Discovering Semantic Web Services using SPARQL and Intelligent Agents

Marco Luca Sbodio\(^a\), David Martin\(^b,\,*\) Claude Moulin\(^c\)

\(^a\)Hewlett-Packard Italy Innovation Center, Corso Trapani 16, 10139 Torino, Italy
\(^b\)Artificial Intelligence Center, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, USA
\(^c\)Heudiasyc, JRU CNRS 7253, Université de Technologie de Compiègne, 60205 Compiègne, France

Abstract

This paper describes a novel approach to the description and discovery of Semantic Web services. We propose SPARQL as a formal language to describe the preconditions and postconditions of services, as well as the goals of agents. In addition, we show that SPARQL query evaluation can be used to check the truth of preconditions in a given context, construct the postconditions that will result from the execution of a service in a context, and determine whether a service execution with those results will satisfy the goal of an agent. We also show how certain optimizations of these tasks can be implemented in our framework.

Key words: Semantic Web services, Service discovery, SPARQL, Intelligent agents

1. Introduction

Service discovery – the identification of services that are capable of accomplishing a given objective – is a central problem in Semantic Web services (SWS) research. Most SWS work on discovery, either explicitly or implicitly, aims to support the autonomous identification of suitable services by software agents, to support the satisfaction of their goals. Effective service discovery depends directly on service descriptions that are adequately expressive. SWS service descriptions, in turn, usually include the specification of preconditions and postconditions. Preconditions are conditions that must hold true before invoking a service, to ensure successful use of the service, and postconditions are conditions that will hold true after the successful use of a service. The specification of preconditions and postconditions, and the drawing of inferences based upon them, is a distinguishing feature of most work on Semantic Web services.

In this paper, we show how the SPARQL [51] query language can be used to express the preconditions and postconditions of services, as well as the goals of agents. In addition, we show that SPARQL query evaluation can be used to check the truth of a precondition in a given context, construct the postcondition that will result from the execution of a service in a context, and determine whether a service execution with those results will satisfy the goal of an agent. In a nutshell, the truth of a service precondition indicates that the service can successfully be used, the resulting postcondition reveals what will be true after using the service, and the satisfiability of the agent’s goal indicates that the service is a candidate for use in accomplishing that goal – thus

\* Corresponding author. Tel: +1-650-859-4119, Fax: +1-650-859-3735

\Email addresses: marco.sbodio@hp.com (Marco Luca Sbodio), david.martin@sri.com (David Martin), claude.moulin@utc.fr (Claude Moulin).
providing a solution to the discovery problem. We also show how certain optimizations of these tasks can be implemented in our framework.

SPARQL has been standardized at the World Wide Web Consortium, and is by far the most widely used query language for knowledge bases employing the Resource Description Framework (RDF) [38] and/or the Web Ontology Language (OWL) [46]. Additional background on SPARQL is given in Section 3.

To situate our approach in a larger context of practice, we discuss how it may be used with OWL for Services (OWL-S) [44], but the approach is applicable to any SWS framework based on knowledge representation using RDF or OWL. OWL-S is deliberately under-constrained with respect to the specification and use of preconditions and postconditions. That is, it allows for a service description to “escape” into a language other than OWL for the specification of preconditions and postconditions. This path was chosen because OWL is not well-suited for expressing pre- and postconditions, both in terms of its expressiveness and its lack of naturalness for this purpose. For example, OWL’s lack of variables makes it difficult to express conditions with a suitable degree of generality, flexibility and naturalness. Thus, in the definition of OWL-S [43], several languages are declared as candidates for use in expressing pre- and postconditions, including, in addition to SPARQL, KIF [21], SWRL [29], and several other possibilities. (This collection of possibilities is meant by the authors of OWL-S to be illustrative, rather than exclusive.) In the examples accompanying the OWL-S documentation, and in subsequent work, SWRL has been used most widely – primarily because it was designed for use with OWL. SWRL, however, has its own drawbacks with respect to the expression of pre- and postconditions. These include expressiveness limitations (e.g., no disjunction), tractability, and lack of standardization. Other rules languages raise similar issues, and greater difficulties in terms of integration with OWL. Still more expressive languages, such as KIF, also raise issues of tractability and, in many cases, lack of standardization and tool support.

SPARQL provides a way out of these difficulties. SPARQL queries, as we shall show, allow for a natural, flexible, and expressive formulation of conditions and goals. In addition, since SPARQL is designed to be an integral part of the Semantic Web technology family, its use with RDF and OWL is already well understood and supported by many tools and environments, and its usage is in keeping with OWL-S’s objective to remain firmly situated in the world of Semantic Web standards. Additionally, it has been shown [1] that the expressive power of SPARQL is equivalent to that of non-recursive safe Datalog with negation, and hence to Relational Algebra.

OWL-S is focused on describing services and the processes that they encapsulate. Thus, agents have been left implicit in the world of OWL-S; that is, there are no ontology elements for agent, goal, or certain other central concepts from the world of agents. In many OWL-S research efforts, matchmaking techniques have been studied in isolation from the agents that might employ them. In most of these efforts, there has been more focus on classification (of services and service requests) as the basis of matchmaking, rather than on reasoning about preconditions, postconditions, and goals. We show here that SPARQL, in addition to specifying pre- and postconditions, is also well-suited to the specification of goals, thus filling a gap in the realm of OWL-S usage, and providing a single, standards-based framework that seamlessly handles the representation of these agent concepts along with OWL-S’s service concepts, and a mechanism for drawing inferences from them.

In the following section, we first provide a conceptual framework for our approach, in terms of the world states and transitions of modal dynamic logic. Section 3 gives an overview of SPARQL, and then shows how it can be used to characterize Web service operations and agents’ goals, with examples. In Section 4, we show, at a high level, how our approach can be used in a belief-desire-intention (BDI) agent framework. Section 5 spells out our SPARQL-based service discovery algorithm, with an example, and then goes on to discuss how it can be optimized for use with a remote registry. Section 5 also shows how SPARQL features can be used to support discovery with relaxation of preconditions and/or goals. Section 6 describes our prototype implementation and the experimental evaluation of our work. Section 7 discusses related work and Section 8 concludes.

1 With respect to tractability, SWRL is undecidable, but it should also be noted that a decidable language can be obtained by restricting SWRL to “DL-safe” rules, which would provide a more suitable candidate for expressing preconditions and postconditions.
1.1. A note on terminology

The structural elements that carry preconditions and postconditions vary across SWS and agent research. For example, in OWL-S it is the process that carries preconditions and effects (postconditions). In the Web Services Modeling Ontology (WSMO) [57] a service has a capability, and the capability has preconditions and postconditions. Actually, WSMO distinguishes two kinds of preconditions, termed preconditions and assumptions, and two kinds of postconditions, termed postconditions and effects, but these distinctions do not concern us here. In the Web Service Description Language (WSDL) [11], although preconditions and postconditions are not specified, conceptually it is the operation to which they could appropriately belong. Because WSDL and the related Semantic Annotations for WSDL and XML Schema (SAWSDL) [18] are standards and familiar to a large audience, in our conceptual framework we use the concept operation as the bearer of preconditions and postconditions.

2. Conceptual framework

We assume that a large number of Web services are published in a networked environment (i.e., the Internet, an intranet, or perhaps some kind of virtual organization), and their syntactic and semantic descriptions are held in registries (syntactic and semantic descriptions are not necessarily stored in the same registry). Each Web service has a syntactic description expressed in WSDL. Each Web service also has a formal semantic description, which defines for each operation:

- **semantics of input and output types**: input and output types are specified as OWL classes;
- **preconditions**: logical conditions that must hold before invoking the operation;
- **postconditions**: semantic description of the operation’s effects; that is, conditions that are guaranteed to hold true after a successful execution of the operation. We use the terms postconditions and effects interchangeably.

