Geoinformatica
DOI 10.1007/s10707-009-0093-4

Functional description of geoprocessing services as conjunctive datalog queries

Daniel Fitzner · Jörg Hoffmann · Eva Klien

Received: 10 January 2008 / Revised: 18 February 2009
Accepted: 14 September 2009

Abstract Discovery of suitable web services is a crucial task in Spatial Data Infrastructures (SDI). In this work, we develop a novel approach to the discovery of geoprocessing services (WPS). Discovery requests and Web Processing Services are annotated as conjunctive queries in a logic programming (LP) language and the discovery process is based on Logic Programming query containment checking between these descriptions. Besides the types of input and output, we explicitly formalise the relation between them and hence are able to capture the functionality of a WPS more precisely. The use of Logic Programming query containment allows for effective reasoning during discovery. Furthermore, the relative simplicity of the semantic descriptions is advantageous for their creation by non-logics experts. The developed approach is applicable in the Web Service Modeling Framework (WSMF), a state-of-the-art semantic web service framework.

Keywords Geoprocessing services · Service discovery · Semantic anotation · Logic programming · Conjunctive queries

1 Introduction

In recent years, the paradigm shift from the use of monolithic software systems and databases to Service Oriented Architectures (SOA) had a great impact on geospatial information products. Both, spatially referenced data as well as geospatial functionality

D. Fitzner (✉) · E. Klien
Fraunhofer Institut für Graphische Datenverarbeitung, Abteilung Graphische Informationssysteme, Fraunhoferstraße 5, 64283 Darmstadt, Germany
e-mail: daniel.fitzner@igd.fraunhofer.de

E. Klien
e-mail: eva.klien@igd.fraunhofer.de

J. Hoffmann
SAP Research, Vincenz-Prießnitz-Straße 1, 76131 Karlsruhe, Germany
e-mail: joe.hoffmann@sap.com

usually provided by Geographic Information Systems (GIS), are increasingly spread over the internet and made accessible to users all over the world via web services.

This paradigm change can be illustrated with the help of a simple—but practically relevant—example. A regional planner has the task to find potential building sites. He has several constraints on the areas. For example, potential areas that are lowlands should be at least 500 meter away from rivers to minimize the risk of being flooded. In order to process this information, he has data sets containing river objects (represented as *lines* or *polygons*) as well as data sets with potential developable areas (represented as *polygons*). To get the information product he wants, further processing on this data is needed.

Usually, this processing is performed on a standalone PC: First, the regional planner needs to gather the regional data on *rivers* and *potential building sites*. After receiving the data, e.g. on a CD-ROM or via download from the provider's web site, the regional planner imports it into his standalone GIS and employs the system's processing functionality in order to achieve a suitable output, i.e. all potential developing areas more than 500 meter away from rivers. In this example, this processing consists of a 500 meter buffering around the river objects and the subsequent application of the *difference* operation. The application of the *difference* operation returns a polygon-layer containing all potential developable areas (or parts of them) that do not intersect with the buffered polygon layer that represents a 500 meter area around rivers.

As pointed out in several publications such as [1] or [2], this common approach to geospatial tasks has several drawbacks. For example, each time the regional planer wants to repeat his task of finding potential building sites, he has to check, if the data he received is still valid/up do date, since especially the data on potential developable areas is subject to rapid changes. Hence, the regional planner probably needs to reorder the data each time he performs the task, which can be time consuming and involves additional work. Moreover, the regional planner needs a GIS to perform the necessary processing steps. If he has no GIS available, he will not be able to achieve his task.

Spatial Data Infrastructures (SDI) have been designed to overcome these problems. SDI provide multidisciplinary access to distributed and heterogeneous geodata and geoservices with the help of web service technology. This has the advantage that the regional planner does not need to have specialised software or specific data sets on his computer in order to achieve his task. In this case, both, the data (*rivers, potential developable areas*) as well as the processing functionality *(buffer, difference*) are offered via geospatial web services. These services can be accessed and combined from users all over the world via the World Wide Web.

To employ and compose the web services that (altogether) support the regional planner in generating the information product he wants, he first needs to discover these services. Discovery is one of the crucial tasks in such open and distributed environments and can be characterized as the process of locating web services that provide the requested data or functionality.

This paper is concerned with the discovery of geoprocessing functionality, such as *buffer* or *overlay* operations. We assume that the input data in our example (i.e. data representing rivers and potential developable areas), has already been located, e.g. via the discovery of suitable Web Feature Services (WFS) [3]. The regional planner's task is now to discover suitable processing functionality that can help him in achieving his task, i.e. finding all potential developable areas that are not too close to rivers. Hence, the regional planer needs to discover Web Processing Services (WPS [10]) that implement the *buffer* and *difference* operations and that are able to execute on the input data provided by the (already discovered) WFS.

Discovery of web service is usually realised with so called registries or catalogue services. Web service providers publish descriptions of the web services' capabilities in some registry or catalogue. Requesters can then query the catalogue in order to locate web services that deliver the requested service. These web service advertisements used for discovery are often called *functional descriptions* ([4–6]). Disregarding on how to invoke a web service and how to process the output (and how the web service achieves this output), functional descriptions are declarative description of what is provided (output) and what is required (input) by the web service they describe. Hence, functional descriptions "shall provide a black box description for determining the usability of a web service for some request or usage scenario with respect to the provided functionality" [5]. For example, in the case of a *buffer* operation, a functional description contains the information that the buffer inputs some geometrical objects (the objects to be buffered) and some number (the buffer distance).

Multiple possibilities for web service discovery exist. The most obvious one is to base the discovery process on the comparison of natural language descriptions of requests and web services. Another possibility is to use some sort of shared vocabulary inside the functional descriptions. This shared vocabulary can either have the form of keywords (e.g. "*polygon"* or *"buffer"*) or more complex expressions in some formal language. In the latter case, discovery requests and web service advertisements are described using the shared and formalised background knowledge, which is usually provided in the form of domain ontologies. Ontologies can be characterised as explicit specifications of conceptualisations [7]. The term conceptualisation refers to an "abstract simplified view of the world" [7]. In case of domain ontologies, the conceptualisation covers a whole knowledge domain such as the domain of Regional Planning. In computer science and information systems, ontologies are formalised. This formalisation (usually in a declarative, logical language) enables automatic reasoning/inferencing tasks within the domain that can usually only be performed by humans.

Within ontology-based approaches to web service discovery such as [8] or [9], the discovery process relies on automatic inference mechanisms based on the ontology language. These approaches are usually subsumed under the term *semantic web service discovery*. This is the approach we follow in this work. The advantage of using ontologies for search and discovery of geospatial information is clearly pointed out in several publications such as [18, 19] or [20]. Ontologies deliver unambiguous semantics of terms as well as the possibility for automated reasoning. Both are of huge importance for geospatial information discovery and retrieval. Herein, we develop a new approach for exploiting this for improved precision and recall in the discovery of WPS services.

In our approach, two domain ontologies provide the formalised background knowledge to be used inside the functional descriptions of requests and geoprocessing functionality. The functional descriptions have the form of *conjunctive datalog queries*. The discovery process then establishes a query containment relation between these descriptions. Figure 1 gives an overview. The requester, who wants to discover a WPS, formalises his request as a conjunctive datalog query. The terms within the query are taken from the background ontologies. The web service provider does the same for the advertised WPS. The discovery itself then relies on query containment, a well known reasoning technique in the area of databases and, especially, query optimisation.

In a little more detail, our approach requires two background ontologies, one formalizing the possible datatypes of the web service's arguments (e.g. polygon, point), and one formalizing the possible operations performed by web services (e.g. intersect). To describe a Web service, the provider uses terms from these ontologies, conjoining them into a

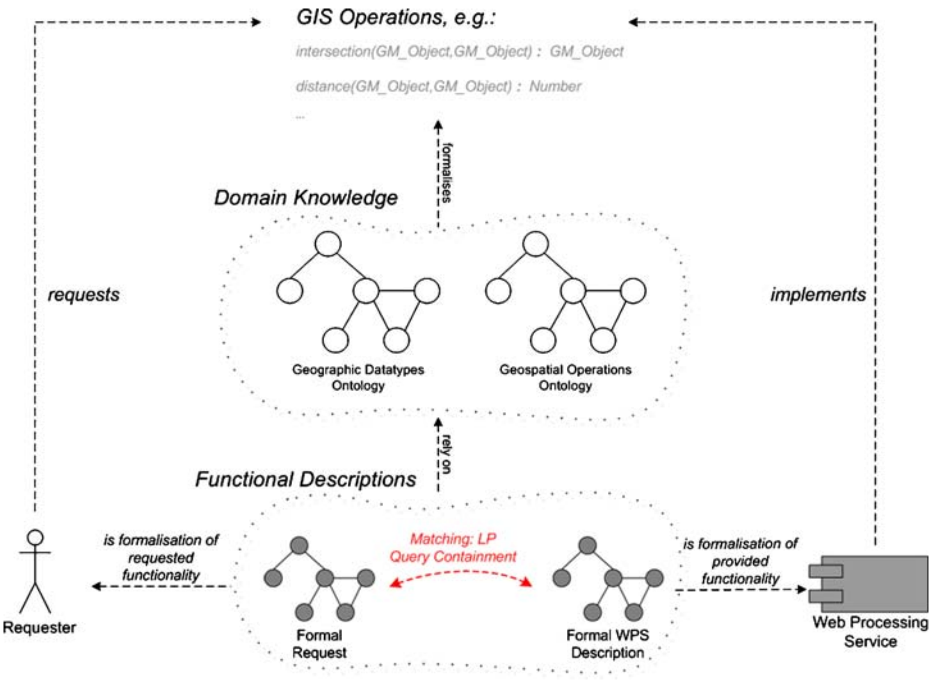

GIS Operations, e.g.:
intersection(GM_Object,GM_Object) : GM_Object
distance(GM_Object,GM_Object) : Number
...
formalises
Domain Knowledge
requests
implements
Geographic Datatypes Ontology
Geospatial Operations Ontology
rely on
Functional Descriptions
is formalisation of requested functionality
Matching: LP Query Containment
is formalisation of provided functionality
Requester
Formal Request
Formal WPS Description
Web Processing Service

Fig. 1 Overview on the presented approach to web service discovery

conjunctive query which specifies what types the arguments have, what operation is performed on them, and how the output parameters relate to the input parameters.[1] The same is done by the web service requester to construct a discovery query. The discovery is then performed through matching each web service description—the respective conjunctive query—against the discovery query. A match occurs iff there exists a containment relation between the queries. The discovery result consists of those Web services whose descriptions match the query.

The core question in an approach such as the above is: *How are the queries formed and how are they compared to each other?* That is, how do we describe Web services/discovery requests, and how do we match them? Herein, we develop a specific approach that is particularly suitable for handling Web Processing Services. We target an aspect that is not dealt with adequately by any of the existing approaches, namely how the output parameters of the Web service result from its inputs. Note that this is crucial for WPS services, which after all compute a function of their inputs. Our matching operation is derived from a more basic well-known operation known as the "Plug-in match". According to what we have just discussed, we term our approach *In-Out-Aware Plug-in match.* We prove that this match adheres with a number of desirable properties regarding inputs and outputs; we provide encouraging initial empirical results.

