

# Using Semantic Services to Answer Queries with Quantified Incompleteness

Tatiana Kichkaylo and Vijay Karamcheti

New York University  
New York, NY 10012, USA  
{kichkay, vijayk}@cs.nyu.edu

## Abstract

The introduction of self-describing web services has opened up new avenues for the creation of information gathering agents, which are capable of discovering and employing such services at run time to answer user queries. It is desirable for such agents to not only build and execute a query plan, but also specify what information is not returned. In this paper we present a model for expressing the semantics of web services to provide information for such incompleteness analysis. The model relies on an external type system, which, in addition to types, specifies operations that can be performed on the types and properties of these operations. We also describe an algorithm for answering user queries in this model.

## 1 Introduction

The vision of the Semantic Web is that objects in the Internet are self-described in such a way so as to permit programmatic agents to reason about them and use them. Applied to web-accessible services, this means that a service description provides not only information about the service interface (how the service can be invoked), but also about preconditions and effects of the service invocation, including semantics of the returned values.

One application of such self-described services is information gathering agents [10, 8, 9]. Agents could answer a user query by automatically discovering and invoking services that provide relevant information. It is also desirable that such an agent could reason about the cost of answering the query and completeness of the answer.

While appealing, the problem of constructing a model for semantic web service descriptions is hard, because such a model needs to support thousands of services belonging to different providers. Several models have been proposed to describe services [16, 12], interactions between them [4], and to create large-scale repositories of service descriptions [15]. We believe that the information provided by these models is not sufficient for an information gathering agent of the kind mentioned above. WSDL and UDDI do not provide information about the semantics of returned answers. In OWL-S [12], although such information can be folded into precondition formulas, it is not required by the standard.

In this paper we propose a model for semantic service description that allows for quantification of (in)completeness of a query answer, as opposed to just detecting the fact that the answer might be incomplete. We limit ourselves to actions that only *read* data encapsulated by the service, as opposed to modifying the state of the world. We also describe our representation of a query plan as a combination of ser-

vice calls and local database operations, and present an algorithm for constructing such plans.

## 2 Modeling Semantics of Web Services

The two most popular approaches for describing semantics of an information service are the relational model and description logics.

The relational model is widely accepted in the database community. It is simple, well understood, and fast algorithms have been developed for processing of relational databases. Several information gathering systems have adopted the relational model extended with binding patterns [8, 9]. Different service vendors can either commit to a global schema at design time of the services, or provide descriptions of their services in terms of a global schema using non-trivial SQL queries. The former approach is administratively infeasible, while the latter does not scale with the number of services that can be used to answer a query.

Description logics (DL) are another model that has been used in the multi-database community [3, 10]. DL is a subset of first-order logic that can be used to describe complex ontologies in terms of concepts, roles, and relations between them. Description logics have become even more important now, with the growing popularity of Semantic Web and ontology-based data integration. The recently proposed language for specifying ontologies on the web, OWL [11], is based on the DL model.

The flexibility of description logics comes at the cost of efficiency. With the increased expressiveness of a DL, the complexity of reasoning about concept subsumption goes from polynomial to undecidable. Creation of an automated information gathering agent requires performing efficient search over expressive service descriptions, which involves such reasoning. Therefore, we choose a combined approach, where an ontology-based description of a service is precompiled into a relational description. The relational description we use is based on a type system which has simple hierarchical structure, and allows for fast subsumption reasoning.

### 2.1 Type system

Our type system defines types and operations on them.

There are **basic** and **complex types**. The basic (primitive) types represent scalar values, such as String, Number, or Location. Complex types describe tuples of values. Using database terminology, complex types can be viewed as relations (tables).

For example, a complex type for general information

about flights can be defined in terms of other types as follows

```

TYPEDEF AbstractFlight {
  Location      from, to;
  DateTime      departure;
  TimeInterval  duration;
  String        flightno; }

```

Values of types can be used in expressions involving equalities and inequalities. To associate meaning with the latter, we assume that a total order relation is defined for every type used in an inequality.

Complex types can be mapped to OWL classes and properties with restrictions on cardinality, domain, and range. For example, `from` is a property with domain `AbstractFlight` and range `Location`, and class `AbstractFlight` has an “exactly one” cardinality restriction on property `from`.

An **operation** is defined by its name, types of parameters, and a result type. An operation may have any number of arguments. However, all examples in this paper refer only to binary operations. We assume referential transparency (i.e. we can substitute any expression for an equal one).

For each operation, **monotonicity** information may be provided for each of the arguments. For example, the traditional *minus* operation defined as  $(int, int) \rightarrow int$  is increasing with respect to the first argument and decreasing with respect to the second. We denote this information using the notation  $incr(" ", arg1)$  and  $decr(" ", arg2)$ .

Finally, operations can be grouped into **clusters** which specify how to compute an argument of an operation given the result and other arguments, when it is possible. Therefore a cluster for an  $n$ -ary operation can have up to  $n + 1$  elements. Taking integer arithmetic as an example, one can create the following cluster, which describes relations between the addends and the sum:

```

Arguments: int A, int B, int C;
{ plus(A, B) = C;
  minus(C, A) = B;
  minus(C, B) = A; }

```

Operation properties are usually not described by ontologies. Agents assume properties of some (basic) types to perform operations on them. For example, the above *plus-minus* cluster is usually assumed for all numeric types. Making operation properties explicit allows the agent to reason about expressions on non-built-in types. For example, it is possible to define an operation of multiplication involving `Appointment` and `Float` operands. The agent does not need to understand that this operation means increasing the length of the appointment, but it may need to know how to compute (i.e. build an expression for) the length of the original appointment given the new length and the factor.