A software agent wants to accomplish some task or to achieve some goal, and therefore looks for a Web service that offers an appropriate operation to achieve its intentions. The software agent is situated in an environment; that is, it has a (possibly incomplete) description of the current state of the world. This world state description is given in the agent’s knowledge base as an RDF graph.

From an abstract point of view, a Web service operation can be seen as an action that the agent can invoke if and only if some conditions hold (the preconditions). The preconditions are evaluated against a state description (the current world state, as known by the agent’s knowledge base). The execution of the operation causes a state transition, and it has some effects or postconditions, which express what will be true in the world state resulting from the execution.

Let \( \Omega = \{\omega_0, \omega_1, \omega_2, \ldots\} \) be a countably infinite set of world states; let \( \Delta = \{\delta_0, \delta_1, \delta_2, \ldots\} \) be a countably infinite set of descriptions given as RDF graphs. The relation \( D = \Omega \times \Delta \) associates every world state with its corresponding RDF description, each of which expresses what is true in that world state. The relation \( D \) captures the intuition that the same world state \( \omega \) may be associated with multiple descriptions, each one representing a possible view on \( \omega \) from the perspective of a particular agent. The agent’s view on the world state \( \omega \) corresponds to the content of the agent’s knowledge base. Given the same world state \( \omega \), different agents may have different views on it, depending on their knowledge and perceptions. In general, one may define for each agent \( \alpha \) a function \( d_\alpha : \Omega \rightarrow \Delta \), such that for every world state \( \omega_i \in \Omega \), the result of \( d_\alpha(\omega_i) \) is the RDF graph describing the agent’s view on the world state \( \omega_i \) (i.e., the content of the agent’s knowledge base when the world state is \( \omega_i \)). In this paper we always refer to a single agent \( \alpha \), and therefore when we write \( \omega_i, \delta_i \) we mean that \( \delta_i = d_\alpha(\omega_i) \).

Let \( \Sigma = \{\sigma_0, \sigma_1, \sigma_2, \ldots\} \) be the set of Web service operations published in a registry. Note that in general two distinct operations \( \sigma_i, \sigma_j \) may belong to the same Web service or to different Web services, but this is not relevant here.

In abstract terms the world state transitions caused by an operation \( \sigma_i, \in \Sigma \) can be given in terms of a transition relation of a modal dynamic logic [39]. A modal dynamic logic allows one to reason about actions and their composition. For our purposes it is sufficient to consider atomic actions, which are isomorphic to Web service operations; the set \( \Sigma \) introduced above coincides with the set of atomic actions of our modal dynamic logic. Regular expres-

---

2 The description is assumed to be both internally consistent and accurate with respect to the state of the world. The matter of maintaining internal consistency – the belief revision problem – is discussed briefly below.
The above definition says that an operation \( \sigma_i \) causes a world transition from the world state \( \omega_j \) to the world state \( \omega_k \) if and only if \( \delta_j \) and \( \delta_k \) are RDF graphs describing respectively \( \omega_j \) and \( \omega_k \), the preconditions of \( \sigma_i \) are satisfied in \( \omega_j \), and \( \delta_k \) is the merge between the RDF graphs \( \delta_j \) and the RDF graph describing the effects of \( \sigma_i \) when invoked from the world state \( \omega_j \). When \( (\omega_j, \sigma_i, \omega_k) \in \mathcal{R} \) we use the graphical representation shown in Figure 1.

![World state transition](image1)

In general, given an initial world state \( (\omega_0, \delta_0) \), we have the situation shown in Figure 2.

![World state transitions tree](image2)

When an agent invokes a Web service operation, the RDF graph describing the effects of the operation augments the agent knowledge base. For example, referring to Figure 2, if the agent knowledge base is \( \delta_0 \), then the agent can invoke the operation \( \sigma_a \), which causes the effects described by \( f(\sigma_a, \delta_0) \). After invoking \( \sigma_a \), the agent knowledge base is the result of the merging operation \( g(\delta_0, f(\sigma_a, \delta_0)) \).

Depending on one’s perspective, the possible evolution of an agent’s world state can be modeled as a tree or a graph. One may argue that it is possible to reach the same world state \( (\omega_z, \delta_z) \) by executing a service \( \sigma_a \) from a world state \( (\omega_z, \delta_z) \), and by executing a service \( \sigma_b \neq \sigma_a \) from another world state \( (\omega_y, \delta_y) \). However, this would require that \( g(\delta_z, f(\sigma_a, \delta_z)) = g(\delta_y, f(\sigma_b, \delta_y)) \). Since \( \delta_z \neq \delta_y \), this is only possible when considering a very specific combination of effects of the two operations, and possibly inconsistencies arising when merging
such effects with the initial RDF graphs $\delta_x$ and $\delta_y$. Although it is theoretically possible to craft such a specific case, it seems to represent a contrived rather than a realistic situation. Moreover, it is straightforward to ensure the uniqueness of every world state simply by recording the history of operations whose execution has led to that state (and reasonable to assume that an agent would be interested in such a record). Consequently, we prefer to describe the evolution of the agent’s world state (and associated knowledge base) as a tree rather than a graph. We observe also that our choice of the tree representation does not cause any loss of generality, because the approach and the algorithms described in the following sections do not rely on this particular choice.

The following sections describe how SPARQL can be used to define agent goals, and the preconditions and postconditions of Web service operations according to the conceptual framework described above. We also show how this approach allows for an implementation of intelligent agents that can identify appropriate Web service operations to achieve their goals. In this approach, we will employ SPARQL CONSTRUCT queries to check the truth of preconditions and create RDF graphs representing the effects of available operations, and ASK queries to determine whether those effects satisfy the goals of agents. In the following section, we provide an overview of the SPARQL features that make this approach possible.

### 3. SPARQL as an expression language

Semantic Web service research has not yet led to a consensus recommendation on the expression language for defining such conditions. For example, OWL-S allows several kind of expression languages, whereas WSMO specifies the use of the Web Services Modeling Language (WSML) [8], and SAWSDL [18] does not explicitly deal with preconditions and postconditions, nor make any recommendations about language(s) to be used for semantic specifications. We present here an approach based on SPARQL. We first give an overview of SPARQL, and then we show how SPARQL features allow for a compact and effective representation of preconditions and postconditions of Web service operations, and of agents’ goals. We describe also how the usage of SPARQL for preconditions, postconditions, and goals fits with the conceptual framework presented in Section 2.

#### 3.1. SPARQL features

SPARQL is the query language for RDF that has been standardized by W3C [51]. SPARQL allows to query RDF data sources as directed labelled graphs. It has capabilities for querying required and optional graph patterns and their conjunctions and/or disjunctions; it also supports value testing and filtering of results. SPARQL queries can yield either result sets or RDF graphs. This section provides a brief overview of SPARQL features, syntax and semantics. The reference document for SPARQL is [51]; an in depth analysis of SPARQL syntax and semantics is available in [49] and [50].

The syntax of SPARQL is based on the following pairwise disjoint infinite sets of symbols:
- $T$: the set of Internationalized Resource Identifiers (IRIs), as defined in RFC3987 [16];
- $B$: the set of blank nodes in RDF graphs;
- $L$: the set of RDF literals;
- $V$: the set of variables.