The remainder paper is organized as follows: In Section 2, the necessary background for our approach is given. This contains a discussion of foundations of web service discovery, a

[1] Another property of some WPS services is how they deal with the non-spatial attributes of their inputs; herein, we ignore this issue and focus only on the spatial aspects. Note that the two issues are entirely orthogonal.

list of requirements on the WPS discovery process and an introduction to the technologies used. Section 3 introduces the structure of the functional descriptions we use as well as examples of simple domain ontologies and standard matching techniques. Section 4 contains the heart of our approach, for the In-Out-Aware Plug-in match. Preliminary empirical results are described in Section 5. Section 6 discusses related work, and Section 7 concludes.

2 Background

In the following, we provide the reader with the necessary background knowledge. Section 2.1 discusses foundations of web service (especially WPS-) discovery. Section 2.2 introduces the notion of subtyping of software operations (functional descriptions of WPS and requests, in our case). In Section 2.3, the key notions from Logic Programming, especially Query Containment and Containment Mappings are given.

2.1 Foundations of WPS discovery

In this section, we introduce Web Processing services and discuss some general foundations of WPS discovery. Afterwards, we give a number of requirements we consider being crucial for the *functional descriptions* and the discovery of WPS.

The process of web service discovery strongly depends on the type of web service to be discovered. Discovery of data-providing web services such as the OGC´s Web Feature Service [3] mainly deals with metadata descriptions of the provided data. Requesters can query them in order to locate those web services that provide suitable data. Discovery of functionality-providing web services such as geoprocessing services is different. In contrast to data-providing web services, they offer *operations* on spatially referenced data instead of the data itself. WPS are intended to offer multiple types of operations on spatially referenced (vector or raster) data, which include well-known GIS operations such as *overlay*, but also arbitrarily complex calculations such as a global climate change model [10]. Despite the wide variety of possible operations, WPS have a few things in common, as pointed out in [11]:

- *WPS* are *information providing* web services. This means, there is no "real world" counterpart to the computation performed. The execution of a WPS does not affect the real world but only the information space in which the WPS is executed.
- After WPS execution, either a *new* fact becomes known or a *known* fact has been updated. For example a web service offering the geospatial *intersection* operation creates a new spatial feature based on the input features, whereas a web service offering coordinate transformation updates the spatial attribute of the input.

One category of GIS operations that are potentially offered by WPS is overlay. "The reason overlay plays a key role is that most applications of geographic information must integrate information from different sources" [12]. For this reason, we use overlay operations as our running example. Since the main driving force behind SOA is integration of information from different sources, it is likely that overlay operations offered via WPS will play a crucial role in future SDI.

Overlay operations receive multiple (at least two) layers of spatially referenced data as input and yield one layer as output that is the result of spatially overlaying the input. As pointed out in [12], overlay operations have a certain similarity to *joins* between tables in a relational data model in the sense that they combine different datasets based on a common key. In the case of overlay operations, this key is the geometry instead of some non-spatial attribute inside the attribute tables. For this reason, a useful output layer can only be computed, if all of the input layers adhere to a common spatial reference system (unless the WPS also performs coordinate transformation). Different ways for computing the output geometries from the input exist, namely *Intersection*, *Union*, *Difference* and *Symmetric-Difference*. For example, *intersecting* two polygons A and B delivers a polygon that covers the geometric area where *A* and *B* overlap. Figure 2 shows the different overlay operations on polygons.

In general, the possibility to describe the functionality of a WPS for the purpose of discovery ranges from its classification by keywords to a detailed description of the implemented mathematical equations or algorithms. Both options have several drawbacks. Automated discovery based on keywords or natural language descriptions suffers from low precision and recall due to synonyms and homonyms in natural language and moreover due to the inability of computers to perform reasoning/inferencing on this kind of information. On the other hand, the automated discovery based on algorithm descriptions is also not a good idea. It requires the requester to provide a detailed description of the requested algorithm, which is not a realistic assumption since geoprocessing algorithms can be arbitrarily complex. Moreover, if a requester is able to provide such a detailed description, then she can simply use an executable programming language for encoding her request, and no longer needs a WPS.

Therefore, a suitable level of detail for functional descriptions is needed. On the one hand, these descriptions should be expressive and detailed enough to enable web service discovery with high precision and recall. On the other hand, the task of annotating a WPS or formulating a request should still be feasible. In order to achieve such level of detail, we formulate the following requirements for functional descriptions to be used for WPS discovery. In the following, the term functional description refers to a WPS advertisement as well as a request. Hence, we adopt the usual way of treating requests as "desired" web

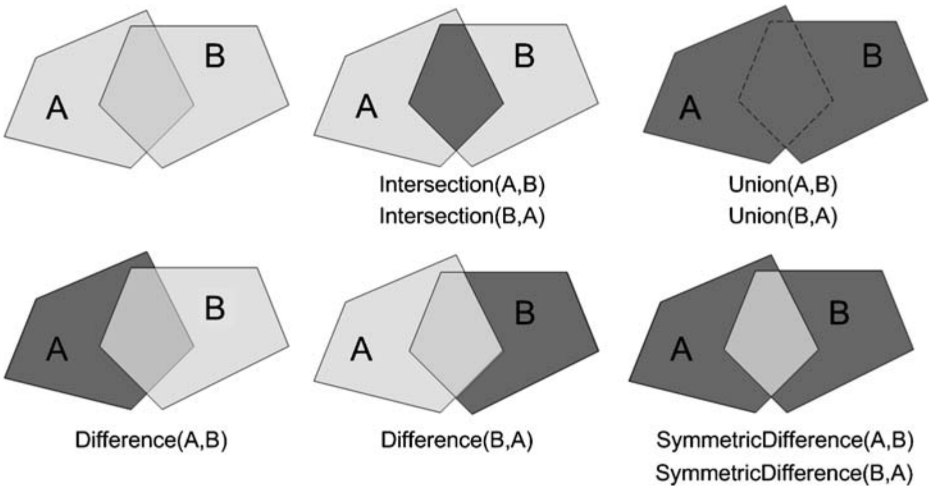


Fig. 2 Different overlay operations on polygons

services. In our opinion, functional descriptions used for WPS discovery should at least contain some description of...:

- ...type signatures (the input/output types)
- ...constraints on in- and output
- ...the operation that is performed/requested
- ...the dependencies between input and output

The description of the signature is a crucial part of functional descriptions. It ensures syntactic interoperability between requester and web service. Corresponding type signatures guarantee that the web service can execute on the provided input and that the requester can accept the delivered output with respect to their syntactic types. In Section 2.2, we introduce the common notion of *function subtypes* for comparing type signatures. To additionally ensure that the web service really executes on the provided input in the way expected, constraints need to be formulated that further narrow the possible input and output values. As described above, a WPS offering overlay calculations on polygons requires all input values (besides being of type polygon) to adhere to a common coordinate reference system in order to be able to calculate a useful output. Furthermore, since web services with equal type signatures and equal constraints on the input and output variables can compute very different operations, some description of the implemented algorithm or operation is needed.

However, this will again not be sufficient in many cases. For example, consider a WPS offering the difference operation on polygons. When discovery is only performed based on some description of type signatures, constraints and the operation, it is e.g. not possible to differ between the calculation of Difference(A,B) and Difference(B,A), although their results are quite different. Referring to the real world example from the introduction, the execution of the difference operation with a wrong permutation of the two input layers would result in a layer of polygons that consists of all areas within a 500 meter radius around rivers, which do not intersect with developable areas. Regarding the initial regional planer’s task, this is obviously a wrong result.

Hence, in order to ensure that the WPS′ input variables are instantiated with a valid permutation of the requester input, some description of the relation or dependency between input and output is needed. This description should guarantee that both, requester and web service, agree on the way the output is computed from the input. We consider these dependencies as crucial for the discovery of WPS, since they lie in the very nature of processing services, providing an output computed as some function of a set of given input parameters. In particular, these dependencies constitute a characteristic difference between data *processing* and data *providing* services.

2.2 Function subtypes

This section introduces the common notion of Function Subtypes and a characterisation of Function Subtypes for multi-argument functions/operations is given.

In the literature, multiple efforts are described that try to clarify whether software operations (or web services) *match*, that is: whether one of them can be replaced "with another without affecting the observable behaviour of the entire system" [13]. Usually, two software components are considered as a match, if both, their *signatures* (i.e. the input/output datatypes) and their *behaviour* (the operation or algorithm) correspond. For determining this correspondence, several notions of *signature* and *behavioural subtypes* for operations have been identified. Transferring this to web service discovery, a web service should be discovered if it is a (signature and behavioural) subtype of the "desired" web service described in the request. We characterise function subtypes for multi-argument operations or web services as follows:

Let $X_1,...,X_n$ ($A_1,...A_m$) be a list of input types.
Let Y and B be output types.

The function type $X_1,...,X_n \rightarrow Y$ is a subtype of the function type $A_1,...,A_m \rightarrow B$ if

- *for each input type $X_i \in X_1, ..., X_n, 1 \leq i \leq n$ there exists a distinct input type $A_j \in A_1, ..., A_m, 1 \leq j \leq m$ with A_j being a subtype of X_i*
- *Y is a subtype of B*

Intuitively, this means that each input variable of the subtype operation (i.e. the web service) needs to have a corresponding input variable of the supertype operation (i.e. the request), whose type is a subtype. The output type of the web service is required to be a subtype of the desired output type of the request.[2] Note that, with the above characterisation, the request is allowed to provide more input than required by the web service, as long as some of this input is sufficient for executing the web service. We consider this as an intuitive condition for *matchmaking*.

Example 1: (Function Subtypes) Consider the signatures of two software operations or web services A and B:

Operation A: *(Polygon, Polygon) → Polygon*
Operation B: **(***GM_Object, GM_Object) → Polygon*

The type *GM_Object* represents a generic geometric object and we assume that a subtype relationship holds between *Polygon* and *GM_Object*. In the following, we show that **Operation A** can not be a function subtype of **Operation B**, that is: **Operation A** can not replace **Operation B**. The reason is, since *Polygon* is a subtype of *GM_Object*, there are values of *GM_Object* that are no values of *Polygon*. Hence, some of the input values of **Operation B** are not acceptable by **Operation A**, namely those values that are *GM_Objects* but no *Polygons*. For this reason, **Operation A** can not be executed in place of **Operation B** without risking a runtime exception. Hence, **Operation A** can not be a subtype of **Operation B**.

Although software operations or web services can be identified by their input and output types, this is by far not sufficient as described in Section 1. Considering type signatures delivers not much information about the actual "behaviour" of the operations since software operations or web services that have equal type signatures can compute very different functions (e.g "Intersection(A,B) and "Difference(A,B)" have the same type signature). *Behavioural subtyping* tries to overcome this lack by further considering the operation's behaviour. Usually, this behaviour is described in terms of *pre-* and *postconditions* [14] that hold before or after the operation's execution. The most common notion of behavioural subtyping is that of a *Plug-in match* that directly transfers the function subtype characterisation to the operation's behaviour. We will further explain a variant of the *Plug-in match* in Section 3.3.2.

2.3 Query containment in LP

This Section introduces the technology we use for WPS discovery, namely Logic Programming, Query Containment in LP and Containment Mappings.