## 2.2 Service description

**Services** are viewed as collections of methods that operate on complex types. In this paper we consider only information gathering methods, i.e. those that *read* the state of the world but do not *update* it.

Each **method** has a name, a set of typed parameters, and a set of typed outputs. For example, the method `GetFlightInfo` below of a travel reservation service takes three parameters (two airport locations and a departure date), and returns exact departure time, flight number, and a flight duration. Each method can supply information about possibly multiple types. For instance, our example method can be used to

get both information about a flight and estimate travel time between two locations, corresponding to complex types `AbstractFlight` and `TransportationTime` respectively.

The **restriction** specifies what subset of instances of a complex type will be returned when the method is called with particular parameters. For example, for the `AbstractFlight` type only records with the given source and destination locations and departure time within 12 hours of the specified moment will be returned.<sup>1</sup> In general, the restriction of a method is a conjunction of inequalities involving parameters of the method, fields of the complex type, constants, and operations defined by the type system. Restrictions correspond to preconditions and conditions of conditional outputs of an OWL-S service.

The **assignment** section of the method specification describes how to obtain values of the fields of the complex type given values returned by the method call. When `GetFlightInfo` is used to obtain values of type `AbstractFlight`, outputs of the method are directly assigned to the fields of tuples. In case of the `TransportationTime` type, the constant time interval of three hours is added to the flight duration to account for check-in procedures.

```

QMETHOD GetFlightInfo {
  In:  Location P1, Location P2,
      DateTime P3;
  Out: DateTime O1, String O2,
      TimeInterval O3;
Relation AbstractFlight:
  Restr: from=P1, to=P2,
        departure>P3-12h, departure<P3+12h;
  Assign: departure=O1, flightno=O2,
        duration=O3;
Relation TransportationTime:
  Restr: from=P1, to=P2, means=flight;
  Assign: duration=O3+3h; }

```

Note that a method providing information about a particular type does not have to return all instances of the type (rows of the virtual table) or fill in all fields of the tuples (columns of the virtual table). The set of rows returned is described by the restriction section. The assignment section enumerates all provided fields.

In the rest of this paper, we will use a simplified version of this service description. We will assume that each service provides information about exactly one relation, and output assignments always assign method output directly to the tuple fields. Thus, the `Out` part of the method specification contains all fields bound by the method. These simplifications do not affect the algorithm we describe, but make our examples more readable. For example, the version of the above description for the `AbstractFlight` type is

```

QMETHOD GetFlightInfo {
  In:  Location P1, Location P2,
      DateTime P3;
  Out:  from, to, departure, duration,
      flightno;
  Restr: from=P1, to=P2, departure>P3-12h,
        departure<P3+12h; }

```

## 2.3 Queries

In the context of an information gathering agent, the goal of service integration is to obtain information specified by a

<sup>1</sup>We use infix notation for operations for brevity,  $P3 - 12h$  is actually  $minus(P3, 12h)$ .

query using calls to service methods and local processing. A **query** is defined as a complex type, a set of restrictions, and a set of fields of the complex type which need to be filled in.

For example, the query to get flight numbers for all flights between New York (NYC) and San Francisco (SFO) on March 21, 2004 (without duration information), can be specified as follows

```
Query Q = [AbstractFlight,
  (departure="21MAR2004", from="NYC",
  to="SFO"), (flightno)]
```

Such a query is similar to OWL-QL queries [7]. In OWL-QL, a query is described by a query pattern, a list of parameters to bind, and an optional answer pattern. The query pattern is a conjunction of OWL clauses essentially specifying the type of information searched for. For the above example, an OWL-QL query pattern would specify that the query is about `AbstractFlights` that have particular restrictions on the `departure`, `from`, and `to` properties, and also have some value for the `flightno` property. The bind list specifies what variables need to be returned. In our case, only values for the `flightno` property are of interest.

We chose the database-like representation of queries for the purpose of illustrating our algorithm. Furthermore, in examples in the rest of this paper, for simplicity, we will assume that all fields of the complex type are required, and will skip the name of the complex type when it is clear from the context.

## 2.4 Query plans

If the complex type is viewed as a relational table, both method calls and queries specify some regions of this table. A **query plan** combines the parameterized regions corresponding to method calls using local `SELECT`, `UNION`, and `JOIN` operations to approximate the shape of the query region. The `SELECT` operation filters out tuples returned by a method call but irrelevant to the query. Thus, `SELECT` operations (filters) decrease the set of rows in the plan region. The `UNION` operation simply merges sets of tuples, and therefore increases the set of rows in the plan region. The `JOIN` operation is an inner join with a condition that all fields existing in both parts are equal. The `JOIN` operation increases the set of columns in the plan region. Note that although it is desirable that the plan region have exactly the same set of rows as the query region (the same set of instances), extra columns (fields of the instances) do not have to be removed even if they are not required by the query.

