vec{x}$.

The semantics of such a description is that, for an input $\vec{a}$ of the type $\vec{X}$, the service S returns a tuple of boolean values corresponding to the queries $\vec{q}\,(\vec{a}\,)$.

Note that we have dropped all other ingredients of a service description here, since one can easily combine various extensions of the basic notion of service (see Definition 3.2), as we already mentioned at the beginning of Section 6.

Definition 6.4 (Boolean service matching). Given boolean services $S := \langle \vec{x} : \vec{X}; \vec{q}\,(\vec{x}\,) \rangle$ and $Q := \langle \vec{z} : \vec{Z}; \vec{p}\,(\vec{z}\,) \rangle$ with $|\vec{x}\,| = |\vec{z}\,|$ and $|\vec{p}| \leqslant |\vec{q}\,|$, we say that the service S *matches* the request Q w.r.t. an ontology $\mathcal{T}$ if condition **(i)** from Definition 3.5 holds and there exists an injection $\pi : \{1, \dots, |\vec{p}|\} \to \{1, \dots, |\vec{q}\,|\}$ such that $\mathcal{T} \models p_i(\tau(\vec{z})) \doteq q_{\pi(i)}(\vec{z})$ for all $i \leqslant |\vec{p}|$, where τ is the permutation from condition **(i)**.

Intuitively, modulo permutations of inputs and outputs and after removing "redundant" outputs of S, the services S and Q return the same answers on each input that conforms to the request Q.

Clearly, this definition does not introduce any new reasoning problems – the problem of matching boolean services is reducible to query subsumption w.r.t. an ontology.

7 Related work

In this section we compare the service description formalism presented in this paper to other approaches to the service discovery problem, namely the ones based on OWL-S and WSMO. We argue that, on the one hand, our framework is compatible with them and can be incorporated into them with little effort; on the other hand, it adds extra expressivity in description of functionality of web services, without need to increase the expressivity of the underlying language.

7.1 OWL-S Service Matchmaking

To be done.

7.2 WSMO Web Service Discovery

The service discovery framework based on the Web Service Modeling Ontology (WSMO) is described in [12] (with implementation issues considered in the unfinished work [13]). To align our framework to WSMO, one should keep in mind the difference in terminology: in [12], a service advertisement and request are referred to as a *service* and a *goal*.

Furthermore, they make a strict distinction between *service discovery* and *web service discovery*; we recall it briefly by virtue of an example. If a customer wants to travel from one city to another, then he is looking for a *service* (an airline, a train, or a bus company) that can provide this to him. For this aim, a customer is looking for a *web service* that can help him to find that service (e.g., by providing information or allowing to buy a ticket, etc.). The latter search is based on semantic annotation of web services. The principal consequence of such a distinction is that the semantic annotation of a web service should not contain the exhaustive information about all the services (or databases, or knowledge bases) it allows to access. In our example, the description of a web service need not contain the complete information about available departures, destinations, times, prices, etc. To put it in another way, web service discovery should not try to replace or duplicate the functionality of the services (or databases).

Both the WSMO approach and our framework comply to this requirement. Indeed, we do not include a complete set of pairs "input–output" into a service description, but rather specify, in a generic way, the relationship between inputs and outputs of a web service. On the contrary, it is pointed out in [12] that the approach proposed in the paper [14] assumes a superfluously detailed (neither realistic, nor desirable) annotation of web service.

In [12], three approaches to service discovery are considered, which require different effort in annotation of services and requests and deliver discovery results of different accuracy. The basic one – the *keyword-based* discovery – briefly considered there and is not of much interest to us as well. The other two are based on so called "simple" and "rich" semantic description of services.

Simple Semantic Description of Services

A simple semantic description of a web service or a request consists of two components:

- (a description of) a set of *relevant objects*;
- an *intention*, which can be either existential (∃) or universal (∀).