[2] We assume that a WPS has a single *output*, e.g. a single polygon or polygon-layer

Logic programming (LP) transfers the declarative style of 1st order logics to the realm of computer programming. In particular, LP focuses on logical reasoning problems that are decidable and that are hence suitable as the basis of practical applications. LP languages provide a formal syntax that includes: constant symbols for the representation of individuals; predicate symbols for specifying relations between individuals; logical connectives such as "or", "and"; variables for making general statements about unknown individuals. In particular, herein we focus on datalog [15], which features LP rules of the form

$$P : -Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n$$

The meaning of such a rule is that, whenever all the Q_i are true, then P is also true. The left hand side of the rule is its head, the right hand side is its body. Here, P and each Q_i has the form $p(t_1,...,t_m)$ where p is a predicate of arity m, and each t_i is either a variable or a constant. One special form of rules are those with an empty body and a head which is ground, i.e. whose arguments contain no variables. Such rules are called facts. Facts can be used to specify the instances (represented as constant symbols) on which a certain predicate holds.

The most prominent reasoning task in LP is query answering, i.e. the process of deriving new facts from a database of already established facts. Following the characterization of conjunctive datalog queries in [16], we define:

> A conjunctive query is a rule in which a predicate is defined in terms of one or more predicates other than itself.

In other words, every non-recursive rule is a conjunctive query. The predicates appearing in the body of a conjunctive query are referred to as the query's subgoals. The variables appearing in the head of a conjunctive query are called *exported* variables; this may be an arbitrary subset of the variables appearing in the rule body. Within this paper, we will ignore this issue, since it is not of essential importance for the approach we are proposing, where queries correspond to functional preconditions/postconditions which do not have a natural notion of "exported" variables.[3] We will simply assume that all variables are exported, and we will often ignore the rule head; by convention, this should be taken to mean that the head consists of some new predicate containing all of the body's variables.

One question that is of particular importance in our work is *query containment*, which is defined as follows:

> A conjunctive query Q_1 is contained in a conjunctive query Q_2, written $Q_1 \subseteq Q_2$, if, whatever the established database of facts is, the set of additional facts provable from Q_1 is a subset of those provable from Q_2 [16] .

In other words, $Q_1 \subseteq Q_2$, if and only if the set of facts on which the head of Q_1 holds is a subset of the set of facts on which the head of Q_2 holds, independently from the actual database that is evaluated by the two queries. Due to this independence from a particular database, query containment can be tested syntactically based on the structure of the two queries and on the set of rules given in the datalog ontology.

[3] This notwithstanding, marking precondition/postcondition variables as "exported" may be useful to explicitly distinguish input/output variables; such special markers are not supported by current Semantic Web Services approaches, and exploring this option is a topic for future work.

2.3.1 Containment mappings

An interesting characterization of when a containment holds is in the form of so-called *containment mappings* [17]. Containment mappings turn the containing query Q_2 into the contained query Q_1 by mapping each subgoal from Q_2 to a corresponding subgoal that can be derived—by applying rules—from the body of Q_1. Namely, a function h from the set of symbols (predicates, constants, variables) used in Q_2 into the set of symbols used in Q_1 is said to be *a containment mapping from Q_2 to Q_1 if:*

- *h is the identity function on constants;*
- *and, for each subgoal $G_i(y_1, ..., y_n)$ in the body of Q_2 ,$G_i(h(y_1), ..., h(y_n))$ is a fact in $\overline{Q_1}$, where $\overline{Q_1}$ is the deductive closure of the body of Q_1.*

In other words, as mentioned in [15], a containment mapping from a query Q_2 to a query Q_1 exists, if and only if the body of Q_1 logically implies the body of Q_2: the deductive closure of the body of Q_1 is the set of all facts that can be derived from the body of Q_1 by iteratively applying datalog rules until a fixpoint is reached. It is then easy to see that a query containment relation $Q_1 \subseteq Q_2$ holds if and only if there exists a containment mapping from Q_2 to Q_1. To illustrate this by an extreme case: if the database of rules is empty, i.e., if the ontology specifies no rules on the behaviour of the predicates, then the deductive closure of the body of Q_1 is the same as that body itself and the containment mapping must map every subgoal $G_i(y_1,...y_n)$ in the body of Q_2 to a subgoal in the body of Q_1. A less extreme version of this special case is when the particular predicate G_i to be mapped does not appear in the head of any of the datalog rules: then, G_i cannot be deduced and hence, as in the extreme case, it must be mapped directly to a subgoal in the body of Q_1. This latter special case will be relevant in our investigation, where in the In-Out-Aware Plug-in match we use certain special new predicates—that are not mentioned in the underlying datalog ontologies—in order to constrain the queries in a way enforcing the desired behaviour of the matching.

Example 2: (Query Containment and Containment Mappings) Consider the following datalog ontology O:

$$gm\_object(X) : -point(X).$$

Consider two conjunctive queries Q_1 and Q_2:

$$Q_1 : q(X) : -point(X) \wedge hasSRS(X, Y) \wedge geogSRS(Y)$$

$$Q_2 : q(A) : -gm\_object(A) \wedge hasSRS(A, B)$$

Applying the rule of the datalog ontology O on Q_1 delivers the deductive closure $\overline{Q_1}$:

$$\overline{Q_1} : \; q(X) : -point(x) \wedge hasSRS(x, y) \wedge geogSRS(y) \wedge gm\_object(x)$$

As can easily be seen, each subgoal in the body of Q_2 has a corresponding fact in the deductive closure $\overline{Q_1}$. Therefore, the containment relation $Q_1 \subseteq Q_2$ holds with the containment mapping that maps A to X and B to Y.

3 Functional descriptions for WPS, and standard matching techniques

In this section, we introduce our ontology based approach to semantic WPS discovery. In Section 3.1 we explain what kinds of background ontologies our approach relies on. Section 3.2 introduces the structure of functional descriptions used for discovery. In Section 3.3, standard matching techniques between functional descriptions are described and their drawbacks are identified.

3.1 Ontologies for functional descriptions of WPS

Our discovery approach relies on two ontologies in the background. (1) a *geographic datatypes ontology* is used to formalize geographic datatypes derived from already well-agreed standards such as the ISO Spatial Schema [21]. That is, the scope of the geographic datatype ontology is the terminology used to describe the typing of the parameters of the relevant WPS services, such as "Polygon" (2) a *geospatial operation ontology* is used to formalize GIS operations. The scope of this ontology is a categorization of different operations; at an informal level, one may view it as an organized collection of keywords, where each keyword names one particular operation such as "SymmetricDifference".

Herein, we do not provide sophisticated instances of these two ontologies. Our focus is on discovery, and within this section we only present simple ontologies for illustration purposes. From a theoretical point of view (disregarding runtime and performance issues), our approach works with any background ontology, as long as it is formalised in a datalog language and consistent.

Within our example ontologies, geographic datatypes are represented as *unary predicates* and sub type relationships are formalised as LP implication between them. For example, a *polygon* is defined as a subtype of a generic *gm_object*:

$$gm\_object(A) : -polygon(A). \tag{3.1}$$

Additionally, the geographic datatypes ontology contains instance specifications such as the projected spatial reference system *Gauß-Krüger*:

$$projSRS(gk). \tag{3.2}$$

The adopted standards for geographic datatypes such as ISO Spatial Schema can serve as a basis for the development of a *geographic datatypes ontology* and therefore for the annotation of type signatures. But (as mentioned in [11]) they provide no information about the *functionality* or *behaviour* of geospatial operations. For this reason and since, as described in Section 2, the functional descriptions should not in detail refer to the mathematical equations of the underlying algorithms, an abstract conceptualisation of operations is needed. The geospatial operations ontology contains a lightweight characterisation of operations as *ternary predicates*. The first two variables refer to the operations *input*. The third argument represents the operations *output*. For example the statement *difference(A,B,C)* denotes that C is the output of the difference operation on A and B. In order to allow requesters to search for all of the different overlay operations using a single request, we formalise the following dependencies:

$$\begin{aligned} &overlay(A,B,C) : -union(A,B,C) \\ &overlay(A,B,C) : -intersection(A,B,C) \\ &overlay(A,B,C) : -difference(A,B,C) \\ &overlay(A,B,C) : -symmetricDifference(A,B,C) \end{aligned} \tag{3.3}$$

Additionally, some overlay operations compute the same output when executed with different permutations of the input. As can be seen in Fig. 2, the geometric output of *symmetric difference(A,B)* is equivalent to the geometric output of *symmetric difference(B,A)*. We formalise this as LP implication between the predicates that represent the operations:

$$symmetricDifference(A,B,C) : -symmetricDifference(B,A,C) \tag{3.4}$$

Furthermore, since the different overlay operations are quite similar to set-theoretic relationships, it is possible to naturally formulate dependencies between them. Some overlay operations can be computed via other overlay operations. For example, as can be seen in Fig. 2, a *symmetric difference* operation can be computed via *difference* and *union* operations. These dependencies between operations are formalised as LP rules:

$$\begin{aligned} symmetricDifference(A,B,C) : -&difference(A,B,X)\wedge \\ &difference(B,A,Y) \wedge union(X,Y,C) \end{aligned} \tag{3.5}$$

Formalising dependencies between the operations theoretically allows to automatically generate a service chain out of several WPS for a given request. For example, Eq. 3.5 allows to chain a *difference* and a *union* WPS for calculating a *symmetric difference* (assuming that the type signatures correspond). However, how this is exactly done is outside the scope of this paper and subject to future research.

3.2 Functional descriptions

The functional descriptions of requests and WPS are formalised using the formal background knowledge from the domain ontologies. Following the framework for semantic web services provided by WSMO [22], they consist of *preconditions*, *postconditions* and a set of *shared variables* appearing in both. Note: WSMO additionally differs between the "real world" (*assumptions/effects*) and the information space (*pre-/postconditions*). Since, as described in Section 2, WPS are information providing web services, only pre- and postconditions are required.

Each of the formulas—precondition and postcondition—is a *conjunctive query*, i.e., a list of logical literals constraining the possible values of the set of variables they refer to; some of the variables may be *shared* between precondition and postcondition, meaning they must be instantiated with the same entities in both. Hence, they can be used to formalise dependencies between web service input and output.

These *pre/postcondition* definitions include a specification of the signature as well as further constraints on the input and output variables. The terminology is derived from the ontology of geographic datatypes. The postcondition specification additionally contains the *operation* description derived from the ontology of geospatial operations.

Example 3: (Functional Description) The *functional description F* (either request or web service advertisement) annotates the *intersection* operation on two polygons adhering to the *Gauß-Krüger* reference system.

$$F_{pre} : polygon(A) \wedge polygon(B) \wedge hasSRS(A, gk) \wedge hasSRS(B, gk)$$
$$F_{post} : polygon(C) \wedge intersection(A, B, C)$$
$$F_{share} : A, B$$

In the example above, the variables A and B are shared and so the meaning is that "C"—the output—is the intersection between the two input polygons and not between some other anonymous polygons. While this may seem a marginal observation at first sight, it is important to note that the most wide-spread formalism used for semantically describing Web services is Description Logics (DL), where it is not possible to formulate this kind of dependencies. Further, the input/output dependencies exhibited by WPS descriptions/requests involve all sorts of other, more subtle, issues regarding whether the variables are instantiated in a correct way, not using the same value to instantiate two input variables, not using a shared variable as if it was not shared, etc. A substantial part of our work concentrates on formulating the In-Out-Aware Plug-in match in a way so that all these details are treated right

3.3 Standard matches

In the following, we present different standard notions of *matchmaking* between the functional descriptions introduced in the previous section. All of them have several drawbacks, which will be discussed in detail.