We use the term **planner** to refer to the algorithm for finding a query plan for a given query. The planner presented in this paper takes a type system as an argument and does not assume any properties of operations except those explicitly specified. Although an algorithm that makes assumptions about some operation properties (e.g. linearity) can possibly achieve better performance, we believe that the flexibility offered by our approach is necessary in the Semantic Web world.

## 3 Examples

To illustrate our ideas, we will use two scenarios. In the rest of this paper, a superscript in formulas is used to specify the scenario being referred to. The first scenario (superscript A for *airline*) is used to demonstrate how multiple method calls can be combined with local processing to obtain complete information. The second scenario (superscript C for *calendar*) highlights computation of method parameters by

transforming restrictions. In the examples, `String`, `Number`, `DateTime`, `TimeInterval`, and `Location` are primitive types.

### 3.1 Airline scenario

In the first scenario, the goal is to collect information about available flights between two locations on a given date by interacting with services providing information about airlines and schedules. The query is defined in terms of a type (relation) `AirTicketInfo`, which has the following fields:

```
TYPEDEF AirTicketInfo {
  Location from, to;
  DateTime depart;
  String flightnum, airline;
  Number numseats; }
```

An instance of this type describes a flight, including the source and destination airport codes, the departure time, the flight number (two letters followed by four digits), the name of the airline, and the number of seats available.

Our earlier query about flights from New York to San Francisco can be concisely stated as:

$$Q^A = (\text{depart} = "21\text{MAR}2004.12 : 00", \\ \text{from} = "NYC", \text{to} = "SFO")$$

Let us assume that the service discoverer finds three service methods that can provide relevant information. The `GetSchedule` method returns flight numbers and departure and arrival times for flights between two locations on a given date carried by United and Continental, and takes the airline number as an argument. The discoverer generates two artificial methods, `GetSchedule1` and `GetSchedule2`, corresponding to the two airlines. Method `GetAirline` returns a name of an airline given a flight number. Finally, method `GetNumAvailableSeats` returns the number of available seats given the flight number and departure date.

```
QMETHOD GetSchedule1{//sim.,GetSchedule2
  In:   Location P1, Location P2,
        DateTime P3, String airline;
  Out:  from, to, depart, flight;
  Restr: from=P1, to=P2, depart>P3-"12h",
        depart<P3+"12h",airline="United"; }
QMETHOD GetAirline {
  In:   String P1;
  Out:  flight, airline;
  Restr: flight=P1; }
QMETHOD GetNumAvailableSeats {
  In:   String P1, DateTime P2;
  Out:  flight, depart, numseats;
  Restr: flight=P1; depart=P2; }
```

We would like the planner to produce the following plan:

1. In parallel
  - Call `GetSchedule1` with  $P1="NYC"$ ,  $P2="SFO"$ ,  $P3="21\text{MAR}2004"$ .
  - Call `GetSchedule2` with  $P1="NYC"$ ,  $P2="SFO"$ ,  $P3="21\text{MAR}2004"$ .
2. Combine results of the above method calls using the `UNION` operation. This produces a set of required tuples with four fields filled in.
3. To fill in the two missing fields, do the following operations in parallel and `JOIN` the newly obtained fields into the existing tuples
  - Call `GetAirline` for each of the tuples setting  $P1$  to be equal to the flight number.

- Call `GetNumAvailableSeats` for each of the tuples setting `P1` to be equal to the flight number and `P2` to the departure time.

This plan will extract as many required tuples as possible given available services. The planner should also state what part of information can be extracted. In this case, only flights by United and Continental are retrieved. If there is a Delta flight which satisfies the restrictions, it will not be found by this plan, because information about Delta flights is not accessible using the available services.

### 3.2 Calendar scenario

The second scenario models a situation where we want to find calendar appointments that start and end between two given time points by consulting a service that searches appointments by their beginning and duration.

The relation for calendar records is defined as follows:

```
TYPEDDEF CalendarEntry {
  DateTime    beginning;
  TimeInterval duration;
  String      label; }
```

The query we want to answer is

$$Q^C = (\textit{beginning} > C1, \textit{beginning} + \textit{duration} < C2)$$

where `C1` and `C2` are some constants of type `DateTime`, i.e. we are looking for all appointments starting after a given moment `C1` and ending before another moment `C2`.

Suppose, the discoverer finds only one method that returns information of the relevant type, but limits the duration of the appointment instead of its ending time:

```
QMETHOD CalendarM {
  In:    DateTime P1, TimeInterval P2;
  Out:   beginning, duration, label;
  Restr: beginning > P1, duration < P2; }
```

A possible plan to answer our query involves calling `CalendarM` with `P1=C1` and `P2=C2-C1`, and then taking only those of the returned tuples that satisfy  $(\textit{beginning} + \textit{duration} < C2)$ .

As these examples show, to create a plan our planner needs to be able to compute parameters of method calls and find combinations of method calls and local database operations that produce a complete (if possible) set of instances required by a query. In the following sections we first introduce our plan model and then describe an algorithm and demonstrate how it finds query plans in the above scenarios.