An RDF term is a member of the set $T = I \cup B \cup L$. A triple pattern is a member of the set $P = (T \cup V) \times (T \cup V) \times (T \cup V)$; a triple is a tuple $(s, p, o)$, where $s, p, o$ are respectively the subject, predicate and object. SPARQL queries are built using graph patterns, which, following the inductive definition found in [49], can be defined as:

$$
\Gamma ::= \{ p_0, p_1, \ldots, p_n \} | \\
\{ \gamma_0, \gamma_1 \ldots, \gamma_n \} | \\
\gamma_0 \text{OPTIONAL} \gamma_1 | \\
\gamma_0 \text{UNION} \gamma_1 \ldots, \gamma_n | \\
\gamma_0 \text{FILTER} r
$$

where $p_0, \ldots, p_n \in P$, $\gamma_0, \ldots, \gamma_n \in \Gamma$, and $r$ is an expression that eliminates those solutions that, when substituted into $r$, either result in an effective boolean value of false or produce an error. The definition (1) describes the following types of graph patterns:

- **Basic Graph Pattern** $\{ p_0, p_1, \ldots, p_n \}$, is a set of triple patterns, where each query result must match all triple patterns;
- **Group Graph Pattern** $\{ \gamma_0, \gamma_1 \ldots, \gamma_n \}$, is a set of graph patterns, where each query result must match all graph patterns (the group graph pattern is equivalent to the conjunction of its graph patterns); an empty Group Graph Pattern is called **Empty Graph Pattern** (it always matches with one solution that does not bind any variable);
**Optional Graph Pattern** \( \gamma_0 \text{ OPTIONAL } \gamma_1 \), is made of a pair of graph patterns, where the second pattern extends the results of the first, but does not cause the overall graph pattern to fail;

**Alternative Graph Pattern** the graph pattern \( \gamma_0 \text{ UNION } \gamma_1 \text{ UNION } \ldots \gamma_n \), is the disjunction of two (or more) graph patterns, where any graph pattern can match.

A filter expression \( r \) can be applied to any graph pattern, and it restricts the set of solutions; \( r \) can use a broad set of operators, including arithmetic, logical operators, regular expression matching on strings, etc. (the complete list of available operators can be found in [51]). Queries may also use modifiers, which allows for modifications of the solution sequence (for example ordering the solutions, or avoiding repetitions, or limiting the number of solutions).

SPARQL queries refer to an RDF Dataset, which is defined as

\[
DS = \{ \delta, (u_1, \delta_1), (u_2, \delta_2), \ldots (u_n, \delta_n) \}
\]

where \( \delta, \delta_1, \ldots \delta_n \) are RDF graphs, and \( u_1, \ldots, u_n \) are IRIs; \( \delta \) is called the *default graph*, and the tuples \( (u_i, \delta_i) \) are called *named graphs*.

SPARQL has four *query forms*, which use the solutions from the graph pattern matching to build result sets or RDF graphs:

- **SELECT** query form returns all, or a subset of, the variable bindings that result from each match of the query graph pattern;
- **CONSTRUCT** query form returns an RDF graph specified by a graph template; the resulting RDF graph is formed by taking each query solution, substituting for the variables into the graph template, and combining the triples into a single RDF graph by set union;
- **ASK** query form returns a boolean that indicates whether the query graph pattern matches or not;
- **DESCRIBE** query form returns an RDF graph that describes the resources found; the result RDF graph is not constrained by a template as in the **CONSTRUCT** query form, but it is determined by the SPARQL query processor.

A SPARQL query is formally defined in [51] as a tuple \( (E, DS, R) \), where \( E \) is a SPARQL algebra expression, \( DS \) is an RDF Dataset, and \( R \) is a query form. \( E \) is an expression in SPARQL relational algebra, which is an abstract intermediate language for the definition and analysis of queries, and it is used in the query planning and optimization phase. An early description of a relational algebra for SPARQL can be found in [12]. The W3C reference document [51] describes how to convert graph patterns into SPARQL algebra expressions. For our purposes it is not necessary to consider the translation of graph patterns into relational algebra expressions, rather it is more meaningful to use the graph patterns, and therefore we define a SPARQL query as:

\[
\text{SPARQL query} ::= (\gamma, DS, R) \quad (2)
\]

where \( \gamma \) is a SPARQL graph pattern as defined in (1), \( DS \) is an RDF Dataset, and \( R \) is one of the four SPARQL query forms. In the following sections we use **CONSTRUCT** and **ASK** query forms. The **CONSTRUCT** query form is characterized not only by its graph pattern \( \gamma \), which is used to find query solutions, but also by its template pattern \( \tau \), which is essentially a basic graph pattern (i.e. a set of triple patterns) that is used to build the result RDF graph by substituting variables in \( \tau \) with query solutions.

For the sake of clarity we make the template pattern explicit in the query notation, and so we write **CONSTRUCT** queries as \((\gamma, DS, \text{CONSTRUCT } \tau)\).

### 3.2. Web service operations

We characterize a Web service operation \( \sigma_i \in \Sigma \) with a pair of SPARQL graph patterns \((\gamma_i, \tau_i)\), where \( \gamma_i \) defines the preconditions of \( \sigma_i \), and \( \tau_i \) is used as a template to build the RDF graph defining the effects of \( \sigma_i \).

The graph pattern \( \gamma_i \) may contain variables that either refer to the inputs or outputs of \( \sigma_i \), or to additional parameters that are required to express the preconditions. The template pattern \( \tau_i \) may contain not only grounded triples, but also triple patterns including variables. The variables in \( \tau_i \) refer either to the inputs or outputs of the Web service operation, or to the additional parameters that are also used in \( \gamma_i \). The template pattern \( \tau_i \) allows for the sharing of variables between the precondition and effects, and it constrains the creation of the RDF graph describing the effect of \( \sigma_i \).

We give a concrete implementation of functions \( f \) and \( h \) introduced in Section 2 using the SPARQL query forms described in Section 3.1. Given a world state \( \omega_j \in \Omega \) such that \( (\omega_j, \delta_j) \in D \), and given a Web service operation \( \sigma_i \in \Sigma \) such that \( \sigma_i = (\gamma_i, \tau_i) \), we define:

\[
f(\sigma_i, \delta_j) ::= (\gamma_i, \{ \delta_j \}, \text{CONSTRUCT } \tau_i) \quad (3)
\]

\[
h(\sigma_i, \delta_j) ::= (\gamma_i, \{ \delta_j \}, \text{ASK}) \quad (4)
\]
Given the world state \( \omega_j \), whose description is given by the RDF graph \( \delta_j \), checking the satisfiability of the preconditions of \( \sigma_j \) in \( \omega_j \), is equivalent to verifying that the SPARQL graph pattern \( \gamma_i \) has a non-empty set of solutions on the RDF Dataset whose default graph is \( \delta_i \). We use a SPARQL ASK query form to verify such a condition.

The CONSTRUCT query form allows for the programmatic creation of the RDF graph that expresses the effects of \( \sigma \); the construction of this graph is constrained by the template pattern \( \tau_i \). Note that if the graph pattern \( \gamma_i \) has no solution over the Dataset \( \delta_j \), then the CONSTRUCT query yields an empty graph pattern. This represents the case where the preconditions of the operation \( \sigma \) do not hold in the world state \( \omega_j \), and therefore the operation cannot be performed.

We observe that the RDF graph \( f(\sigma_i, \delta_j) \) resulting from the execution of the CONSTRUCT query is to be merged with the RDF graph \( \delta_j \) representing the description of the initial world by using the function \( g \) described in section 2. The RDF triples of \( f(\sigma_i, \delta_j) \) can be regarded as the add list used in STRIPS [20] to describe the positive effects of an operator. STRIPS also defines a delete list, which gives the negative effects of an operator, and uses these lists to cope with the notorious frame problem [45,26]. STRIPS assumes that the world does not change much from one instant to the next: if an action is executed, then the world description will be fully updated by adding the facts in the add list of the action, and by removing the facts identified by its delete list.