3.3.1 Matchmaking with shared variables: predicate matches

In this section, we discuss the notion of ***predicate matches*** introduced in [23]. But instead of treating a *functional description* as a logical implication between pre- and postconditions, we build the conjunction of both that is: $R_{pred} = R_{pre} \wedge R_{post}$ and $S_{pred} = S_{pre} \wedge S_{post}$, with $R_{pre}(S_{pre})$ being the request's (web service's) *precondition*, $R_{post}(S_{post})$ the request's (web service's) *postcondition*. Hence, a functional description is handled as one *conjunctive query* and it is possible to share variables between pre- and postconditions and therefore to formulate dependencies between web service in- and output.

Note: For the same reason, it is impossible to distinguish between these conditions when matching two functional descriptions. In order to do so, one has to introduce e.g. some predicates marking the variables appearing in either pre- or postconditions. Furthermore, no state changes between pre- and poststates can be considered since they are represented in one *conjunctive query.*

Treating a functional description as a single conjunctive query allows two different ways of matchmaking that only differ in the direction of the query containment relation. In the following, we only introduce one of them, namely the generalized predicate match:

$$\begin{aligned} &Generalized\ Predicate\ Match: \\ &match_{gen\_pred}(S, R) = S_{pred} \subseteq R_{pred} \end{aligned} \tag{3.6}$$

The *generalized predicate match* holds, if the *functional description* representing a web service is more restrictive than the *functional description* representing a request. It is therefore especially useful for matching simple queries such as *"give me all services that compute overlay"* with quite detailed web service advertisements. However, there are some cases, where this match possibly delivers wrong results:

- If R_{pre} is more specific/restrictive than S_{pre}, the *generalized predicate match* does not succeed, although the web service could execute on the input provided by the requester. Hence, this match possibly decreases *recall.*
- If S_{pre} is more specific than R_{pre}, the *generalized predicate match* succeeds although the user cannot guarantee to provide an input that is acceptable by *S*. Hence, this match possibly decreases *precision*.

Note: Since we include the input and output types into the pre- and postcondition definitions, the possible drawbacks not only refer to the behaviour of the operations but also to their types. The *generalized predicate match* <u>reverses</u> the function subtype relation for the input types. Hence, even if the *generalized predicate match* succeeds, it is not sure if the (requester-) delivered input values are syntactically inside the range of values that are acceptable by the discovered web services.

Example 4: (Predicate Matches) R requests an *overlay* calculation on *polygons A* and *B,* without exactly specifying which type of *overlay.* Both input polygons adhere to the coordinate reference system *Gauß-Krüger*:

$$R_{pre} : polygon(A) \wedge polygon(B) \wedge hasSRS(A, gk) \wedge hasSRS(B, gk)$$

$$R_{post} : polygon(C) \wedge overlay(A, B, C)$$

$$R_{share} : A, B$$

S advertises the difference calculation on *polygons.* Both input polygons should belong to a common (anonymous) projected spatial reference system:

$$S_{pre} : polygon(X) \wedge polygon(Y) \wedge hasSRS(X, SRS) \wedge hasSRS(Y, SRS) \wedge projSRS(SRS)$$

$$S_{post} : polygon(Z) \wedge difference(X, Y, Z)$$

$$S_{share} : X, Y$$

Obviously, the WPS could deliver a suitable output when executed with the requests input.[4] Nonetheless, neither of the *predicate matches* succeeds. In order to apply them, we transfer *R* and *S* to the following *conjunctive queries*:

Request R:

$$q(A, B, C) : -polygon(A) \wedge polygon(B) \wedge hasSRS(A, gk) \wedge hasSRS(B, gk) \wedge$$
$$polygon(C) \wedge overlay(A, B, C)$$

[4] Assuming the (taxononomic) relationships from the domain ontologies that a difference calculation *is an* overlay calculation and Gauß-Krüger *is a specific* projected spatial reference system.

Service S:

$$q(X,Y,Z,SRS) : -polygon(X) \wedge polygon(Y) \wedge hasSRS(X,SRS) \wedge hasSRS(Y,SRS) \wedge projSRS(SRS) \wedge polygon(Z) \wedge difference(X,Y,Z)$$

Applying the *generalized predicate match* to both descriptions does not succeed since the web services precondition is more generic than the requests precondition. Applying the reversed version of the generalized predicate match would also not succeed since the web services postcondition is more specific (*difference*) than the requests postcondition (*overlay*).

3.3.2 Matchmaking based on function subtypes: Plug-in match

In this section, we present a matchmaking technique that is capable of avoiding the drawbacks from the predicate matches identified in the previous section. This can be done by comparing the preconditions and postconditions separately. However, this comes at a cost: the *shared variables* and therefore the dependencies between input and output can no longer be considered in the matchmaking process. By comparing the pre- and postconditions separately, a functional description is no longer handled as a conjunction of pre- and postconditions. In the following, we introduce the *Plug-In match,* and exemplarily show for Example 4 from the previous section why it is unsuitable for our purposes. Afterwards, we formulate conditions on the use of the *shared variables* and extend the *Plug-in match* in order to meet these conditions.

The *Plug-in Match* is well known in the area of behavioural matchmaking between software components. It compares two functional descriptions based on the notion of *function subtypes* as introduced in Section 2.2. We define the *Plug-in match* based on query containment as follows:

$$\begin{aligned}&Plug-in\ match:\\&match_{plug\_in}(S,R) \equiv \left(R_{pre} \subseteq S_{pre}\right) \wedge \left(S_{post} \subseteq R_{post}\right)\end{aligned} \tag{3.7}$$

By matching a request *R* with a web service description *S* using the *Plug-in match*, the following conditions are ensured:

1. *If R_{pre} holds before executing the service, then S_{pre} also holds (this is directly expressed in the query containment relation). Therefore, the service can be executed with R_{pre}.*
2. *If we assume that, executing S on S_{pre} delivers S_{post} (which is a reasonable definition of a web service), then R_{post} also holds after execution (again, this is directly expressed in the query containment relation).*

These two conditions lead to the fact that *S* can be plugged-in for *R*. Since we use query containment for matchmaking and do not consider any variable renaming, it is impossible to maintain the roles "played" by the *shared variables* in the preconditions when comparing the postconditions. Hence, it is impossible to formulate dependencies between *input* and *output*. Using the definition of a *Plug-in match* from Eq. 3.2, the *shared variables* in the postcondition containment match are treated as if they were new variables. They could be replaced with arbitrarily chosen variable names without any impact on the result of the matchmaking.

Yet, as described in Section 2, the dependencies between input and output and therefore the use of the *shared variables* in the matchmaking process are crucial for WPS. For this reason, the *Plug-in match* in the current form is unsuitable for our purposes.

To further explain these drawbacks, we briefly recall the characterisation of *query containment* and how it relates to *containment mappings* as introduced in Section 2.3: a *query containment* relation between two queries holds, if and only if there is a *containment mapping* from the containing to the contained query. Hence, we can assume that, if the *Plug-in match* holds between some request specification R and some web service specification S, then there exists some (at least one) *input mapping* from S_{pre} to R_{pre} and additionally some (at least one) *output mapping* from R_{post} to S_{post}.

Example 5: (Plug-in match) In the following, we apply the *Plug-in match* to Example 4. We translate the pre- and postconditions of S and R to the following *conjunctive queries*:

Request R:

$$q(A, B, gk) : -polygon(A) \wedge polygon(B) \wedge hasSRS(A, gk) \wedge hasSRS(B, gk)$$

$$q(A, B, C) : -polygon(C) \wedge overlay(A, B, C)$$

Web Service S:

$$q(X, Y, SRS) : -polygon(X) \wedge polygon(Y) \wedge hasSRS(X, SRS) \wedge hasSRS(Y, SRS) \\ \wedge projSRS(SRS)$$

$$q(X, Y, Z) : -polygon(Z) \wedge difference(X, Y, Z)$$

Applying the *Plug-in match* succeeds and yields the following input and output mappings:

Input mapping 1: $im_1(X) = A, im_1(Y) = B, im_1(SRS) = gk$
Input mapping 2: $im_2(X) = B, im_2(Y) = A, im_2(SRS) = gk$
Output mapping: $om(C) = Z, om(A) = X, om(B) = Y$

Intuitively, the fact that we get two different input mappings means that the web service can be executed with two different permutations of the requests input variables. However, since the web service calculates the difference operation, it delivers only for one of them a suitable result, namely the difference between A and B instead of B and A. The *Plug-in match* as introduced above does not filter out those permutations of input variables that possibly deliver wrong results. As described above, the shared variables A and B (resp. X and Y) are simply treated as new variables in the postcondition containment check.

4 The In-Out-Aware Plug-in match

In the following, we introduce a technique of matchmaking that combines the advantages of both, the predicate matches (shared variables) as well as Plug-in matches (function subtypes). In order to do this, we provide formalised conditions on the input and output containment mappings in Section 4.1. The role of each condition in the matching process is shown by means of concrete discovery examples. Afterwards, Section 4.2 extends the Plug-in match in order to satisfy these conditions. Section 4.3 concludes with a concrete discovery example that shows the benefits of the In-Out-Aware Plug-in match.

4.1 Formalised correctness conditions

In the following, we give some formal conditions on the input- (*im*) and output (*om*) mappings derived from Plug-in matching two functional descriptions. The *Plug-in match* should only succeed if:

(1) *im* is *injective...*

(a) ... with respect to R_{share}; that is:

$$\forall x, x' \in S_{pre},\ im(x) = im(x') \in R_{share} \Rightarrow x = x'$$

(b) ... over all; that is: $\forall x, x' \in S_{pre}, im(x) = im(x') \Rightarrow\ x = x'$

(2) $\forall a \in R_{share}$**,** there exists $x \in S_{pre}$ with $im(x)=a$
(3) $\forall a \in R_{share}, \forall x \in S_{pre} : im(x) = a \Rightarrow x \in S_{share}$
(4) $\forall a \in R_{share} : im^{-1}(a) = om(a)$

Condition **(1a)** ensures that the WPS` distinct input variables are instantiated with distinct user input, as long as this input is marked as *shared*. Condition **(1a)** is a slight relaxation of Condition **(1b)**, which requires distinct web service input to be instantiated with distinct user input in any case, independent from the shared variables. Condition **(2)** ensures that every of the request's *shared variables* is used as input, when executing the WPS. Condition **(3)** goes one step further since it requires that every of the request's *shared variables* is not only used as input but also appears in the output when executing the service (i.e. that the *shared variables* are used as such). This ensures (together with condition **(4)**) that the web service "*behaves"* in the way requested.

In the following, a more detailed description of these conditions and their impact on the discovery process is given by means of concrete discovery examples.