## 4 Plan Model

First, let us introduce some definitions. A **restriction** is a logical formula over an instance of a complex type. Restrictions in queries and service descriptions are conjunctions of (in)equalities. By **empty restriction** we denote the restriction equal to `TRUE`. The total set of instances of a complex type is denoted by  $\mathcal{U}$ . For a given restriction  $R$ ,  $SET(R)$  is the set of instances of  $\mathcal{U}$  that satisfies  $R$ . The following relations between restrictions and sets hold for any two restrictions  $R_1$  and  $R_2$ :

$$\begin{aligned} SET(TRUE) &= \mathcal{U} \\ (R_1 \rightarrow R_2) &\equiv (SET(R_1) \subseteq SET(R_2)) \\ SET(R_1 \wedge R_2) &= SET(R_1) \cap SET(R_2) \\ SET(R_1 \vee R_2) &= SET(R_1) \cup SET(R_2) \end{aligned}$$

In the rest of this paper,  $Q$  denotes a query restriction,  $SM$  a service method (e.g. a URL of the service description that might be used to invoke the service),  $SM(P)$  a restriction of a service method with a set of parameters  $P$  (i.e., the restriction formula from the corresponding service description),  $FIELDS(SM)$  and  $FIELDS(Q)$  are respectively the sets of fields filled-in by the method and required by the query.

A plan is a general step defined recursively using `UNION` and `JOIN` operations as described below. The terminal case of a general step is a basic step.

### 4.1 Basic step

The **basic step** of a query plan is a call to a service with possible filtering of the returned results. The filtering is realized by performing a `SELECT` operation on the returned tuples.

The general structure of the basic step is represented by the tuple  $BS = [SM, P, F, M, L]$ , where  $SM$  is the method to be called,  $P$  is the set of values for parameters of the method call,  $F$  is a condition for the `SELECT` operation (a logical formula),  $M$  is a mask restriction discussed below, and  $L$  is the list of fields filled-in by the method call.

It is possible, that no single method call provides all the information required by a query. The method call may return only a subset of columns of the virtual table (fields) and/or a subset of rows (instances). The two logical formulas,  $F$  and  $M$ , describe how the set of tuples returned by the method call differs from that required by the query. The **filtering condition**  $F$  removes the tuples returned by the method that do not belong to the query set  $SET(Q)$ . The **mask condition**  $M$  describes what part of the query tuples is contained in the method results. An empty mask ( $M = TRUE$ ) means that the query is answered completely. In general, the following holds:

$$SM(P) \wedge F = Q \wedge M \quad (1)$$

### 4.2 General step

A general step combines results from several steps.

A **general step** has a structural element  $V$  (e.g. an operation and a set of child steps the operation is applied to), a cumulative mask  $M$ , a set of fields  $L$  the step is guaranteed to produce, and a set of ordering constraints on the steps  $O$ :  $GS = [V, M, L, O]$ . The set of ordering constraints  $O$  is a set of pairs  $S_i \prec S_j$  stating that step  $S_i$  should complete before step  $S_j$  can be executed.

In the terminal case, the general step contains one basic step and empty ordering constraints. The mask and the set of fields are those of the basic step:  $GS = [BS, M, L, \{\}].$

A general step can also merge results of several child steps using a `UNION` or `JOIN` operation. In case of the `UNION` operation, the results of the child steps are merged, the mask restrictions are combined using the disjunction operation, and the set of produced fields is computed as an intersection of the sets of fields produced by the child steps:

$$\begin{aligned} GS &= [UNION(\{CS_i\}), M, L, O] \\ M &= \vee_i M_i \quad L = \cap_i L_i \quad O = \cup_i O_i \\ SET(GS) &= \cup_i SET(CS_i) \end{aligned}$$

The `JOIN` operation is performed when the child steps share some fields. This can happen either when the steps produce the same fields, or when fields produced by one step are used as parameters within the other step. The mask restrictions of child steps are combined using the `AND` operation, and the sets of produced fields are unioned. The set of

ordering constraints of the new general step includes a union of those of its child steps. Additional constraints are added if there is a data dependency between the child steps:

$$\begin{aligned} GS &= [JOIN(\{CS_i\}), M, L, O] \\ M &= \wedge_i M_i & L &= \cup_i L_i & O &\supseteq \cup_i O_i \\ SET(GS) &\subseteq \times_i SET(CS_i) \end{aligned}$$

### 4.3 Query plan

A **query plan** is just a general step which produces all of the fields required by a query. We require that a plan does not produce any irrelevant tuples (this is easily achieved using filtering conditions). A good plan also has an empty cumulative mask, i.e., it finds all tuples required by the query. Note that because of (1),  $Q \rightarrow M$  is equivalent to  $M = TRUE$ . We will discuss the notion of a cost of the plan in Section 7.

## 5 Planner Algorithm

The algorithm has two parts: the CreateCall procedure creates basic steps, and the CreateStep procedure aggregates steps (both basic and general) into general steps.

### 5.1 Creating a basic step

CreateCall receives a method  $SM$ , a query  $Q$ , and a set of fields known so far  $C$  and, if possible, produces a basic step  $[SM, P, F, M, L]$  or returns failure.

The known fields  $C$  can be considered constants, and can be included in parameter expressions. For example, the flight number field in the airline scenario of Section 3.1 is a constant for method calls to GetAirline and GetNumAvailableSeats.