We handled essentially the same approach: when executing a Web service, the only changes in the description \( \delta_j \) of the initial world are those given by the RDF graph generated by the CONSTRUCT query, which is then merged with \( \delta_j \), preserving consistency as outlined in section 2. Compared to STRIPS, our definition of a Web service \( \sigma_i = (\gamma_i, \tau_i) \) lacks the specification of negative effects, and this is a limitation when addressing the frame problem. However, we can easily extend our definition as follows: \( \sigma_i = (\gamma_i, \tau_i^+, \tau_i^-) \), where the SPARQL graph pattern \( \gamma_i \) defines the preconditions of \( \sigma_i \), and the SPARQL template patterns \( \tau_i^+ \) and \( \tau_i^- \) define respectively the positive and negative effects of the Web service. In terms of our computational model, this would require two separate CONSTRUCT queries to build the RDF graphs corresponding to the template patterns \( \tau_i^+ \) and \( \tau_i^- \), and a merging operation that adds/subtracts them to/from the description \( \delta_j \) of the initial world. Incidentally, we observe that a recent W3C Member Submission [62] has proposed SPARQL/Update, which is a companion language to SPARQL enabling updates to an RDF graph. More precisely, SPARQL/Update provides the following facilities: insert new triples to an RDF graph, delete triples from an RDF graph, and perform a group of update operations as a single action. The proposed SPARQL/Update provides a graph update operation named MODIFY, whose outline syntax is as follows:

```
MODIFY [ <uri> ] • DELETE template
  INSERT template
[ WHERE pattern ]
```

where the optional \( \text{uri} \) identifies a named graph (if omitted, the operation works on the default graph), \( \text{template} \) is a SPARQL template pattern, and the optional \( \text{pattern} \) is a SPARQL graph pattern. Intuitively, the MODIFY operation works as follows:

- if \( \text{pattern} \) is empty (or missing), then the DELETE and INSERT template should contain only ground triples (without variables), and the DELETE operation is performed before the INSERT;
- if \( \text{pattern} \) is non-empty, and it has solutions over the specified RDF graph, then such solutions are used to instantiate the DELETE and INSERT templates, and then the DELETE operation is performed before the INSERT;
- if \( \text{pattern} \) is non-empty, and it has no solutions over the specified RDF graph, then no operation is performed.

Such a MODIFY operation suggests a straightforward computational model for our extended definition of a Web service as \( \sigma_i = (\gamma_i, \tau_i^+, \tau_i^-) \), and allows us to better address the frame problem. However, since SPARQL/Update is not yet consolidated, and the proposed MODIFY operation is potentially subject to changes in its syntax and semantics, we restrict ourselves to the original definition of a Web service as \( \sigma_i = (\gamma_i, \tau_i) \), and to the use of the CONSTRUCT query form as explained above. Such a restriction does not diminish the generality of the approach presented in this paper.

3.3. Operation example

As an example we consider the Express Congo Buy Bookselling Service \(^3\), which essentially allows regis-

\(^3\) The Express Congo Buy Bookselling Service was originally developed as an OWL-S exam-
tered users to buy books on-line. The corresponding Web service implements several operations, among them the buy operation, whose successful execution can be described as follows:

- the operation takes as inputs the identifier of the required book, the user credentials, and credit card information;
- the operation outputs an acknowledgement of the transaction;
- the operation’s precondition requires that (i) the identifier of the required book is an International Standard Book Number (ISBN), (ii) the user credentials specify a Congo Service Account, and (iii) the given credit card information (card number, type, and expiration date) is for a valid account;
- the operation’s effect specifies (i) the shipment of the required book to the address specified in the user’s Congo Service Account, and (ii) the sending of a shipping acknowledgment.

Adopting the notation introduced in the previous sections, we say that the buy operation of the Express Congo Buy Bookselling Service is defined as $\sigma_b = (\gamma_b, \tau_b)$, where the subscript $\tau$ stands for “buy”. Figure 3 shows its representation as a SPARQL CONSTRUCT query. (We omit prefix declarations in all figures for the sake of brevity. In addition, in some places we use the N3 notation ‘a’, as in $\text{congo:AcctID} = \text{?acctID}$, instead of using rdf:type.) Figure 3 shows how the SPARQL query graph pattern $\gamma_b$ encodes the precondition: variables correspond to the inputs (?bookISBN, ?signInInfo, and ?creditCard), and additional required parameters related to the inputs. The graph pattern includes also type checking of input types (profileHierarchy:ISBN, congo:SignInData, and congo:CreditCard), which is useful (in OWL-S preconditions do not explicitly check input types, but these type checks are frequently regarded as implicit preconditions) The precondition uses a FILTER to ensure syntactic constraints on the credit card number and expiration date (note that more sophisticated constraints can be added; for example a regular expression could control the pattern of the credit card number).

To enable the evaluation of references to inputs in preconditions, we adopt a convention that inputs

\begin{verbatim}
CONSTRUCT {
_:book a profileHierarchy:InStockBook ;
_:output a congo:orderShippedAck .
_:shipment a congo:Shipment ;
congo:shippedBook _:book .
congo:shippedTo ?acctID ;
}
WHERE {
?signInInfo a congo:SignInData ;
congo:hasAcctID ?acctID .
?acctID a congo:AcctID .
?creditCard a congo:CreditCard ;
congo:cardExpiration ?expirationDate ;
congo:cardType [ a congo:CreditCardType ] ;
congo:cardNumber ?creditCardNumber .

FILTER ( dataType(?creditCardNumber) = xsd:string &&
dataType(?expirationDate) = xsd:gYearMonth )
}
\end{verbatim}

Fig. 3. Buy operation $\sigma_b = (\gamma_b, \tau_b)$ of Express Congo Buy Bookselling Service

will be asserted in the agent’s knowledge base prior to the evaluation of the query.

The template pattern $\tau_b$ yields an RDF graph whose structure is shown in Figure 4.

The template pattern $\gamma_b$ encodes outputs and additional parameters as blank nodes to ensure generality, and it uses variables from the query graph pattern for tying nodes of the resulting RDF graph with nodes of the input RDF graph, so that input variable bindings will be propagated to output variables. The binding of variables happens when the CONSTRUCT query is executed on an input RDF graph $\delta_i$ satisfying the constraints of query graph pattern $\gamma_b$. Such an execution corresponds to the computation of the function $f(\sigma_b, \delta_i)$.
3.4. Agent’s goal

In terms of our conceptual model, the goal of a software agent is often conceptualized as a state of affairs that the agent wants to bring about in the world, and this is the concept adopted here. A goal can be formally characterized by a condition that expresses what the agent wants to be true; this condition is then treated by the agent as the desired effects of its actions. Checking whether a goal is achieved or not corresponds to verifying whether the description $\delta_k$ of some world state $\omega_k \in \Omega$ contains these desired effects; that is, the condition evaluates to true in $\delta_k$.

Since $\delta_k$ is an RDF graph, we can easily perform the above check by querying a Dataset whose default graph is $\delta_k$ using a graph pattern $\gamma_g$ that expresses the desired effects (i.e. the agent goal).