Relevant objects are objects that a service provides information about, or objects that a user wants to know about. This set can be described by a DL concept or, more generally, by a formula in some (e.g., first order) language. For example, if a service provides (or a user wants to find) information about flights between European cities, then the set of relevant objects is (cf. [12, p. 16])

$$\{\ f \ \mid\ \exists s, e \left(\mathsf{flight}(f, s, e)\ \wedge\ \mathsf{isLocatedIn}(s, \mathsf{Europe})\ \wedge\ \mathsf{isLocatedIn}(e, \mathsf{Europe}) \right)\ \},$$

where $\mathsf{flight}(f, s, e)$ is the relation "f is a flight from location s to location e". Notably, the formula in this example has the form of conjunctive query. But this is only a coincidence, as one can easily imagine a service that provides information about flights *or* train services.

From the exposition in [12], however, it is not clear how the set of relevant elements is related to the outputs of a service (but these notions are definitely not the same, in general), and there is no clear explanation of how one can systematically assign a set of relevant objects to a service. For instance, what is the set of relevant objects for the service "given a name of a region, return the list of wines that are produced in that region"? Apparently, it is not the class Wine, nor GeoRegion. The most relevant would be the set of pairs $\langle r, w\rangle$ with r being a region and w a wine that is produced in r. But this is precisely how we propose to describe services in our framework: this set of pairs is specified by the formula $\Phi(r, w)$. More interesting question is: how could one assign relevant objects (other than pairs) to services so that to distinguish the previous service from the following: "given a name of a region, return the list of wines that are sold in that region"?

If we forget about the second component (intention) in service descriptions, then services are matched by checking the equality (or inclusion, or intersection) of the sets of relevant objects specified by a service provider and requester. The intention component was introduced in order to add extra flexibility in describing services or requests. If a provider advertises his web service with the intention $\forall$ (resp., $\exists$), this means that he guarantees that the service delivers information about *all* (resp., *some*, but not necessarily all[3]) objects from the set of relevant objects. Similarly, if a user formulates his request with the intention $\forall$ (resp., $\exists$), then he needs an information about *all* (resp., *some*, but not necessarily all) relevant objects. In the case of a request with the intention $\exists$, the user will be satisfied if he finds a service that delivers information about at least one relevant object, whereas the intention $\forall$ indicates that he will be satisfied only after finding a service (or a group of services) that provide (altogether) information about all relevant objects.

The matching conditions that take the intention component into account can be built on top of a "usual" matching condition in a straightforward way. In [12], this is done explicitly for the case of simple service descriptions;[4] for "rich" descriptions (see below), it is said that it can be done analogously. Several kinds of the notion of match are considered in the cited paper, where the intention component plays an important role. Namely one can distinguish between *exact match* (service delivers all the requested information), *subsumption match* (service delivers only part of the requested information, and nothing "irrelevant" to the request), *plugin match* (service delivers all the requested information and, possibly, some irrelevant information), and *intersection match* (service delivers at least some of the requested information and, possibly, some irrelevant one).

Let us show that the intention component can be naturally built into service descriptions within our framework. Definition 3.2 of a service is extended by a new component $\alpha \in \{\exists, \forall\}$. In Definition 3.3, we say (in item 2) that a service s implements a description S if it returns all (for the case $\alpha = \forall$) or some (for the case $\alpha = \exists$) tuples $\vec{b}$ that satisfy the condition $\mathcal{KB} \models \vec{b}:\vec{Y} \wedge \Phi(\vec{a}, \vec{b})$. Now consider two service descriptions from Definition 3.4 enriched with intensions:

$$\begin{array}{rl} \text{Service:} & S = \langle\, x\colon X;\;\; y\colon Y;\;\; \Phi(x,y);\;\; \alpha\,\rangle, \\ \text{Request:} & Q = \langle\, z\colon Z;\;\; w\colon W;\;\; \Psi(z,w);\;\; \beta\,\rangle, \end{array} \tag{9}$$

where $\alpha, \beta \in \{\exists, \forall\}$. Definition 3.4 itself can be regarded as giving the matching conditions for the case $\alpha = \beta = \forall$. If $\beta = \exists$, then (whatever α is) the service S matches the request Q if, on any input of Q, all answers of S are contained in answers of Q, i.e., when S is less general than Q according to Definition 3.7. Finally, in the case $\alpha = \exists$ and $\beta = \forall$, we cannot guarantee a perfect match, and we can only say that S partially matches Q (i.e., the subsumption match holds here).