We require that a method call fill in some new fields with respect to  $C$ . If  $L = C$ , CreateCall returns failure. Results of multiple steps producing the same set of fields (but different sets of tuples) may be UNIONED during creation of a general step.

The problem solved by CreateCall can be stated as follows: *find a set of parameters  $P$  such that  $SET(SM(P)) \cap SET(Q) \neq \emptyset$ .* Ideally, we also want to maximize the intersection and minimize the difference. Optimality, and even completeness, of an algorithm solving this problem depends on information about types and operations. The algorithm we present in this paper is provably terminating and correct, and we believe that in most real-life scenarios the parameters produced by this algorithm are close to optimum. Completeness of the algorithm, i.e. its ability to find a set of parameters  $P$  that produces a non-empty intersection of the plan and query sets if such a set of parameters exists, depends on the information provided by the type system.

We distinguish between four possible relations between the method set  $SET(SM(P))$  and the query set  $SET(Q)$ .

1. There exists a set of parameters  $P$  such that the method returns complete information:  $\exists P SET(Q) \subseteq SET(SM(P))$ .
2. There exists a set of parameters  $P$  such that all information returned by the method is relevant for the query:  $\exists P SET(SM(P)) \subseteq SET(Q)$ .
3. There exists a set of parameters  $P$  such that the method returns some relevant information:  $\exists P SET(SM(P)) \cap SET(Q) \neq \emptyset$ .
4. For any set of parameters  $P$ , the information provided by the service is completely irrelevant to the query:  $\forall P SET(SM(P)) \cap SET(Q) = \emptyset$ .

The last case corresponds to failure. The desirability of the other three cases decreases in the order shown.

In accordance with this classification, CreateCall calls three procedures: CheckComplete, CheckRelevant, and CheckIntersect. If one of these procedures succeeds, the obtained parameters are returned as a result of CreateCall. Otherwise CreateCall returns failure.

**Checking inclusion.** Procedures CheckComplete and CheckRelevant are dual, and we will describe them together.

Recall that the following relation holds between the method restrictions, the query, and mask and filter restrictions of a basic step:

$$SM(P) \wedge F = Q \wedge M$$

CheckComplete checks the situation when the query set is completely contained in the method set, i.e. the mask restriction is empty. This means that

$$Q \rightarrow SM(P) \quad (2)$$

$$SM(P) \wedge F \rightarrow Q \quad (3)$$

$$M = TRUE \quad (4)$$

In case of CheckRelevant, the filter is empty:

$$SM(P) \rightarrow Q \quad (5)$$

$$Q \wedge M \rightarrow SM(P) \quad (6)$$

$$F = TRUE \quad (7)$$

CheckComplete and CheckRelevant work as follows. First, the main implication (2 or 5) is checked, and suitable values of  $P$  are found. Then, a filter/mask condition is found such that the second implication holds (3 or 6). Note, that the latter implications have a trivial solution  $F = Q$  and  $M = SM(P)$ , and the algorithm just simplifies these expressions.

**Checking the implication.** The first part of the algorithm (the implication checking) works with three sets:

**database** is a set of known inequalities usually coming from the LHS of the implication we are trying to prove; logically this set represents a conjunction;

**toprove** is a set of sets of inequalities we are trying to prove; in each of the sets, all inequalities are equivalent in the sense that proving any one of them is sufficient to prove the others; initially, toprove is filled with single-element sets for inequalities from the RHS of the implication;

**assumptions** is an initially empty set of inequalities limiting the values of parameters.

The algorithm modifies these data structures (see below) until a contradiction is found or the toprove set becomes empty. In the latter case the assumptions set is used to compute the values of method parameters in a greedy way, e.g. translating the inequality  $p > a$  into a parameter value assignment  $p = a$ .

There are three types of modifications: expansion of the database set, matching of database and toprove sets, and modifications to the toprove and assumptions sets.

The first class of modifications adds new expressions into the database set by performing syntactic rewriting and simple reasoning based on the type system information. All expressions added as a result of these modifications logically follow from the database and the type system, and therefore do not affect the implications mentioned above.

- M1. Using **transitivity**:  $(a > b) \wedge (b > c) \rightarrow (a > c)$ . We assume that the order relation  $>$  is defined within a type, therefore in the above expression all variables are of the same type.
- M2. Using **monotonicity**: use monotonicity information about operations (when supplied) to replace an argument of an operation. For example,  $(a + b > c) \wedge (d > b) \wedge \text{incr}("+", \text{arg2}) \rightarrow (a + d > c)$ .
- M3. Applying **operation clusters**. Operation clusters are applied to an inequality to produce a set of equivalent inequalities. For example, with traditional arithmetic, applying the cluster from Section 2.1 to  $(a+b > c)$  produces additional inequalities  $(a > c - b)$  and  $(b > c - a)$ .

The only modification of the second class changes the `toprove` set using a syntactic match against the database set.

- M4. **Exact match**. If there is  $(Exp\ ROp\ Exp1)$  in database and  $(Exp\ ROp\ Exp2)$  in a set in `toprove`, and the number of fields used in  $Exp1$  is less than that in  $Exp$ , add  $(Exp1\ ROp\ Exp2)$  to the set in `toprove`.