According to the conceptual framework introduced in section 2, and using the notation described in Section 3.1, we can write:

$$\text{goal } g \text{ is achievable } \iff \exists \omega_k \in \Omega : (\omega_k, \delta_k) \in D \land (\gamma_g, \delta_k, \text{ASK}) = \text{true}$$

(5)

Intuitively, it is possible to achieve the goal $g$ if and only if there exists a world state $\omega_k$ such that $\delta_k$ is the description of $\omega_k$, and the SPARQL ASK query whose graph pattern is $\gamma_g$, and whose RDF Dataset has $\delta_k$ as default graph, returns true.

We therefore represent a goal $g$ with a SPARQL graph pattern $\gamma_g$, and we use the SPARQL ASK query form to check for achievement of $g$; that is, we check whether or not $\gamma_g$ has a solution over some $\delta_k$. If such a solution exists, then the goal $g$ is achieved in the corresponding world state $\omega_k$, otherwise it is not. The SPARQL ASK query representing an agent goal is executed over the RDF graph obtained by merging the initial world description (i.e. the content of the agent knowledge base) and the RDF graph describing the effects of a Web service operation (that is the result of function $f$ defined in (3)). A detailed description of how an agent checks for achievement of its goal is given in Section 5.

3.5. Goal example

Let us assume that Marco has an intelligent agent that performs operations on his behalf. The agent wants to achieve a world state where an in-stock-book, whose ISBN is the one that Marco is looking for, is shipped to Marco’s account. The agent’s goal is represented by the SPARQL ASK query shown in Figure 5.

$$\text{Fig. 5. Agent goal}$$

Marco’s agent has a knowledge base, which gives information on a number of things. Such a knowledge base is encoded in RDF, and figure 6 shows a fragment of it.

$$\text{Fig. 6. Fragment of the agent knowledge base}$$

The agent knowledge base contains the description of Marco (a foaf:Person). The agent knows Marco’s accounts on various services, among which the Express Congo Buy Bookselling Service (akb:myAccountId). It knows also the details of Marco’s credit cards (akb:myVisa). Additionally, the agent has its own internal representation of the fact that Marco is looking for a particular book, which is identified by its ISBN (akb:wantedISBN). In section 5.1 we describe how Marco’s agent can fulfill its goal.

4. Intelligent agents

An intelligent agent can be defined according to [72] as a computer system situated in some environ-
ment, and able to perform autonomous actions in order to achieve its design objectives. An intelligent agent is also characterized by some degree of reactivity (ability to respond to changes in the environment), proactiveness (goal-directed behaviour), and social ability (capability of interacting with other agents).

There are several concrete architectures for intelligent agents. Our software agent is close to a belief-desire-intention agent (BDI agent), in which decision making depends on the manipulation of data structures that represent the beliefs, desires and intentions of the agent. The BDI theory was originally developed in [6] and [7], and it is a theory of practical reasoning consisting of two main activities: deciding what state of affairs the agent wants to achieve (deliberation), and deciding how to achieve that state of affairs (means-ends reasoning). A BDI agent is characterized by three sets: Beliefs, Desires and Intentions. The set of beliefs is a (usually incomplete) knowledge base containing the information that the agent has about the world. Desires represent the state of affairs that agent would wish to bring about, and the intentions are desires that the agent has committed to achieve.

Traditional BDI agents have a plan library (or task library in some BDI frameworks), which contains a set of procedural plans for various situations in which an agent may find itself. Situations in which plans must be executed are usually determined by a triggering event and a context: the former identifies the moment when the plan may be necessary, whereas the latter specifies the preconditions for the plan to be applicable. Plans may be composite (structured) or primitive. Composite plans bottom out in primitive plans, which are indivisible actions (from the agent’s perspective), and are typically implemented as hard-coded procedures.

One implementation of a BDI agent architecture is the Procedural Reasoning System (PRS) [22]; there are various works dealing with the formalization of the BDI model, among which are [53], [54] and [55]. There are also some agent-oriented languages, such as AgentSpeak(L), which has been introduced in [52], and which has also an interpreter implementation named Jason (see [4] and [5]).

The functioning of our agent is inspired by that of the AgentSpeak(L) interpreter as described in [52] [14]. Figure 7, which is adapted from [41], gives a visual representation of the architecture. In Figure 7 rectangles represent sets (of beliefs, events, plans, and intentions), circles represents agent’s processing steps, and diamonds represent selection functions (which extract one element from an input set). An AgentSpeak(L) agent processes perceptions of the world through a Belief Revision Function (BRF). On the basis of the current percept and the current beliefs, the BRF is used to update both the Events set and the Belief Base itself. Events are filtered by a selection function \( S_E \), which extracts a single event from the agent’s Events set; events may be the addition (or deletion) of a belief, or the addition (or deletion) of a goal, which represents a desire. The processing step \( U_E \) unifies the selected event with the triggering events of the agent’s plans (Plan Library) identifying a set of relevant plans, i.e. those plans that are triggerable by the current event. The processing step \( U_C \) checks for every relevant plan whether its context unifies with the set of current beliefs (Belief Base), thus determining the set of applicable plans, i.e. those plans that are actually usable to handle the selected event. The function \( S_I \) selects one applicable plan, which becomes an intention of the agent. Finally, at every execution cycle, the function \( S_O \) selects one intention and executes it. If the selected intention is a primitive plan, its execution \( (E_A) \) yields an action that may affect the state of the world, or may generate new internal events.

Similarly to an AgentSpeak(L) agent, our agent has a Belief Base (or knowledge base), which is described as an RDF graph. The perception of the world, and the Belief Revision Function are out of scope of this paper. The relevant events for our agent (i.e. those selected by \( S_E \)) are those representing the addition of a goal, which is defined by a SPARQL ASK query as described in Section 3.4.

Where the AgentSpeak(L) agent has primitive plans (indivisible actions), our agent has access to
a set $\Sigma$ of Web service operations, as introduced in Section 2. The descriptions of these operations are stored in a registry, which could either be local (internal to the agent) or remote. When a new goal $g$ is given to the agent, it has to discover an appropriate Web service operation $\sigma \in \Sigma$ to achieve $g$. The discovery operation performed by our agent corresponds to the processing steps $U_E$ and $U_C$ shown in Figure 7. More precisely:

- in an AgentSpeak(L) agent the step $U_E$ unifies the selected event with the triggering event of the agent’s plans; in our agent the step $U_E$ checks whether the agent goal $g$ is achievable (see definition (5)) in some world state (i.e., it includes an ABox) is essential for our approach, to allow for checking the truth of services’ preconditions in the current world state resulting from the execution of the service. The availability of such an ABox is essential for our approach, to allow for checking the truth of services’ preconditions in the current world state (i.e., it includes an ABox in description logic terminology). The availability of such an ABox returns true, then the goal is achieved in the world state resulting from the execution of the service.

- in an AgentSpeak(L) agent the step $U_C$ checks for every relevant plan (those identified by $U_E$) whether its context unifies with the current Belief Base of the agent; in our agent the step $U_C$ checks whether the preconditions of the selected Web service operation $\sigma$ are satisfied by the current agent knowledge base ($U_C$ is the Unification of Conditions).

The algorithm implementing the discovery process for our agent is presented in Section 5. The execution of the discovery algorithm yields a set $\Sigma_g \subset \Sigma$, which contains the Web service operations whose effects allow achievement of the agent goal $g$, and whose preconditions are satisfied by the current agent knowledge base. The execution step for our agent consists in selecting a Web service operation $\sigma_g \in \Sigma_g$ and invoking it to achieve the goal $g$. The description of how to select $\sigma_g$ and of how to invoke it, is out of scope of this paper.