Rich Semantic Description of Services

As opposed to the simple descriptions presented above, where a service characterised by a single set, here the relationship between inputs and outputs is taken into account. Hence these rich descriptions are closer to our framework. However, a service request is still modelled as a single set (the desired information).

According to [12], a *rich semantic description* of a web service with n inputs and one output (for several outputs, the definition extends in a straightforward way) consists of two components:

- an $(n+1)$-ary predicate $w(\vec{x}, y)$;
- an *intention*, which can be either existential ($\exists$) or universal ($\forall$).

[3]This may happen if, for instance, a service is supplied with a incomplete (temporary) annotation, which is subject to further refinements.

[4]By considering 5 different relationships between the sets of objects relevant to a request ($\mathcal{R}$) and to a service ($\mathcal{W}$), namely equality ($\mathcal{R} = \mathcal{W}$), inclusions ($\mathcal{R} \subseteq \mathcal{W}$ or $\mathcal{R} \supseteq \mathcal{W}$), intersection ($\mathcal{R} \cap \mathcal{W} \neq \varnothing$) and disjointness ($\mathcal{R} \cap \mathcal{W} = \varnothing$), and 4 different combinations of intensions. In total, 20 combinations are considered.

The meaning of $w(\vec{x}\,, y)$ is that the value y will be delivered by the web service if it is invoked with the input values $\vec{x}$. In the simplest case, it can be given by a single formula (cf. [12, p. 34]):

$$w(\vec{x}\,, y) \;:=\; \psi^{pre}(\vec{x}\,) \wedge \psi^{post}(\vec{x}\,, y),$$

where $\psi^{pre}(\vec{x}\,)$ and $\psi^{post}(\vec{x}, y)$ are arbitrary first-order formulas describing pre- and post-conditions. Now it is easy to observe that preconditions $\psi^{pre}(\vec{x}\,)$ correspond to what we called types of inputs $\vec{x} : \vec{X}$, with the only difference that in our framework, these types are DL concepts, not arbitrary FO formulas. Furthemore, $\psi^{post}(\vec{x}\,, y)$ corresponds to our conjunction $y{:}\,Y \wedge \Phi(\vec{x}\,, y)$, again restricted syntactically to a conjunctive query over a DL (see also the discussion after Definition 3.3).

In contrast to this rich description of services, requests are still described by a set of relevant objects, and this set is defined by an arbitrary first-order formula $g(y)$. From this one can conclude, in particular, that here and in the previous approach with simple descriptions of services, the set of *relevant objects* is nothing more than the set of *outputs* of a service, or more exactly, the union of its outputs over all possible inputs. Now it remains to observe that, for a service of the form $S = \langle \vec{x} : \vec{X}; y{:}\,Y; \Phi(\vec{x}, y)\rangle$, this union is equal to the set defined by the formula $g(y) := \exists\vec{x}\;(\vec{x} : \vec{X} \wedge \Phi(\vec{x}\,, y) \wedge y{:}\,Y)$, which has the form of a conjunctive query with one distinguished variable y and describes a service with no inputs and a single output. Therefore, it becomes evident that the "simple semantic description" approach is embedded (except for the restriction of the underlying language) into our framework as a special case for services with no inputs and a single output (indeed, the matching condition in this approach coincides with that in our framework for this special kind of services).

The notion of service matching within this approach is twofold: given a service described by a formula $w(\vec{x}\,, y)$ and a request $g(y)$, one is interested in whether the service w can produce the requested information in a *single execution* or in *multiple executions*. We will discuss here only the notion of exact match, since it extends to other ones (plugin match, etc.) in an obvious way. We also leave the intention component of descriptions apart, as it is always clear how to handle it.