4.1.1 The role of condition 1

The following example further explains Condition 1 (injectivity). Consider a user request *R* that requests an overlay calculation on two polygons *A* and *B*:

$$R_{pre} : polygon(A) \wedge polygon(B)$$
$$R_{post} : polygon(C) \wedge overlay(A, B, C)$$
$$R_{share} : A, B$$

A web service *S* advertises an overlay calculation on two polygons *X* and *Y.*

$$S_{pre} : polygon(X) \wedge polygon(Y)$$
$$S_{post} : polygon(Z) \wedge overlay(X, Y, Z)$$
$$S_{share} : X, Y$$

Obviously, the service *S* delivers the requested functionality. However, applying the *Plug-in match* succeeds with exactly four different input mappings. For two of these input mappings, namely those that map X to A and Y to A (resp. X to B and Y to B), the service delivers a wrong result. For example, consider the input mapping that maps both variables X and Y to the single variable A. When executed with this mapping, the services input variables X and Y are both instantiated with the single user input A. As can be seen in Fig. 2 in Section 2, executing an overlay on a single polygon delivers either a polygon with no geometric extent (e.g. in case of the difference operation) or a polygon with the same geometric extent as the single input polygon (e.g. in case of the intersection operation). Hence, we want to exclude those cases, where (at least) two distinct web service input variables map to a single (shared) input variable of the request. This is exactly the purpose of condition **(1a)**, which requires the input mapping to be injective with respect to the requests shared variables.

Condition **(1a)** is a slight relaxation of condition **(1b)** which requires the input mapping to be injective over all variables. It is not a priori clear which of the two conditions is more adequate in which situation; both are thinkable. In what follows, we concentrate on condition **(1b)** which is slightly easier to present and digest; we will summarize how our definitions need to be adapted to handle **(1a)** instead.

To further explain condition **(2),** we introduce the following example.

4.1.2 The role of Condition 2

The role of Condition 2 is further explained with the following example: *F* is a *functional description* representing a request. *F* requests some overlay calculation on two polygons that are defined with respect to the same (not further specified) spatial reference system *SRS*. The resulting overlay polygon *C* should also refer to *SRS*. Hence, the variable *SRS* is (besides the input polygons *A* and *B*) shared between pre- and postconditions.

$$\begin{aligned}
&F_{pre} : polygon(A) \wedge polygon(B) \wedge hasSRS(A, SRS) \wedge hasSRS(B, SRS)\\
&F_{post} : polygon(C) \wedge overlay(A, B, C) \wedge hasSRS(C, SRS)\\
&F_{share} : A, B, SRS
\end{aligned}$$

E advertises an overlay calculation on two polygons. It ensures that the output polygon is defined with respect to the *Gauß-Krüger* reference system

$$\begin{aligned}
&E_{pre} : polygon(X) \wedge polygon(Y)\\
&E_{post} : polygon(Z) \wedge overlay(X, Y, Z) \wedge hasSRS(Z, gk)\\
&E_{share} : X, Y
\end{aligned}$$

Obviously, the web service *E* can not exactly deliver the functionality requested by *F*. It cannot ensure that the output polygon has the same spatial reference system as the input polygons, since it does not consider any spatial reference system for the input at all (and the output polygon adheres to the *Gauß-Krüger* reference system). Hence, the request's shared variable *SRS* is not mapped to by *im* (i.e. *im* is not surjective with respect to F_{share}). Condition **(2)** ensures that every of the request's *shared variables* is used as input, when executing the WPS. Hence, condition **(2)** prevents the *Plug-in match* from succeeding in this case. Still, condition **(2)** is not sufficient as we show with the following example.

4.1.3 The roles of condition 3 and 4

F is a functional description of an overlay calculation on two polygons that are defined with respect to the same (not further specified) spatial reference system *SRS*. The resulting overlay polygon C should also refer to SRS. Hence, the variable SRS is (besides the input polygons A and B) shared between pre- and postconditions.

$$\begin{aligned}&F_{pre}: polygon(A) \wedge polygon(B) \wedge hasSRS(A, SRS) \wedge hasSRS(B, SRS)\\&F_{post}: polygon(C) \wedge overlay(A, B, C) \wedge hasSRS(C, SRS)\\&F_{share}: A, B, SRS\end{aligned}$$

E' annotates an overlay calculation on polygons that both refer to a common spatial reference system REF. E' ensures that the output polygon refers to some reference system $REFSYS$ which is not necessarily the same as for the input.

$$\begin{aligned}&E'_{pre}: polygon(X) \wedge polygon(Y) \wedge hasSRS(X, REF) \wedge hasSRS(Y, REF)\\&E'_{post}: polygon(Z) \wedge overlay(X, Y, Z) \wedge hasSRS(Z, REFSYS)\\&E'_{share}: X, Y\end{aligned}$$

Now let F be a request and E' a web service description. Hence, the web service has **less** shared variables than the request. In this case, the *Plug-in match* should not succeed since the web service can not exactly deliver the requested functionality. E' cannot ensure that the output polygon has the same spatial reference system as the input since the variable REF is not shared between pre- and postconditions. Nonetheless, the *Plug-in match* succeeds and yields the following input/output mappings:

$$\begin{aligned}&im_1(X) = A, im_1(Y) = B, im_1(REF) = SRS\\&im_2(X) = B, im_2(Y) = A, im_2(REF) = SRS\\&om(C) = Z, om(A) = X, om(B) = Y, om(SRS) = REFSYS\end{aligned}$$

As described above, we do not want the *Plug-in match* to succeed. Condition **(1b)** is obviously not violated for both input mappings. Condition **(2)** is not violated since both input mappings are surjective with respect to F_{share}. Hence, condition **(2)** is obviously not sufficient. To prevent the *Plug-in match* from succeeding is the purpose of condition **(3)** which is violated for both input mappings since the variable REF is not shared.

If—vice versa—F represents a web service description and E' a discovery request, then the web service has **more** shared variables than the request. In this case, the *Plug-in match* should succeed since the web service F can obviously deliver the requested functionality. F ensures that the output polygon has the same spatial reference system as the input although the request E' only requires the output polygon to have some spatial reference system (not necessarily the same as for the input).

The following input/output mappings exist:

$$\begin{aligned}&im_1(A) = X, im_1(B) = Y, im_1(SRS) = REF\\&im_2(A) = Y, im_2(B) = X, im_2(SRS) = REF\\&om(Z) = C, om(X) = A, om(Y) = B, om(REFSYS) = SRS\end{aligned}$$

Condition **(1)** is obviously not violated for both input mappings. Condition **(2)** is also not violated since both input mappings are surjective with respect to E'_{share}. Condition **(3)** is not violated since each of E's *shared variables* is mapped to by a *shared variable* of F.

Conditions **(1)**, **(2)** and **(3)** also imply that im is a *bijection* between a subset of F_{share} and E'_{share} and we can assume that the inverse mapping im^{-1} exists for all of the request's *shared variables*. For im_2, condition **(4)** is obviously violated since it has no corresponding output mapping at all. Therefore, the *Plug-in match* only succeeds for im_1 with $im_1{}^{-1}(X) = A = om(X)$ and $im_1{}^{-1}(Y) = B = om(Y)$.

4.2 Definition and correctness of the in-out-aware plug-in match

To give a short summary of the conditions: Condition **(1a)** ensures that the WPS` distinct input variables are instantiated with distinct user input, as long as this input is marked as *shared.* Condition **(1a)** is a slight relaxation of Condition **(1b)**, which requires distinct web service input to be instantiated with distinct user input in any case, independent from the shared variables. Condition **(2)** ensures that every of the request's *shared variables* is used as input, when executing the WPS. Condition **(3)** goes one step further since it requires that every of the request's *shared variables* is not only used as input but also appears in the output when executing the service (i.e. that the *shared variables* are used as such). This ensures (together with condition **(4)**) that the web service "*behaves"* in the way requested. From conditions **(1)** to **(3)**, it directly follows that a web service must have **at least** as many shared variables as a request. Conditions **(1a)**, **(2)** and **(3)** also imply that *im* is a *bijection* between a subset of S_{share} and R_{share}. Hence we can assume that the inverse mapping im^{-1} exists for all of the request's *shared variables* R_{share}. In condition **(4)**, we require this *inverse input mapping* to be equal to the *output mapping*. This ensures that request and web service agree on the way, the output is computed from the input. Hence, condition **(4)** allows the consideration of dependencies between web service input and output in the matchmaking. Condition **(4)** is obviously the strongest one and definitely requires *im* to meet conditions **(1a)**, **(2)** and **(3)**. In order to satisfy all of these conditions when Plug-in matching two *functional descriptions*, we define the following *In-Out-Aware Plug-in match*:

Denoting with $x_1...x_n$ ($a_1...a_m$) the web service's (request's) precondition variables, with $y_1...y_l$ ($b_1...b_k$) the web services (requests) postcondition variables. We define the following two queries: [5]

$$\begin{aligned} S_Q &\equiv S_{post} \wedge order(x_1, ..., x_n) \wedge \\ &\quad \left(\bigwedge_{x_i \in S_{share}} shared(x_i) \right) \wedge \left(\bigwedge_{x_i, x_j \in S_{pre}, i \neq j} dif\left(x_i, x_j\right) \right) \\ R_Q &\equiv R_{post} \wedge order(im(x_1), ..., im(x_n)) \wedge \\ &\quad \left(\bigwedge_{a_i \in R_{share}} shared(a_i) \right) \wedge \left(\bigwedge_{a_i, a_j \in X, i \neq j} dif\left(a_i, a_j\right) \right) \end{aligned} \tag{4.1}$$

with

$$X = R_{share} \cup \left\{ a_i \in R_{pre} \middle| \exists y_j \in S_{pre} : im\left(y_j\right) = a_i \right\}$$

The *order(...)*-predicate is used to filter out those input mappings *im* that cannot fulfil condition **(1b)** (*injectivity* over all variables).The *dif(..)* predicates ensure condition **(2)** (*surjectivity* with respect to R_{share}). We have to include the *dif(..)*-predicate into R_Q not only for all of the requests *shared variable*s but also for all variables that are mapped to by *im* (hence the definition of *X*) in order to ensure that *im* is still *surjective* concerning R_{share} even if web service and request only have one *shared variable* each.

The *shared(.)*-predicates guarantee that *im* maps only web service variables that are *shared* to the request's *shared variables* and hence ensures condition **(3)**. All of these

[5] We assume that the *dif(..)* predicate is symmetric.

predicates are required to ensure condition **(4).** We define the *In-Out-Aware Plug-in match*:

$$\begin{aligned}&In-Out-Aware\ Plug-in\ match:\\&1.\ R_{pre}\subseteq S_{pre}\\&2.\ \text{Test for every } im \text{ whether}: S_Q\subseteq R_Q\end{aligned}\tag{4.2}$$

Theorem 1: (In-Out-Aware Plug-in match) The *In-Out-Aware Plug-in match* succeeds only if the following conditions hold:

(1) im is *injective* over all; that is: $\forall x,x' \in S_{pre}, im(x)=im(x') \Rightarrow x=x'$
(2) $\forall a\in R_{share}$**,** there exists $x\in S_{pre}$ with $im(x)=a$
(3) $\forall a\in R_{share}, \forall x\in S_{pre}: im(x)=a\Rightarrow x\in S_{share}$
(4) $\forall a\in R_{share}: im^{-1}(a)=om(a)$

Proof

Condition (1b)

Suppose S Plug-in matches R using the *In-Out-Aware Plug-in match* and the input mapping im is non-injective. Then there exist some $x_i, x_j\in S_{pre}, i\neq j$ such that $im(x_i)=im(x_j)=a_k\in R_{pre}$. Consider only the *order(...)* predicates in S_Q and R$_Q$: $....order(...,x_i,...,x_j,...)....\subseteq ...order(...,a_k,...,a_k,...)....$ Since no possible containment mapping om exists, this contradicts our assumption that im is non-njective.