Note, that the new inequality added to the set of the `toprove` set as a result of this modification is a sufficient (but not necessary) condition to prove other inequalities in the same set. The rationale behind this modification is that it might be easier to find a syntactic match for this new inequality than for the original ones.

Finally, modifications of the third class strengthen the assumptions so that the `toprove` set can be simplified, i.e. some of the expressions in `toprove` can be *proved*.

- M5. Remove **ready assumption**. If a set in `toprove` contains an inequality of the form  $(p > CONST)$  or  $(p < CONST)$ , where  $p$  is a parameter (unknown), the inequality is moved to `assumptions` (with consistency check) and the set is removed from `toprove`.
- M6. **Strengthen assumption**. In the `assumptions`, replace an inequality with equality. Ideally, this is the last thing we want to do, because we can get a false negative if it is done too early. For example, if we change  $x > a$  to  $x = a$ , we will get a false conflict if we see  $(x > b) \wedge (b > a)$  later.

The straightforward algorithm simply performs the modifications in any order until a solution or contradiction is found, or no further modifications are possible. The claim is that (i) any sequence of these operations terminate, (ii) if such a sequence produces an answer, the answer is correct. By correctness we mean that if the algorithm returns some values of  $P$  (as opposed to failure), then the following holds

$$AS(P) \rightarrow (DB(P) \rightarrow TP(P)) \quad (8)$$

$$\neg(AS(P) \equiv \text{false}) \quad (9)$$

where  $AS(P)$  is a conjunction of restrictions describing the set of parameters, and  $DB(P)$  and  $TP(P)$  are the two conjunctions describing the sets of instances in the database and `toprove` sets.

While the formal proofs are omitted for space reasons, the termination property follows from the fact that the database set can be extended only finitely many times. The total number of expressions that can be constructed using all values mentioned in the database gives the upper bound on the algorithm's complexity. The correctness property is proved by showing an invariant of all modifications which implies 8 and 9.

**Computing mask and filter.** After the implication checking part of the algorithm succeeds, we can find the filter/mask restrictions using the same ideas. Now all parameter values are fixed, so the inequalities contain only fields and constants. The following describes finding filter restrictions. The algorithm for finding mask restrictions is similar.

Recall that, by definition of filter,  $SM(P) \wedge F \rightarrow Q$ , and the pessimistic guess is  $F = Q$ . The idea is to simplify this original guess by checking all inequalities in  $Q$  and taking only those that cannot be proved from  $SM(P)$ :

$$F = \{e | e \in Q \wedge \neg(SM(P) \rightarrow e)\} \quad (10)$$

To do this, initialize the database with  $SM(P)$ . Apply transitivity, clusters, and monotonicity modifications in the database. For each of the inequalities  $q$  in  $Q$ : (i) apply clusters to  $q$ ; (ii) if there is no match with the database, include  $q$  in the filter.

**Example.** Consider the calendar scenario of Section 3.2. In this example, the method `CalendarM` provides complete information for the query  $Q^C$ . Here are the corresponding restrictions ( $b$  stands for *beginning* and  $d$  for *duration*):

$$Q^C : \quad b > C1, b + d < C2$$

$$\text{CalendarM}(P) : \quad b > P1, d < P2$$

Table 1 shows the initial values for the three sets.

database	toprove	assumptions
$b > C1$	$\{b > P1\}$	
$b + d < C2$	$\{d < P2\}$	

Table 1: Initial values for the three sets

The following modifications are performed to the sets:

1. Monotonicity of  $+$  with respect to the first argument and inequalities  $b+d < C2$  and  $b > C1$  from the database give  $C1 + d < C2$ .
2. Applying *plus-minus* cluster to the last inequality gives  $d < C2 - C1$  and  $C1 < C2 - d$ .
3. Two exact matches are performed producing  $C1 > P1$  and  $C1 - C2 < P2$  in the `toprove` set. Now `toprove` contains two sets of two inequalities each (see Table 2).
4. Two ready assumption modifications complete the process. At this moment the `toprove` set is empty, and the `assumptions` set contains  $C1 > P1$  and  $C2 - C1 < P2$ .

The values for parameters  $P1$  and  $P2$  can be computed from the assumption inequalities in a greedy way:  $P1 = C1$ ,  $P2 = C2 - C1$ .

database	toprove	assumptions
$b > C1$	$\{b > P1,$	
$b + d < C2$	$C1 > P1\}$	
$C1 + d < C2$	$\{d < P2,$	
$d < C2 - C1$	$C1 - C2 < P2\}$	
$C1 < C2 - d$		

Table 2: Values for the three sets after step 3

Now we can compute the filtering expression. After substituting the computed constant values for the method parameters, we obtain method restriction  $b > C1 \wedge d < C2 - C1$ . The first of the query inequalities,  $b > C1$ , follows from it trivially, and the second one,  $b + d < C2$ , cannot be proved. Therefore, the filter condition is  $b + d < C2$ .

**Checking intersection.** If neither (2) nor (5) hold, it is still possible that the service produces some relevant information, i.e. that the following holds:

$$AS(P) \rightarrow (SET(SM(P)) \rightarrow SET(Q) \neq \emptyset) \\ \neg(AS(P) \equiv false)$$

The pessimistic guess ( $F = Q$  and  $M = SM(P)$ ) is still valid. However, information available to the planner may not be sufficient to determine if the intersection is empty. In Section 7 we discuss possible ways to address this problem. For now, let us point out that the CreateStep procedure is capable of producing a correct (although possibly suboptimal) plan and quantify its incompleteness regardless of the algorithm used for checking intersection.