The matching condition for the case of a *single execution* is: there exists an input tuple $\vec{x}$ such that the set of outputs returned by the service on the input $\vec{x}$ coincides with the requested set. This can be expressed by the formula:

$$\mathcal{T} \models\; \exists\vec{x}\;\forall y\,(\,g(y) \;\leftrightarrow\; w(\vec{x}\,, y)\,).$$

In fact, this "pure" existential quantifier does not reflect the intuition behind the matching condition we define. Even if we check this entailment, the user is interested in executing the service with the suitable input. Hence he wants not only to know that such an input exists, but to have it explicitly, as a tuple of individual names. Therefore, it is more appropriate to reformulate the condition as follows: there exists a tuple of individuals $\vec{a}$ such that the set of outputs returned by the service on the input $\vec{a}$ coincides with the requested set: $\mathcal{T} \models g(y) \doteq w(\vec{a}\,, y)$. Clearly, this is the notion of service instance (as introduced in our Definition 5.2) matching another service.

For the case of *multiple executions*, the matching condition is: an element is in the requested set iff it is returned by the service on at least one input. This is expressed by the formula:[5]

$$\mathcal{T} \models\; \forall y\,(\,g(y) \;\leftrightarrow\; \exists\vec{x}\;w(\vec{x}\,, y)\,).$$

As follows from this definition, in order to obtain the whole requested information, one should need a potentially infinite number of invocations of the service with different inputs (since the set defined by the formula $g(y)$ has no restrictions on cardinality). On practice, however, the requested information is always finite, since it is returned as the set of *individuals*, and there is a finite number thereof in a knowledge base. Therefore, a more suitable would be the following formulation of this matching condition: an individual is in the requested set iff it is returned by the service on some tuple of individuals as an input; formally, $\mathcal{T} \models g(y) \doteq \hat{\exists}\vec{x}\,w(\vec{x}\,, y)$ (we use the notation from [8], where it is also shown that this kind of problem is reducible to standard reasoning problems).

[5]In the cited paper, this matching condition is written as $\forall y \exists\vec{x}\,(g(y) \leftrightarrow w(\vec{x}\,, y))$, cf. formula (13) in [12], which is an obvious mistake: this formula can be equivalently rewritten as

$$\forall y\,\big[\,(g(y) \rightarrow \exists\vec{x}\,w(\vec{x}\,, y)) \;\wedge\; (\neg g(y) \rightarrow \exists\vec{x}\,\neg w(\vec{x}\,, y))\,\big].$$

Its first conjunct is correct, but the second one says: for any element y beyond the requested set, it is not returned by the service on *at least one* input (whereas it should not be returned at all). A similar mistake occurs in formula (15) in [12], which is actually identical to the second conjunct of the above conjunction.

On the expressivity of the underlying language

In [12], it is also pointed out that one of the advantages of their approach is that the language for describing services and requests is not fixed: the set of relevant objects may be defined in a way not expressible in DLs. Although this yields to higher generality and increase in expressivity, the drawbacks are the possible loss of decidability and thus the limited possibility (or even impossibility) to build an *automated* service discovery engine.

Generally speaking, the logical part of a framework and a choice of the underlying object language are quite independent. In particular, in our framework, we could also allow the types of inputs $\vec{X}$ and outputs $\vec{Y}$ to be not DL concepts, but arbitrary formulas with free variables $\vec{x}$ and $\vec{y}$, resp., and $\Phi(\vec{x}, \vec{y})$ to be a formula. Following this way, we do not need even to modify any other definitions (except for considering arbitrary theories instead of knowledge bases). Also note that, strictly speaking, service descriptions in our approach are not pure DL (i.e., tree-like) expressions: in order to describe relationships between inputs and outputs, we had to invoke conjunctive queries, which may not be tree-like only, but are cyclic in general.