Condition (2)

Suppose S Plug-in matches R using the *In-Out-Aware Plug-in match* and the input mapping im is *non-surjective* with respect to R_{share}. From non-surjectivity, it follows that some $a_i\in R_{share}$ exists that is not mapped to by im. With the definition of query containment, it follows that the output mapping om maps a_i to some $x_k\in S_{pre}$ $(om(a_i)=x_k)$. Again, with the *query containment* definition it follows that there exists some $a_j\in R_{pre}$ that is different from a_i ($i\neq j$) with $im(x_k)=a_j$. Therefore (from the definition of R_Q) some predicate $dif(a_i,a_j)\in R_Q$ must exist.

Remember that we already had that $om(a_i)=x_k$. The *order(...)*, ensures that also $om(a_j)=x_k$ and with the definition of R_Q some $dif(om(a_i),om(a_j))=dif(x_k,x_k)\in S_Q$ must exist. This is not the case and therefore contradicts our assumption that im is non-surjective.

Condition (3)

Suppose S Plug-in matches R using the *In-Out-Aware Plug-in match* and the input mapping im maps some variable $x_i\in S_{pre}$ to some variable $a\in R_{share}$. With the definition of query containment (and the *shared(.)* predicates), it follows that om maps a to some $x_j\in S_{share}$ $(om(a)=x_j)$. The order(..)-predicate ensures that also $om(a)=x_i$. Hence $om(a)=x_i=x_j$ and $x_i\in S_{share}$.

Condition (4)

Suppose S Plug-in matches R using the *In-Out-Aware Plug-in match* and there exists an $a\in R_{share}$ with $im(a)^{-1}\neq om(a)$. Then there exists a $x_i\in S_{pre}$ with $im^{-1}(a)=x_i$ and a $x_j\in S_{share}$ with $om(a)=x_j$ and $i\neq j$ ($x_j\in S_{share}$ follows from the *shared(.)* predicates). The order(..) ensures that also $om(a)=x_i$ and hence $im^{-1}(a)=x_i=om(a)=x_j$.

4.3 Example

In order to show the benefits of the In-Out-Aware Plug-in match, we briefly recall the Example from Section 4.1.3:

F is a functional description of an overlay calculation on two polygons that adhere to the same (not further specified) spatial reference system *SRS*. The resulting overlay polygon *C* should also adhere to *SRS*. Hence, the variable *SRS* is (besides the input polygons *A* and *B*) shared between pre- and postconditions.

$$\begin{aligned}&F_{pre} : polygon(A) \wedge polygon(B) \wedge hasSRS(A, SRS) \wedge hasSRS(B, SRS)\\&F_{post} : polygon(C) \wedge overlay(A, B, C) \wedge hasSRS(C, SRS)\\&F_{share} : A, B, SRS\end{aligned}$$

E describes an overlay calculation on polygons that adhere to a common spatial reference system *REF*. *E* ensures that the output polygon adheres to some reference system *REFSYS* which is not necessarily the same as for the input.

$$\begin{aligned}&E_{pre} : polygon(X) \wedge polygon(Y) \wedge hasSRS(X, REF) \wedge hasSRS(Y, REF)\\&E_{post} : polygon(Z) \wedge overlay(X, Y, Z) \wedge hasSRS(Z, REFSYS)\\&E_{share} : X, Y\end{aligned}$$

Now let *F* again be a request and *E* a web service description. Remember, we do not want the *Plug-In Match* to succeed since the request has more *shared variables* than the web service. Applying **Step 1** of the *In-Out-Aware Plug-in match* yields the following mappings:

$$\begin{aligned}im_1(X) = A, im_1(Y) = B, im_1(REF) = SRS\\im_2(X) = B, im_2(Y) = A, im_2(REF) = SRS\end{aligned}$$

Building the service query E_Q yields:

$$\begin{aligned}&polygon(Z) \wedge overlay(X, Y, Z) \wedge hasSRS(Z, REFSYS) \wedge order(X, Y, REF) \wedge\\&shared(X) \wedge shared(Y) \wedge dif(X, Y) \wedge dif(X, REF) \wedge dif(Y, REF)\end{aligned}$$

Building the requests query F_Q for im_1 yields:

$$\begin{aligned}&polygon(C) \wedge overlay(A, B, C) \wedge hasSRS(C, SRS) \wedge order(A, B, SRS) \wedge\\&shared(A) \wedge shared(B) \wedge shared(SRS) \wedge dif(A, B) \wedge dif(A, SRS) \wedge dif(B, SRS)\end{aligned}$$

The containment check fails since the variable *REF* is not shared.

Building the requests query F_Q for im_2 yields:

$$\begin{aligned}&polygon(C) \wedge overlay(A, B, C) \wedge hasSRS(C, SRS) \wedge order(B, A, SRS) \wedge\\&shared(A) \wedge shared(B) \wedge shared(SRS) \wedge dif(A, B) \wedge dif(A, SRS) \wedge dif(B, SRS)\end{aligned}$$

The containment check fails due to the conflict between *overlay(...)* and *order(...)* (furthermore—as above—it fails since the variable *REF* is not shared).

Now let *E* be a request and *F* a web service description. Remember, we want the *Plug-in match* to succeed for one of the input mappings that has a corresponding output mapping. Applying **Step 1** of the *In-Out-Aware Plug-in match* yields the following mappings:

$$im_1(A) = X, im_1(B) = Y, im_1(SRS) = REF$$
$$im_2(A) = Y, im_2(B) = X, im_2(SRS) = REF$$

Building the service query yields:

$$polygon(C) \wedge overlay(A, B, C) \wedge hasSRS(C, SRS) \wedge order(A, B, SRS) \wedge$$
$$shared(A) \wedge shared(B) \wedge shared(SRS) \wedge dif(A, B) \wedge dif(A, SRS) \wedge dif(B, SRS)$$

Building the requests query for im_1 yields:

$$polygon(Z) \wedge overlay(X, Y, Z) \wedge hasSRS(Z, REFSYS) \wedge order(X, Y, REF) \wedge$$
$$shared(X) \wedge shared(Y) \wedge dif(X, Y) \wedge dif(X, REF) \wedge dif(Y, REF)$$

The containment check succeeds. Note: The output mapping is not injective since *om* (*REFSYS*)=*SRS* and *om*(*REF*) = *SRS*, but this is not required. We skip the containment check for the input mapping im_2 since it obviously fails due to the conflict between the use of variables in *overlay(...)* and *order(...)*.

In the current form, the In-Out-Aware Plug-in match satisfies condition **(1b)**. In order to refer to condition **(1a)** instead of **(1b)**, the scope of the *order(...)* predicate has to be revised by including only a subset of the web services precondition variables. Instead of including each of the web service precondition variables into the *order(...)* predicate in S_Q (and the range of the input mapping into the *order(...)* predicate in R_Q), only those web service precondition variables have to be included, that map to shared variables of the request.

5 Preliminary empirical results

In this section, we show the application of the presented approach within the semantic web service framework WSMO [24]. WSMO is an upper level ontology that provides a conceptualization for various aspects related to semantic web services and the semantic web. It comprises elements such as WSMO ontologies, WSMO web services, WSMO Goals. WSMO ontologies provide the shared and formal background knowledge to be used in all other WSMO elements. WSMO Goals are functional descriptions of user requests, WSMO Web Services represent functional descriptions of web services. WSMO comes with a family of formal languages called WSML [25]. WSML is based on well know logical language paradigms such as FOL, LP or DL. The WSML-variant corresponding to datalog is called WSML-Flight.

Our experimental evaluation is preliminary in that we considered only one small use case relating to our illustrative overlay operations example, We outline our implementation in Section 5.1, describe our test example in Section 5.2, and give our empirical runtime results in Section 5.3.

5.1 Implementation

Our approach is implemented in WSMX [26], a reference implementation of WSMO. IRIS (Integrated Rule Inference System), the reasoner underlying WSMX, is a datalog reasoner extended with (locally) stratified ‘negation as failure’ and with extended support for data types and built-in predicates. While pure datalog only supports integers and strings, IRIS follows the line of WSML and supports all the XML schema data types. Furthermore it allows the use of an extended and extensible set of built-in predicates. IRIS supports different deductive database algorithms and evaluation strategies, like for example naïve or semi-naïve evaluation.

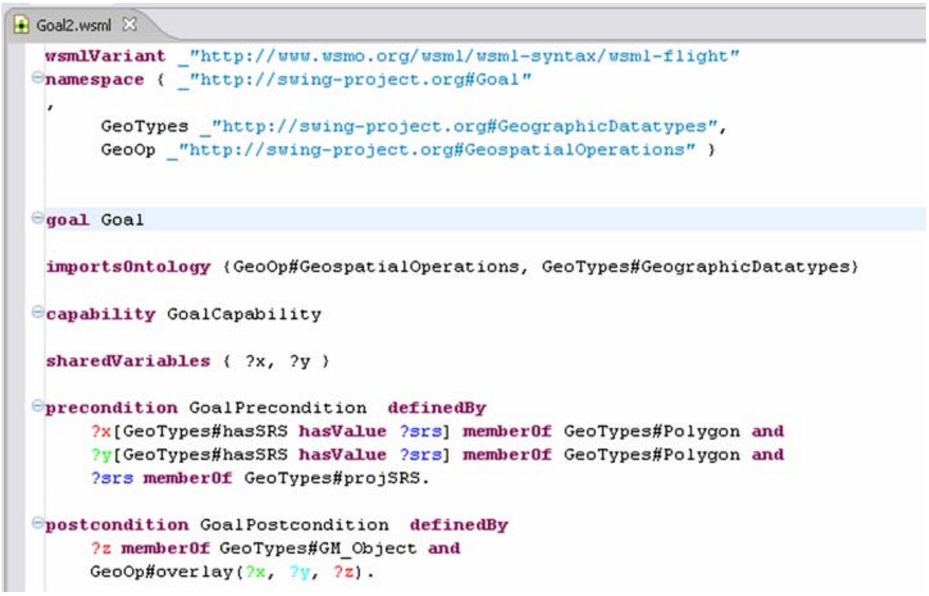
IRIS evaluates queries over a knowledge base consisting of facts (instances of predicates) and rules. The combination of facts, rules and queries (a logic program) forms the input to a reasoning task, as for example query answering or query containment. For query containment, IRIS can either return a Boolean value, saying whether yes or no, a query is contained within another query, or it can return the containment mapping, i.e. the mapping of the variables of one query to the variables of another query. Within IRIS, the "frozen facts" method [27] is used to determine query containment. The datalog engine supports query answering and query containment for WSML-Core and WSML-Flight.

In order to support the In-Out-Aware Plug-in match as defined in Section 4, we extended the WSMX discovery framework. The In-Out-Aware Plug-in match was implemented within WSMX on the basis of IRIS query containment.