5. Agent’s discovery algorithm

Given the set of world states $\Omega = \{\omega_0, \omega_1, \omega_2, \ldots\}$ and the set of RDF graphs $\Delta = \{\delta_0, \delta_1, \delta_2, \ldots\}$ introduced in Section 2, we identify with $(\omega, \delta_u)$ the current world state for a given agent, and the corresponding RDF description. We assume that the agent knowledge base contains the RDF graph $\delta_u$, and thus includes instance data descriptive of the current world state (i.e., it includes an ABox in description logic terminology). The availability of such an ABox is essential for our approach, to allow for checking the truth of services’ preconditions in the current world state. The agent goal $g$ is defined by a SPARQL graph pattern $\gamma_g$ as discussed in Section 3.4. The discovery process of our agent is presented in Algorithm 1.

**Algorithm 1** The agent’s discovery algorithm

**Input:**
- $\delta_u$: the agent knowledge base
- $\gamma_g$: the graph pattern describing the agent goal $g$
- $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_n\}$: the set of Web service operations; for each $\sigma_i$, $\gamma_i$ is the graph pattern and $\tau_i$ the template pattern used in defining $\sigma_i$ (see Section 3.2)

**Output:**
- $\Sigma_g$: the set of selected Web service operations

1: $\Sigma_g \leftarrow \{\}$
2: for all $\sigma_i \in \Sigma$ do
3: $\delta_e \leftarrow (\gamma_i, \{\delta\}, \text{CONSTRUCT } \tau_i)$
4: if $\delta_e$ is not empty then
5: if $(\gamma_g, \{g(\delta_a, \delta_c)\}, \text{ASK}) == \text{true}$ then
6: $\Sigma_g \leftarrow \Sigma_g \cup \{\sigma_i\}$
7: end if
8: end if
9: end for
10: return $\Sigma_g$

Algorithm 1 performs both the Unification of Conditions $U_C$ and the Unification of Effects $U_E$ described in Section 4:

- $U_C$ corresponds to the statement

$$\delta_e \leftarrow (\gamma_i, \{\delta\}, \text{CONSTRUCT } \tau_i)$$

(see line 3), in which the SPARQL CONSTRUCT query representing the Web service operation $\sigma_i$ is executed over the agent knowledge base $\delta_u$; if the graph pattern $\gamma_i$ has solutions over $\delta_u$, then the preconditions of $\sigma_i$ are satisfied, and the CONSTRUCT query yields the RDF graph $\delta_e$ describing the effects of $\sigma_i$;

- $U_E$ corresponds to the check

$$(\gamma_g, \{g(\delta_a, \delta_e)\}, \text{ASK}) == \text{true}$$

(see line 5), in which the SPARQL ASK query representing the agent goal is executed over the RDF graph obtained by merging (function $g$) the initial agent’s knowledge base ($\delta_u$) and the RDF graph describing the effects ($\delta_e$). If the ASK query returns true, then the goal is achieved in the world state resulting from the execution of the service.

Algorithm 1 performs the steps $U_C$ and $U_E$ in the opposite order with respect to what is shown in Fig-
Optimizations: Algorithm 1 assumes that the agent has direct access to the set Σ of all Web service operations, and that it performs a complete scan of Σ. If, however, the Web services are published on a remote registry, this implies a lot of network communication. It is possible to optimize the algorithm distributing it between the remote registry and the agent. The optimized version of the algorithm avoids inefficiency by delegating to the remote registry the processing step $U_C$, and by executing it before $U_E$ (as shown in Figure 7).

Preconditions relaxations: It is possible to progressively relax the constraints in the graph pattern $\gamma_i$ associated with an operation $\sigma_i \in \Sigma$; this corresponds to a partial fulfillment of preconditions.

Goal relaxation: It is possible to progressively relax the constraints in the graph pattern $\gamma_g$ associated with the goal $g$; this corresponds to a partial achievement of the agent's goal.

The refinements of Algorithm 1 are described in more detail in Sections 5.2, 5.3, and 5.4.

5.1. Execution example

We show in this section how Algorithm 1 allows Marco’s intelligent agent (which was introduced in section 3.5) to fulfill its goal.

The agent knowledge base (a fragment of which is shown in Figure 6) corresponds to the RDF description $\delta_a$ of the initial agent’s world state $\omega_a$. Let us assume that the buy operation of the Express Congo Buy Bookselling Service (introduced in Section 3.3) is one of the available Web service operations: $\sigma_b \in \Sigma$, $\sigma_b = (\gamma_b, \tau_b)$. In Algorithm 1, the important steps are at line 3, and 5.

Line 3 corresponds to the execution of the SPARQL query

$$\delta_c \leftarrow (\gamma_b, \{\delta_a\}, \text{CONSTRUCT } \tau_b)$$

which does two things:

- it checks that the preconditions $\gamma_b$ of the buy operation are satisfied, and in doing so it binds the variables in $\gamma_b$ with nodes in the RDF graph $\delta_a$; the knowledge base of Marco’s agent satisfies the preconditions of the buy operation;
- it computes $f(\sigma_b, \delta_a)$, that is the effects of $\sigma_b$ when invoked from the world whose description is the knowledge base of Marco’s agent; this computation yields the RDF graph $\delta_c$.

The execution of the CONSTRUCT query at line 3 checks the precondition of the buy operation (Figure 3) against the agent knowledge base (illustrated partially in Figure 6). At the same time, it constructs the operation’s effects (also shown in Figure 3). In so doing, it simulates a world state transition: the description of the resulting world state $\omega_1$ is given by $\delta_1 = g(\delta_a, \delta_c)$. Figure 8 shows the world state transition caused by the buy operation $\sigma_b$, and it shows the RDF graph $\delta_c = f(\sigma_b, \delta_a)$.

Note that the algorithm does not compute merging $\delta_1$ between $\delta_a$ and $\delta_c$, which is not essential to evaluate the fulfillment of the agent goal. However, Figure 8 shows also $\delta_1$ for completeness.

Finally, line 5 corresponds to the execution of the SPARQL query

$$(\gamma_g, \{g(\delta_a, \delta_c)\}, \text{ASK})$$

which checks the fulfillment of the agent goal $\gamma_g$ (see figure 5) in the world state resulting from the execution of $\sigma_b$. This step returns true, and therefore the algorithm adds the buy operation $\sigma_b$ to the set of selected Web service operations.

This example shows how the use of SPARQL queries combined with algorithm 1 can simulate the world state transitions described in Section 2, and lead the agent in dynamically selecting appropriate services to fulfill its goal.

5.2. Optimization for a Remote Registry

Algorithm 1 requires a full scan of the set $\Sigma$ of Web service operations. Assuming that Web services
are published on a registry, Algorithm 1 requires the agent to retrieve from the registry the details of every operation $\sigma_i \in \Sigma$, thus causing a huge amount of network traffic. Furthermore Algorithm 1 verifies the satisfiability of the preconditions of a Web service operation before unifying its effects with the agent goal, and this leads to inefficiency. Algorithm 2 is an optimization of Algorithm 1 exploiting a remote registry.

Algorithm 2 relies on the availability of the set $\Sigma_e$, which contains the Web service operations whose effects potentially allow achievement of goal $g$. Note that the elements of the set $\Sigma_e$ can be computed by a remote registry as explained below. However, such a computation relies on an approximation (the unconstrained service’s effects described below), and so the agent must check if the computed effects $\delta_e$ actually satisfy its goal $\gamma_g$ (see line 7 of algorithm 2).

The set $\Sigma_e$ is usually significantly smaller than $\Sigma$. The agent scans $\Sigma_e$ to check if its knowledge base $\delta_a$ can satisfy preconditions of any operations $\sigma_i \in \Sigma_e$, thus building the final set $\Sigma_g$ of selected operations.