Our choice of the object language is supported by two reasons. First, it should lead to a *decidable* notion of service matching. At the same time, no decidability or complexity results are obtained in [12] or, to the best of our knowledge, in any later papers on this approach. However, this is one of the main objectives in building a service discovery formalism.

Secondly, the generality of the approach should be as high as is necessary for its applications, so that one should be able to use the standard ontologies that are developed (or under development) for many application domains that are formulated using expressive DLs.

Acknowledgements

The research is supported by an EPSRC grant GR/S63182/01, GR/S63168/01 as part of the DynamO project (`http://dynamo.man.ac.uk/`). The authors would like to thank Ian Horrocks, Ulrike Sattler, and Frank Wolter for their help during the research.

References

[1] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[2] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Data Complexity of Query Answering in Description Logics. In *Proc. of the 2005 Description Logic Workshop (DL 2005)*. Edinburgh, Scotland, UK, July 26–28, 2005. CEUR Workshop Proceedings, ISSN 1613-0073. `CEUR-WS.org/Vol-147/`

[3] D. Calvanese, G. De Giacomo, and M. Lenzerini. On the decidability of query containment under constraints. In *Proc. of the 17th ACM SIGACT SIGMOD SIGART Sym. on Principles of Database Systems (PODS'98)*, pages 149–158, 1998.
`http://www.inf.unibz.it/~calvanese/publications-year.shtml`

[4] Horrocks, I., Sattler, U., Tessaris, S., Tobies, S. How to decide query containment under constraints using a Description Logic. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning (LPAR'2000)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2000.

[5] Horrocks, I., and S. Tessaris. A conjunctive query language for description logic ABoxes. In *Proc. of the 17th Nat. Conf. on Artificial Intelligence (AAAI'2000)*, pp. 399–404, 2000.

[6] Lutz, C. *The Complexity of Description Logic with Concrete Domains*. PhD Thesis, LuFG Theoretical Computer Science, RWTH Aachen, Germany, 2002

[7] Tessaris, S. *Questions and answers: reasoning and querying in Description Logic*. PhD thesis, University of Manchester, 2001.

[8] U. Sattler, E. Zolin. Looking for Individuals: Reasoning with Must-Bind Quantifiers. Manuscript. Available at `http://www.cs.man.ac.uk/~ezolin/logic/publications.html`

[9] The DAML Services Coalition. Bringing Semantics to Web Services: The OWL-S Approach. *Proc. of the 1st Int. Workshop on Semantic Web Services and Web Process Composition (SWSWPC'2004)*, July 6-9, 2004, San Diego, California, USA.

[10] Terry R. Payne, Massimo Paolucci, and Katia Sycara. Advertising and Matching DAML-S Service Descriptions. *Semantic Web Working Symposium (SWWS)*, 2001.

[11] M. M. Ortiz de la Fuente, D. Calvanese, T. Eiter, and E. Franconi. Data complexity of answering conjunctive queries over $\mathcal{SHIQ}$ knowledge bases. Technical report, Faculty of Computer Science, Free University of Bozen-Bolzano, 2005. Also available as CORR technical report at `http://arxiv.org/abs/cs.LO/0507059/`

[12] Uwe Keller, Rubén Lara, Axel Polleres, Ioan Toma, Michel Kifer, Dieter Fensel. WSMO Web Service Discover, D5.1 ver.0.1. Technical Report, DERI, University of Innsbruck, 2004.
`http://www.wsmo.org/2004/d5/d5.1/v0.1/`

[13] Uwe Keller, Rubén Lara, Holger Lausen, Axel Polleres, Livia Predoiu, Ioan Toma. Semantic Web Service Discovery, ver.0.2. WSMX Working Draft 03 October 2005.
`http://www.wsmo.org/TR/d10/`

[14] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In Proceedings of the 12th International Conference on the World Wide Web (WWW 2003), pp. 331–339. Budapest, Hungary, May 2003.