5.2 Use case

For testing the WPS discovery, we used two settings. One setting, the 'Good Case', results in a match between the goal and the Web service, and the second setting, the 'Bad Case', does not result in a match. For the test setting, we used the two background ontologies from Section 3.1, translated to WSML-Flight. The WSMO Goal and WSMO Web Service descriptions that we use in both of our test settings import the two background ontologies.

Figure 3 show the user request. It requests, in short, a WPS that inputs two polygons and outputs a single geometric object that is the result of overlaying the input polygons. The



```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
namespace { _"http://swing-project.org#Goal"
,
     GeoTypes _"http://swing-project.org#GeographicDatatypes",
     GeoOp _"http://swing-project.org#GeospatialOperations" }


goal Goal

importsOntology {GeoOp#GeospatialOperations, GeoTypes#GeographicDatatypes}

capability GoalCapability

sharedVariables { ?x, ?y }

precondition GoalPrecondition  definedBy
     ?x[GeoTypes#hasSRS hasValue ?srs] memberOf GeoTypes#Polygon and
     ?y[GeoTypes#hasSRS hasValue ?srs] memberOf GeoTypes#Polygon and
     ?srs memberOf GeoTypes#projSRS.

postcondition GoalPostcondition  definedBy
     ?z memberOf GeoTypes#GM_Object and
     GeoOp#overlay(?x, ?y, ?z).
```

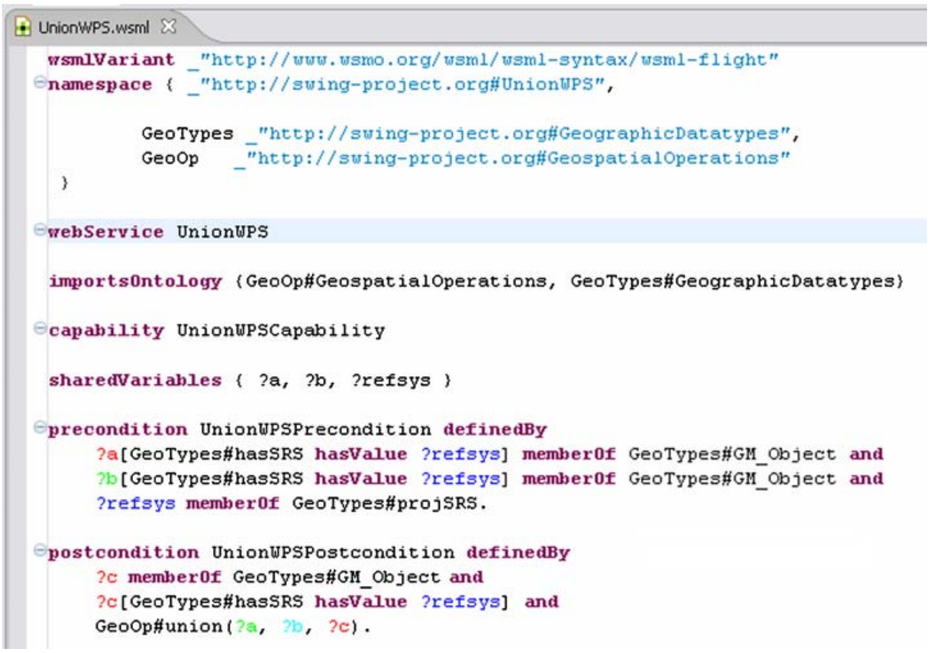
Fig. 3 WSMO test goal

precondition states that the two shared variables *?x* and *?y* denote members of type *Polygon* and both adhere to a spatial reference system that is member of (i.e. instance of) *projSRS*. The postcondition defines that the output must be the result of an overlay operation, without exactly specifying, which type of overlay (*intersection, union*, etc.).

Our corresponding Web service, *UnionWPS*, is shown in Fig. 4. It advertises a WPS that calculates the *union* operation on two geometric objects. Both input geometric objects (and the output geometric object) should be defined with respect to a common projected spatial reference system. To enforce the reference system to be common, the Web service shares three variables between its pre- and postconditions: both the variables denoting the geometric objects to be used in the *union* operation, as well as the variable denoting the reference system.

5.2.1 Good case

Using our prototype to execute discovery on the example goal results in a match. During the discovery execution we do not see anything of the internal behaviour of the discovery engine. The background ontologies provide the necessary information to relate concepts from the goal with concepts from the Web service. For example, the geographic datatypes ontology states that *GeoTypes#Polygon* used in the goal precondition is a subconcept of *GeoTypes#GM_Object* as used in the Web service precondition. The geospatial operations ontology states that the union operation GeoOp#*union* in the Web Service postconditions implies the overlay operation *GeoOp#overlay* as used in the goal postcondition.



```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
namespace { _"http://swing-project.org#UnionWPS",

        GeoTypes _"http://swing-project.org#GeographicDatatypes",
        GeoOp    _"http://swing-project.org#GeospatialOperations"
 }

webService UnionWPS

importsOntology {GeoOp#GeospatialOperations, GeoTypes#GeographicDatatypes}

capability UnionWPSCapability

sharedVariables { ?a, ?b, ?refsys }

precondition UnionWPSPrecondition definedBy
    ?a[GeoTypes#hasSRS hasValue ?refsys] memberOf GeoTypes#GM_Object and
    ?b[GeoTypes#hasSRS hasValue ?refsys] memberOf GeoTypes#GM_Object and
    ?refsys memberOf GeoTypes#projSRS.

postcondition UnionWPSPostcondition definedBy
    ?c memberOf GeoTypes#GM_Object and
    ?c[GeoTypes#hasSRS hasValue ?refsys] and
    GeoOp#union(?a, ?b, ?c).
```

Fig. 4 WSMO test web service

In the first step of the In-Out-Aware Plug-in match, the engine gets four containment mappings from the execution of the descriptions preconditions: *{[?A→?X, ?REFSYS→?SRS, ?B→?X], [?A→?X, ?REFSYS→?SRS, ?B→?Y], [?A→?Y, ?REFSYS→?SRS, ?B→?X], [?A→?Y, ?REFSYS→?SRS, ?B→?Y]}*. In the second step queries are build for each possible containment mapping, and submitted to the containment check. In our example we get exactly one positive containment check, with the mapping *[?A→?X, ?REFSYS→?SRS, ?B→?Y].*

5.2.2 Bad case

In this setting we use the same background ontologies and the same Web service and goal as in the good case. We only changed the goal and Web service descriptions in that they use different shared variables. The goal now wants to share the variables denoting both the polygons and the spatial reference system, while the Web service only shares the variables denoting the geometric objects. The In-Out Aware Plug-in match does not allow the goal to have more shared variables than the Web service, since then the web service can not exactly deliver the requested functionality. But as this is the case in our 'Bad Case' setting, the discovery on the example goal results in an empty set, i.e. we find no match.

5.3 Runtime

The runtime behavior is virtually identical for both the good case and the bad case. The discovery step took 1.500 milliseconds, i.e. 1,5 seconds, total time. It took 844 milliseconds to build up the example background ontologies, the goal and the Web service. Starting the discovery engine in WSMX, and adding the example Web service to it, took 31 milliseconds. Initially registering the ontologies at the IRIS reasoner took 218 milliseconds. The reasoning time itself is absolutely negligible. The first step of the In-Out-Aware Plug-in match, i.e. getting all containment mappings for the preconditions, takes 16 milliseconds. The second step, the containment checks with the queries build upon the containment mapping result takes also 15–16 milliseconds each. The total runtime of the second step (i.e. 312 milliseconds) is very similar to the runtime of the first step, including the above mentioned initial registration at the reasoning engine (i.e. 313 milliseconds only diverged minimally on repeated execution of our discovery example. Of course, the test example is very small and so it remains to be seen whether similar efficiency can be achieved in the context of larger examples, in particular of larger ontologies.

6 Related work

There are multiple approaches for web service discovery. In the following, we discuss those that are either closely related to our work by using e.g. a similar technique of matchmaking or that explicitly deal with WPS discovery.

In [28], a functional description consist of a list of input and output types, formalized as OWL-DL concepts. The dependencies between in- and output are then formalized using the notion of conjunctive DL queries. Obviously, since the relation from in- to output is represented in a single query (quite similar to the predicate matches in our approach), the matchmaking in [28] is not capable to distinguish between pre- and postconditions.

Moreover, the notion of conjunctive DL queries is quite new and not supported by current Semantic Web Services approaches such as OWL-S or WSMO.

The semantic matching in [29] has certain similarity to our work. It is a two step matchmaking, which performs signature in a first, and pre/post matching in a second step. The signatures are compared using subtype rules that have to hold on the input and output types. In a second step, the pre- and postconditions (in [29] called *InConstraints* and *OutConstraints*) are compared using the behavioural notion of *Plug-in match*. These constraints have the form of *Horn rules* and the matching establishes a θ-subsumption relation [30] between them, which has certain similarity to LP query containment. The differences to our approach are the following: Since in [29], the input/output types are not included in the specification of the *InConstraints* and *OutConstraints*, it is not possible to maintain the "roles" of the input and output variables when comparing the constraints. Since no variable renaming (or something similar) is mentioned in [29], it seems quite unclear how this problem is solved. Another difference is that no dependencies between *InConstraints* and *OutConstraints* can be formalised in [29], since no variables are *shared*.

Several approaches for discovering data-providing geospatial web services exist. For example [8] deal with OGC Web Feature Services [3]. The approach relies on DL ontologies that provide a shared vocabulary for the annotation of WFS or, more precisely, the data they offer. At discovery time, concept queries, expressed in terms of the shared vocabulary and classified in a subsumption hierarchy enable service requesters to discover WFS that provide suitable data. Since WFS do not require the requester to deliver a specific input, only the output is semantically annotated/formalised and therefore, these approaches are not suitable for dealing with WPS.

There are only a few approaches that explicitly deal with WPS discovery. For example, Lemmens [9] annotates a WPS or request as a single OWL-DL concept. The discovery is then based on standard OWL-DL concept subsumption. Due to the inherent restrictions of DL, the annotation of WPS as single DL concepts does not allow formalising dependencies between input and output. Moreover, the input- and output types (resp. pre- and postconditions) can not be treated separately, which obviously violates the notion of *function subtypes*.

The only previous approach that is able to capture WPS behaviour at a similar level of detail is that of Lutz [11]. It is based on a combination of DL and FOL. The input and output types are annotated as DL concepts, derived from a DL ontology of geographic datatypes (including semantic types such as *distance*). The pre- and postconditions that further constrain the input and output values are described by means of first-order formulas. Lutz [11] provides a FOL-based conceptualisation for the different distance calculations, such as euclidean or spherical distance. The provided conceptualisation is somehow a trade-off between the mathematical equations of *distance* and the lightweight (keyword-like) conceptualisation in our approach. It traces the different distance calculations back to the underlying space, such as sphere or plane. The discovery is then a two-step process. In a first step, DL subsumption testing is used to match the signatures based on the notion of function subtypes. In a second step, the FOL specifications (pre- and postconditions) of the request and the remaining WPS (from the first step) are compared using FOL-theorem proving. Although the approach presented in [11] is in principle more expressive than ours, it has some drawbacks. Using FOL and DL for annotating web services makes the annotation process a difficult task. Since web service providers and requester are usually no experts in logics, powerful user interfaces have to be developed to hide the complexity of the underlying logics and to make the approach applicable. Moreover, the reasoning process of FOL is quite unpredictable, even undecidable, which heavily threatens the efficiency of the whole process of WPS discovery.