## 5.2 Creating a general step

The CreateStep procedure builds general steps of the plan by growing them from the beginning. The algorithm terminates either when a good plan is found (i.e. a general step that produces all necessary fields and has an empty mask), or no improvements are possible.

Input to the algorithm is a set of methods  $\{SM_i\}$  found by the discovery module and a query  $Q$ . The version of the algorithm presented below monotonically grows a set of general steps. An efficient implementation would drop steps when they become useless, i.e. cannot be used to produce new steps possibly leading to a plan.

1. Call CreateCall with  $C = \emptyset$  for each of the  $SM_i$ . Recall that  $C$  is the set of fields that can be considered constants by the CreateCall procedure. This produces the initial set of general steps  $\{[BS_j, M_j, L_j, \{\}]\}$ .
2. If there exists step  $GS_j = [V_j, M_j, L_j, O_j]$  such that  $M_j = TRUE$  and  $L_j \supseteq FIELDS(Q)$ , return  $GS_j$ .
3. Expand the set of general steps
  - (a) Create all possible UNION steps where there is a chance to improve the mask (the new mask is not equal to any of the masks of the child steps). For existing  $GS_i = [V_i, M_i, L_i, O_i]$  and  $GS_j = [V_j, M_j, L_j, O_j]$ , create  $[UNION(GS_i, GS_j), M_k, L_k, O_k]$ , where  $M_k = M_i \vee M_j$ ,  $L_k = L_i \cap L_j$ ,  $O_k = O_i \cup O_j$  if  $(M_k \neq M_i) \wedge (M_k \neq M_j)$ .
  - (b) For each existing complex step  $GS_i = [V_i, M_i, L_i, O_i]$  and method  $SM_j$  call CreateCall with  $C = L_i$ . If CreateCall succeeds and returns  $GS_j = [BS_j, M_j, L_j, \{\}]$ , create a new general step  $[JOIN(GS_i, BS_j), M_k, L_k, O_k]$ , where  $M_k = M_i \wedge M_j$ ,  $L_k = L_j$ ,  $O_k = O_i \cup \{GS_i \prec BS_j\}$ .
4. If the previous step added something, goto Step 2.
5. If there is a general step  $[V_i, M_i, L_i, O_i]$  with  $L_i \supseteq FIELDS(Q)$ , return such a step with the mask  $M_i$  closest to  $Q$ , because this is the best we can do given the services. Comparison of masks is done by checking implications between them.
6. Otherwise fail.

The above algorithm may create plan structures with repetitive parts and put unnecessarily tight ordering restrictions on the steps. Plans can be post-processed to improve their quality by applying rewriting rules [1]. For example,

$$UNION(JOIN(A, B), JOIN(A, C)) = \\ JOIN(A, UNION(B, C))$$

Since the JOIN operation is required to increase the set of known fields, the general steps constructed by this algorithm are DAGs. The maximum depth of such a DAG is limited by the number of fields the complex type has. Although plans cannot contain loops, loop-like behaviors can be simulated by creating virtual methods with different descriptions, as illustrated by the airline example.

## 5.3 Airline scenario revisited

For the airline example of Section 3.1, CreateStep constructs the following steps. The only two methods that can be called right away form the first two general steps:

$$GS_1^A = [BS_1^A = [GetSchedule1(...), TRUE], \\ (airline = United), \\ \{from, to, depart, flight\}, \{\}] \\ GS_2^A = [BS_2^A = [GetSchedule2(...), TRUE], \\ (airline = Continental), \\ \{from, to, depart, flight\}, \{\}]$$

Step 3a of the algorithm adds a new general step

$$GS_3^A = [UNION(GS_1^A, GS_2^A), \\ M^A = (airline = United \vee \\ airline = Continental), \\ \{from, to, depart, flight\}, \{\}]$$

Now, that the flight number field is known, new basic steps can be constructed (step 3b with  $C = \{from, to, depart, flight\}$ ). The following shows only the chain leading to the final plan.

$$GS_4^A = [JOIN(GS_3^A, \\ BS_3^A = [GetAirline(GS_1^A.flight), TRUE]), \\ M^A, \{from, to, depart, flight, airline\}, \\ \{GS_3^A \prec BS_3^A\}] \\ GS_5^A = [JOIN(GS_4^A, \\ BS_4^A = [GetNumAvailableSeats(GS_3^A.flight, \\ GS_3^A.depart), TRUE]), \\ M^A, \{from, to, depart, flight, airline, hasplace\}, \\ \{GS_3^A \prec BS_3^A, GS_4^A \prec BS_4^A\}]$$

Since it is not possible to further reduce the mask, the last step,  $GS_5^A$ , constitutes the plan.