The creation of $\Sigma_g$ is delegated to the function $\text{findOperationsByGoal}$, which is implemented by the remote registry $R$. Such a function corresponds to the processing step $U_E$ (Unification of Effects) discussed in Section 4. In order to implement the function $\text{findOperationsByGoal}$, the registry $R$ maintains an internal data structure whose elements are tuples of the form $(\sigma_i, \delta_i^*)$, where $\sigma_i$ is a Web service operation ($\sigma_i \in \Sigma$) and $\delta_i^*$ is an RDF graph. The registry $R$ computes each $\delta_i^*$ when the Web service corresponding to $\sigma_i$ is published. The computation of $\delta_i^*$ is based on the following steps:

- let $\sigma_i = (\gamma_i, \tau_i)$, where $\gamma_i$ is the graph pattern and $\tau_i$ the template pattern used in defining $\sigma_i$ (see Section 3.2);
- let $\tau_i^*$ be the template pattern obtained by $\tau_i$ by
substituting every occurrence of a variable with a blank node (informally we can see this substitution as analogous to Skolemization in first order logic); the substitutions preserves the structure of the original template pattern: \( \tau^*_i \) is the structurally-equivalent variable-free version of \( \tau_i \):

- let \( \gamma_0 \) be the empty graph pattern;
- let \( DS_0 \) be the empty data set;
- finally \( \delta^*_i = (\gamma_0, DS_0, CONSTRUCT \ \tau^*_i) \)

**Algorithm 2 Optimized discovery algorithm**

**Input:**
- \( \delta_0 \) : the agent knowledge base
- \( \gamma_g \) : the graph pattern describing the agent goal \( g \)
- \( R \) : a reference to the remote registry

**Output:**
\( \Sigma_g \) : the set of selected Web service operations

1: \( \Sigma_g ← \{ \} \)
2: \( \Sigma_e ⊂ \Sigma \): subset of operations whose effects allow achievement of goal \( g \)
3: \( \Sigma_e ← R.\text{findOperationsByGoal}(\gamma_g) \)
4: for all \( \sigma_i ∈ \Sigma_e \) do
5: \( \delta_e ← (\gamma_1, \{\delta_0\}, \text{CONSTRUCT } \tau_i) \)
6: if \( \delta_e \) is not empty then
7: if \( (\gamma_g, \{g(\delta_0, \delta_e)\}, \text{ASK}) = \text{true} \) then
8: \( \Sigma_g ← \Sigma_g \cup \{\sigma_i\} \)
9: end if
10: end if
11: end for
12: return \( \Sigma_g \)

Note that the use of the empty graph pattern \( \gamma_0 \) in the computation of \( \delta^*_i \) is equivalent to dropping the preconditions of \( \sigma_i \) represented by the graph pattern \( \gamma_i \). Having an empty graph pattern, the CONSTRUCT query can be safely executed on the empty data set. \( DS_0 \). Even though \( \tau^*_i \) and \( \delta^*_i \) have equivalent content, the final step is needed to accomplish a transformation from a template graph pattern to an RDF graph, as these objects are of different types. Finally, the use of the template pattern \( \tau^*_i \) ensures that the CONSTRUCT query always yields an RDF graph whose structure is equivalent to the one representing the effects of \( \sigma_i \). The RDF graph \( \delta^*_i \) represents the unconstrained effects of \( \sigma_i \) that is the effects of \( \sigma_i \) regardless of its preconditions. Algorithm 3 shows the implementation \( \text{findOperationsByGoal} \).

The registry \( R \) can easily check whether the effects of an operation \( \sigma_i \) allow achievement of a goal specified by a graph pattern \( \gamma_i \) simply by executing a SPARQL ASK query whose graph pattern is \( \gamma_g \) and whose Dataset is \( \{\delta^*_i\} \). Note that a concrete implementation may adopt different strategies to improve performance of Algorithm 3. An example of such strategies consist in using the Dataset: \( DS_{\Psi} = \{(u_0, \delta^*_0), (u_1, \delta^*_1), \ldots, (u_n, \delta^*_n)\} \). \( DS_{\Psi} \) is made of named graphs, each of which is identified by an URI \( u_i \), and contains the unconstrained effects of \( \sigma_i \). The registry may run a single SPARQL SELECT query returning the URIs \( u_i \) of the named graphs for which the graph pattern \( \gamma_g \) matches (see section 6).

**Algorithm 3 Function \text{findOperationsByGoal}** implemented by remote registry \( R \)

**Input:**
\( \Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_n\} \) : set of Web service operations
\( \gamma_g \) : graph pattern describing the agent goal \( g \)
\( \Psi = \{\{\sigma_0, \delta^*_0\}, \{\sigma_1, \delta^*_1\}, \ldots, \{\sigma_n, \delta^*_n\}\} \) : registry’s internal data structure

**Output:**
\( \Sigma_e ⊂ \Sigma \) : the subset of operations whose effects allow achievement of goal \( g \)

1: \( \Sigma_e ← \{ \} \)
2: for all \( (\sigma_i, \delta^*_i) ∈ \Psi \) do
3: if \( (\gamma_g, \{\delta^*_i\}, \text{ASK}) = \text{true} \) then
4: \( \Sigma_e ← \Sigma_e \cup \{\sigma_i\} \)
5: end if
6: end for
7: return \( \Sigma_e \)

5.3. Preconditions relaxation

In some cases it is useful to identify Web service operations whose effects allow fulfillment of a goal \( g \), but whose preconditions are not fully satisfied by the agent knowledge base. This is possible by relaxing some of the constraints expressed in the graph pattern \( \gamma \) defining the preconditions. SPARQL allows us to express relaxed constraints by substituting \( \gamma \) with an Optional Graph Pattern \( \rho \):

\[ \rho = \mu \ \text{OPTIONAL} \ o \]

where \( \mu \) and \( o \) are subgraphs of \( \gamma \), and represent respectively mandatory and optional constraints of the relaxed preconditions.
In general, given an RDF dataset $DS = \{\delta_a\}$ whose default graph is the agent knowledge base, and a Web service operation whose preconditions are expressed by the graph pattern $\gamma$, it is possible to build a set $R^\gamma_{DS}$ of relaxed preconditions that are satisfied by $DS$ and originates from $\gamma$. The elements of $R^\gamma_{DS}$ have different mandatory and optional constraints, corresponding respectively to different subgraphs $\mu$ and $\rho$.

Algorithm 4 builds $R^\gamma_{DS}$ when $\gamma$ is a Basic Graph Pattern. The algorithm works on $\gamma$ as a set of triple patterns, and it essentially identifies the subsets of $\gamma$ having solutions over the dataset $DS$. The algorithm splits $\gamma$ in two subsets: $\mu_i$ (the set of mandatory triple patterns) and $\rho_i$ (the set of optional triple patterns). Both $\mu_i$ and $\rho_i$ are Basic Graph Patterns, and $\rho_i$ is the complement of $\mu_i$ with respect to $\gamma$. The algorithm uses a SPARQL ASK query to check if $\mu_i$ has solutions over $DS$. If the ASK query returns true, then the algorithm builds an Optional Graph Pattern having $\mu_i$ as mandatory constraints, and $\rho_i$ as optional constraints.

Algorithm 4 builds the entire set $R^\gamma_{DS}$. However, it is sometime useful to find only some elements of $R^\gamma_{DS}$. For example, when $\gamma$ defines the preconditions of a Web service operation, it is desirable to relax as few as possible of the constraints expressed in $\gamma$. Algorithm 4 processes all possible $\mu_i$ ordered by decreasing value of their cardinality, thus progressively relaxing an increasing number of constraints $|\rho_i|$ increases when $|\mu_i|$ decreases. This means that in order to relax as few constraints as possible it is sufficient to stop Algorithm 4 as soon as $R^\gamma_{DS}$ is non-empty.