7 Conclusion

This work presents an approach to WPS discovery based on the logic programming (datalog) paradigm. The discovery process relies on *functional descriptions* of requests and WPS that are equally structured: they include specifications of pre- and postconditions, formalised as *conjunctive datalog queries*. The pre- and postconditions comprise a specification of type signatures, derived from an *ontology of geographic datatypes,* as well as further constraints on the in- and output variables. The postcondition additionally contains the operation description, derived from a lightweight *ontology of geospatial operations.* The dependencies between input and output are formalised using *shared variables* that appear in both, pre- and postconditions. Several notions of matchmaking between functional descriptions based on *query containment* have been introduced, including the common notion of *Plug-in match*. Various conditions for Plug-in matching two functional descriptions have been identified. The *In-Out-Aware Plug-in match* has been developed in order to satisfy these conditions.

Thus, our approach satisfies the requirements for WPS discovery as introduced in Section 2, by considering *type signatures*, *constraints*, the performed/requested *operation* as well as the *relation* between input and output in the discovery process.

As described in the previous section, the only approach we are aware of that is able to capture WPS behaviour at a similar level of detail, meeting the above requirements, is the approach based on first order logics proposed by Lutz [11]. The obvious difference to our approach lies in the logics used: our approach makes do with datalog, i.e., with a less general form of logics. In that way, we are able to achieve our goal—a sufficient level of detail of semantic description and discovery—while using technology that is less complex. This is quite advantageous in several respects. First, the worst-case complexity of datalog is polynomial in "data size", i.e. in the number of entries of any underlying Geospatial knowledge base. The worst-case complexity is exponential only in the number of variables involved, which can be assumed to be reasonably small in practice. By comparison, first order logics are undecidable. Since runtime is a critical resource in practice, this is quite important.

Possibly even more important is the fact that the use of datalog also facilitates the comparatively easy creation of semantic annotations, i.e., of functional descriptions, simply because datalog is a much more restricted language than the DL or FOL languages used by related works. Still, in order to support users in formalising functional descriptions, sophisticated user interfaces as well as techniques for automated annotation need to be developed. It is realistic to assume that the development of such annotation-supporting techniques is feasible, for a language as simple as datalog (note, e.g., that SQL is much more complicated). For example, the work presented in [31] uses term-matching algorithms for the automatic annotation of Web Feature Services resp. the data they offer. The approach shows promise to be quite useful also for the automatic generation of WPS pre/postconditions as required for our approach.

Further, since the technique of matchmaking developed in this paper is independent from the conceptualisation of operations (as long as they are formalised in datalog), it can be applied to more complex characterisations of geospatial operations.

At the moment, our approach allows only rather abstract descriptions of geoprocessing functionality that rely on predicates and exclude e.g. arithmetics and numerical built-in predicates. The extension of the approach in this direction, e.g. in order to deal with arithmetical values inside the pre/postconditions and arithmetical comparison in the matchmaking process, is subject to future research.

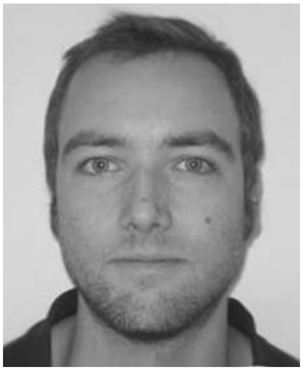
Acknowledgments This work was supported by the SWING project, which is co-funded by the European Commission under the sixth framework programme within contract FP6-26514. It has been carried out while the authors were affiliated with the Institute for Geoinformatics at the University of Muenster (Dr. Eva Klien, Daniel Fitzner) and the Digital Enterprise Research Institute (DERI), Innsbruck (Dr. Jörg Hoffmann).

The authors would like to thank Prof. Dr. Werner Kuhn and the members of his group, the Muenster Semantic Interoperability Lab (MUSIL) for their support. Further, we want to thank Nathalie Steinmetz who provided the proof of concept for the ideas by implementing them.

References

1. Bernard L, Fritzke J, Wagner RM (2005) Geodateninfrastruktur—Grundlagen und Anwendungen. Heidelberg, Wichmann
2. Brox C, Bishr Y, Senkler K, Zens K, Kuhn W (2002) Toward a geospatial data infrastructure for Northrine-Westfalia. Comput Environ Urban Syst 26(1):19–37
3. Open Geospatial Consortium Inc. (2009) OpenGIS web feature service (WFS) implementation specification, Version 1.1. Available at: http://www.opengeospatial.org/standards/wfs, Accessed: 28.1.2009
4. Keller U, Lausen H (2006) WSML deliverable D28.1 v.0.1—functional description of web services. Available at: www.wsmo.org/TR/d28/d28.1/v0.1/d28.1v0.1_20060113.pdf, Accessed 28.1.2009
5. Keller U, Lausen H, Stollberg M (2006) On the semantics of functional descriptions of web services. In Proceedings of the 3rd European Semantic Web *Conference (ESWC2006)*. Budva, Montenegro
6. Martin D, Burstein M, Hobbs J, Lassila O, McDermott D, McIllraith S, Narayanan S, Paolucci M, Parsia B, Payne T, Sirin E, Srinivasan N, Sycara K (2004) OWL-S: semantic markup for web services. W3C Member Submission 22 November 2004. Available at: http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/ Accessed: 28.1.2009
7. Gruber T (1993) A translation approach to portable ontology specifications. Knowl Acquis 5 (2):199–220
8. Lutz M, Klien E (2006) Ontology-based retrieval of geographic information. Int J Geogr Inf Sci (IJGIS) 20(3):233–260
9. Lemmens R (2006) Semantic interoperability of distributed geo-services. Ph.D.-thesis at Delft University of Technology
10. Open Geospatial Consortium Inc. (2009) OpenGIS web processing service specification, Version 1.0.0. Available at: http://www.opengeospatial.org/standards/wps, Accessed 28.1.2009
11. Lutz M (2007) Ontology-based descriptions for semantic discovery and composition of geoprocessing services. Geoinformatica 11(1):1–36
12. Chrisman N (1997) Exploring geographic information systems. Wiley, New York
13. Zaremski AM (1996) Signature and specification matching. Technical Report CMU-CS-96-103, Carnegie Mellon Computer Science Department, Ph.D. thesis
14. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580
15. Ullman JD (1989) Principles of database and knowledge-base systems—Volume II: the new technologies. Computer Science Press, Rockville
16. Ullman JD (1996) The database approach to knowledge representation. Proceedings of the 13th National Conference on Artificial Intelligence (AAAI1996). Portland, Oregon, AAAI Press, MIT Press, USA
17. Chekuri C, Rajaraman A (1997) Conjunctive query containment revisited. Theoretical Computer Science—Special Issue on the 6th International Conference on Database Theory—ICDT`97 239 (2):211–229

18. Egenhofer M (2002) Toward the geospatial semantic web. Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Science. McLean, Virginia, USA
19. Klien E, Lutz M, Kuhn W (2006) Ontology-based discovery of geographic information services-an application in disaster management. Comput Environ Urban Syst 30(1):102–123
20. Kuhn W (2005) Geospatial semantics: why, of what, and how? J Data Semantics III 3534:1–24
21. ISO/TC-211, ISO 19107:2003 Geographic Information — Spatial Schema. Available at: http://www.iso.org/iso/catalogue_detail.htm?csnumber=26012 Accessed: 28.1.2009
22. Fensel D, Lausen H, Polleres A, Stollberg M, Roman D, de Brijin J, Domingue J (2006) Enabling semantic web services. The web service modeling ontology. Springer Verlag, Berlin
23. Zaremski AM, Wing JM (1997) Specification matching of software components. ACM Trans Software Eng Meth 6(4):333–369
24. Roman D, Lausen H, Keller U (2005) Web service modeling ontology—WSMO final draft (13 April 2005). Available at: http://www.wsmo.org/TR/d2/v1.2/20050413/ Accessed: 28.1.2009
25. de Bruijn J, Lausen H, Krummenbacker R, Polleres A, Predoiu L, Kifer N, Fensel D (2005) The web service modeling language WSML—final draft (5 October 2005). Available at: http://www.wsmo.org/TR/d16/d16.1/v0.21/20051005/ Accessed: 28.1.2009
26. Haller A,Cimpian E, Mocan A, Oren E, Bussler C (2005) WSMX—a semantic service-oriented architecture. Processdings of the IEEE International Conference on Web Services (ICWS 2005). Orlando, Florida
27. Ullman JD (1997) Information integration using logical views. Proceedings of the 6th International Conference on Database Theory (ICDT1997). Delphi, Greece
28. Hull D, Zolin E, Bovykin A, Horrocks I, Sattler U, Stevens R (2006) Deciding semantic matching of stateless services. Proceedings of the 21st National Conference on Artificial Intelligence *(AAAI'2006)*. Boston, USA
29. Sycara K, Widoff S (2002) Larks: dynamic matchmaking among heterogeneous software agents in cyberspace. Auton Agent Multi-Agent Syst 5(2):173–203
30. Muggleton S, De Raedt L (1994) Inductive logic programming: theory and methods. J Logic Program 19 (20):629–679
31. Grcar, M. and E. Klien (2007) Using Term-matching algorithms for the annotation of geo-services. Proceedings of the Web Mining 2.0 Workshop in conjunction with ECML-PKDD 2007. Warsaw, Poland



Daniel Fitzner received his diploma in Geoinformatics at the University of Muenster, Germany. He works in the areas of Geographic Information Services, Geospatial Semantics and Semantic Web Services. His current research is focused on the investigation of formal languages for describing content and functionality of geographic information services together with the development of efficient search strategies. Since September 2008, he is a scientific staff member at the Fraunhofer IGD in Darmstadt.

Dr. Jörg Hoffmann obtained a PhD in CS from Freiburg University in 2002, with a thesis that won the annual ECCAI award for the best European dissertation in the field of Artificial Intelligence. Between 2002 and 2008, Jörg Hoffmann worked in various research projects at University Freiburg, Max Planck Institute for Computer Science, Cornell University, and the University of Innsbruck. He submitted his Habilitation at Innsbruck University in 2008, and is currently a Senior Researcher at SAP Research, Karlsruhe, Germany. He has published more than 50 papers in international journals and conferences, receiving several best paper awards. He is an Associate Editor of the Journal of Artificial Intelligence Research (JAIR) and Conference Chair of the 20th International Conference on Automated Planning and Scheduling (ICAPS'10).

Dr. Eva Klien is head of the Graphic Information Systems department at the Fraunhofer Institute for Computer Graphics since March 2008. Before joining Fraunhofer IGD, she has been working as researcher at the Institute for Geoinformatics (IfGI) at the University of Muenster. She received her PhD in Geoinformatics from the University of Muenster in 2008. Her research interest lies in methods for improving the usability of geographic information by enabling semantic interoperability in geographic web service environments. She has authored and co-authored a number of publications on geo-ontologies, semantic annotation of geographic information, and interoperability in geo-service infrastructures.