## 6 Related Work

Description of the relationship between a service method and a query in terms of filter and mask formulas resembles the semantic caching technique, where a client data query to a database is answered by sending a *probe query* to a *semantic region* (a locally cached set of tuples described by a query) and a *remainder query* to the remote database, and then merging the results [6]. The problem of finding a service method that provides relevant information for a query is similar to that of finding a semantic region. The filter corresponds to the probe query, and the mask restriction is the reverse of the remainder query. The main difference is that the semantic caching technique performs no analysis of queries

except surface structure comparisons and simple arithmetic operations. Our algorithm performs more extensive analysis of query expressions and uses external specification of operations and properties.

Several information-gathering agents [8, 9] employ a representation of information sources as relations with binding patterns. Information Manifold [10] permits data sources to specify that they contain complete information. This specification can be used for query optimization, and allows a query planner to detect when a query may not be answered completely. However, we are not aware of any system that can quantify the incompleteness in query results.

Our algorithm for constructing a general step is similar to that used in Occam [9]. As in Occam, our algorithm grows plan prefixes, but in addition to chains of method calls it can also generate DAG structures using the UNION operation.

Reasoning in description logics is computationally hard [5]. [2] advocates a technique where descriptions of data sources are precompiled so that the global schema is represented in terms of local sources. This approach allows for very efficient answering of user queries stated in terms of the global schema in scenarios where many similar queries arrive between updates to the set of data sources. Complete compilation assumes a fixed set of services. For the application scenarios we are considering it is more likely that different queries require access to different data sources. Therefore, we choose to perform partial compilation. Service descriptions are represented in terms of the same relations, but matching of the restrictions is done for each query.

## 7 Summary and Future Work

In this paper we have presented a model for semantic service description that relies on an external specification of types and operations, and an algorithm for answering data queries that can use such services and quantify incompleteness of the answer.

The presented algorithm searches for a query plan, which maximizes completeness of information returned by the plan. A natural extension to the model is to support a notion of cost of a method call, for example, as a monetary price for use of the resource or as a total execution time. The algorithm can accommodate such an extension by selecting the cheapest plan in the general plan construction loop.

Another possible extension is to allow filtering of data not only directly after a method call, but also after UNION and JOIN operations. With plan cost information available, such a modification would provide more flexibility for construction of an optimum plan.

Additional filtering operations can be performed before method calls [14] to make sure that the methods are not called with invalid arguments. Generation of such filters can be based on method preconditions from OWL-S descriptions, and would not affect the rest of the algorithm.

As we pointed out in Section 5.1, information provided by the type system to the planner might be insufficient to determine if the intersection of the query set and the method set is empty. However, it might be the case that the method provides valuable information even if neither of the inclusion conditions (2, 5) holds. The two questions are how to compute the mask restriction and whether to use such a service at all. Additional information about types and operations (e.g. linearity) may make it possible to answer the question about intersection exactly. Given the model presented in this paper, a best-effort algorithm for computing the mask

can be constructed similar to that for checking inclusion, but using a conservative subset of modifications (only structural matches). Note, that this algorithm can detect some situations where the intersection is empty, but cannot guarantee that it is not. Using such a basic step in a plan may possibly increase the plan set, but also the cost of the plan. How to resolve the latter tradeoff and determine the kind of information needed for construction of an exact intersection checking algorithm are open research questions.

## References

- [1] J. L. Ambite and C. Knoblock. Flexible and scalable query planning in distributed and heterogeneous environments. In *AIPS*, pages 3–10, 1998.
- [2] J. L. Ambite, C. Knoblock, I. Muslea, and A. Philpot. Compiling source descriptions for efficient and flexible information integration. *Journal of Intelligent Information Systems*, 16(2):149–187, 2001.
- [3] A. Borgida, R. Brachman, D. McGuinness, and L. A. Resnick. CLASSIC: a structural data model for objects. In *ACM SIGMOD International Conference on Management of Data*, pages 58–67, 1989.
- [4] Business Process Execution Language for Web Services. <http://ibm.com/developerworks/library/ws-bpel/>, 2003.
- [5] D. Calvanese, G. De Giacomo, M. Lenzerini, and D. Nardi. Reasoning in expressive description logics. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1581–1634. 2001.
- [6] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.
- [7] R. Fikes, P. Hayes, and I. Horrocks. OWL-QL - a language for deductive query answering on the semantic web. Knowledge Systems Laboratory, Stanford University, Stanford, CA, 2003.
- [8] C. Knoblock, S. Minton, J. L. Ambite, N. Ashish, I. Muslea, A. Philpot, and S. Tejada. The Ariadne approach to web-based information integration. *International Journal of Cooperative Information Systems*, 10(1-2):145–169, 2001.
- [9] C. Kwok and D. Weld. Planning to gather information. In *AAAI*, pages 32–39, 1996.
- [10] A. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems*, 5(2), 1995.
- [11] D. McGuinness and F. van Harmelen. OWL Web Ontology Language overview. <http://www.w3.org/TR/owl-features/>, 2003.
- [12] OWL-S. <http://www.daml.org/services/>.
- [13] Semantic Web. <http://www.w3.org/2001/sw/>.
- [14] S. Thakkar, C. Knoblock, and J. L. Ambite. A View Integration Approach to Dynamic Composition of Web Services. In *Proc. of ICAPS'03 Workshop on Planning for Web Services*, 2003.
- [15] Universal Description, Discovery and Integration. <http://www.uddi.org>.
- [16] Web Service Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl>.