Every $\rho_i \in R^\gamma_{DS}$ has the form

$$\rho_i = \mu_i \ \text{OPTIONAL} \ \rho_i$$

where both $\mu_i$ and $\rho_i$ are Basic Graph Patterns. Note that since $\rho_i$ is a Basic Graph Pattern, $\rho_i$ essentially says that the triple patterns of $\rho_i$ are optional constraints, but must all hold at the same time. Alternatively, $\rho_i$ itself might be an Optional Graph Pattern, thus allowing for further relaxation of some triple patterns. However, there is no reason for making any triple pattern of $\rho_i$ more optional than any other. In fact the set $\rho_i$ is the complement of $\mu_i$ with respect to $\gamma$, and when the algorithm processes $\mu_i$, it has already checked all its supersets, and thus every element of $\rho_i$ is necessarily an unsatisfiable constraint when considered in conjunction with elements of $\mu_i$.

### Algorithm 4 Construction of $R^\gamma_{DS}$ when $\gamma$ is a Basic Graph Pattern

**Input:**
- $\gamma$: Basic Graph Pattern to be relaxed
- $DS$: dataset whose default graph is the agent knowledge base $\delta_a$

**Output:**
- $R^\gamma_{DS}$: set of Optional Graph Patterns derived from $\gamma$ and matching $DS$

1. $R^\gamma_{DS} \leftarrow \{\}$
2. $\mathcal{L} \leftarrow$ powerset of $\gamma$ ordered by inclusion, i.e. a complete lattice over $\gamma$
3. for $c = |\gamma|$ to 0 do
4.  for all $\mu_i : \mu_i \in \mathcal{L} \land |\mu_i| = c$ do
5.    if $(\mu_i, DS, ASK) == \text{true}$
6.      $\rho_i = \gamma \setminus \mu_i$
7.      $R^\gamma_{DS} \leftarrow R^\gamma_{DS} \cup \{\mu_i \ \text{OPTIONAL} \ \rho_i\}$
8.    end if
9.  end for
10. end for
11. return $R^\gamma_{DS}$

We can also sketch the construction of $R^\gamma_{DS}$ for a generic graph pattern $\Gamma$, based on the inductive definition (1) given in Section 3.1:

- if $\Gamma = \{p_0, p_1, \ldots, p_n\}$, that is $\Gamma$ is a Basic Graph Pattern, then we build $R^\gamma_{DS}$ using Algorithm 4;
- if $\Gamma = \{\gamma_0, \gamma_1, \ldots, \gamma_n\}$, that is $\Gamma$ is a Group Graph Pattern, then we build $R^\gamma_{DS}$ using an adapted version of algorithm 4, where $\Gamma$ is a set of graph patterns $\{\gamma_0, \gamma_1, \ldots, \gamma_n\}$, and $\mu_i$ and $\rho_i$ are Group Graph Patterns;
- if $\Gamma = \gamma_0 \ \text{OPTIONAL} \ \gamma_1$, that is $\Gamma$ is an Optional Graph Pattern, then we build $R^\gamma_{DS}$ in two steps:
  - (i) we build $R^{\gamma_0}_{DS}$, that is we relax the graph pattern $\gamma_0$ according to its structure
  - (ii) we build $R^\gamma_{DS} = \{\rho_i \ \text{OPTIONAL} \ \gamma_1 \mid \rho_i \in R^{\gamma_0}_{DS}\}$, that is we attach $\gamma_1$ as the right-most optional graph pattern to every element of $R^{\gamma_0}_{DS}$ (note that $\gamma_1$ is kept as the right-most optional graph, because the OPTIONAL operator is left-associative);
- if $\Gamma = \gamma_0 \ \text{UNION} \ \gamma_1 \ \text{UNION} \ \ldots \ \gamma_n$, that is $\Gamma$ is an Alternative Graph Pattern, then we need to relax at least one of the graph patterns $\gamma_0, \gamma_1, \ldots, \gamma_n$ according to its structure;
- if $\Gamma = \gamma_0 \ \text{FILTER} \ r$ and $\gamma_0$ has solutions over $DS$ (that is $(\gamma_0, DS, ASK)$ returns true) then we rewrite $\Gamma$ as $\Gamma = \gamma_0 \ \text{OPTIONAL} \ \{\gamma_0 \ \text{FILTER} \ r\}$; if $\gamma_0$ has no solutions over $DS$ (that is the query...
(γ₀, DS, ASK) returns false, then we build \( R_{DS}^γ \) in two steps:

(i) we build \( R_{DS}^{γ_0} \), that is we relax the graph pattern \( γ_0 \) according to its structure

(ii) we build

\[
R_{DS}^γ = \{ ρ_i \text{ OPTIONAL } \{ γ_0 \text{ FILTER } r \} : ρ_i \in R_{DS}^{γ_0} \}
\]

that is we make \( Γ \) the right-most optional graph pattern of every element of \( R_{DS}^γ \).

The relaxation of a graph pattern with a FILTER expression may also take into account the constraints expressed in the FILTER itself, and selectively move unsatisfiable constraints to an optional subgraph pattern. However, this requires special handling of variables in order to avoid scope issues (see [12]). The relaxation techniques described above also affects the results ranking of the matchmaker. In general, services whose preconditions are only partially satisfied have a lower ranking than services whose preconditions are fully satisfied. Additionally, it is possible to establish a partial order among the approximate matches, based on the cardinalities of the sets of optional constraints in their relaxed preconditions. Approximate matches having fewer optional constraints have higher ranking.

From a general perspective, the relaxation of services preconditions is not only useful for approximate matchmaking, but it provides also the means to implement sequential composition of services. The approach can be summarized as follows:

- when the agent discovers a service \( σ_i = (γ_i, τ_i) \) whose effects potentially satisfy its goal, but whose preconditions \( γ_i \) are not satisfied with its current knowledge base \( δ_a \), it tries to relax them over \( δ_a \);
- the relaxation yields a set of graph patterns containing optional subgraphs, which represent the currently unsatisfiable conditions;
- the agent adopts the optional subgraphs as a new goal, and attempts to satisfy it by recursively repeating the process;
- if the agent manages to satisfy the new goal, then it can eventually satisfy the original preconditions.

This approach yields a sequence of services. The preconditions of each service in the sequence are guaranteed to be satisfied either by the agent knowledge base, or by the effects resulting from the execution of the preceding services of the sequence. Ultimately, the last service in the sequence is that service \( σ_i \) whose effects ensure the satisfiability of the original agent’s goal. Although an in depth description is out of scope for this paper (where we concentrate on service discovery), we observe that the approach outlined above is essentially a form of regression planning [58] [23], whose efficiency can be increased by introducing the notion of service cost and by using it to reduce the branching factor of the backward recursive search [60].

5.4. Goal relaxation

In some cases it is interesting to identify Web service operations whose effects allow for a partial fulfillment of an agent goal \( g \). This is possible by relaxing some of the constraints expressed in the corresponding graph pattern \( γ_g \).

The approach described in Section 5.3 can be used to relax a goal graph pattern \( γ_g \) provided that we use an appropriate dataset \( DS \). The achievement of a goal \( g \) is tested on an RDF graph describing the effects of a Web service operation \( σ \) (see Algorithm 1). For testing the goal achievement we can disregard the preconditions of \( σ \), and therefore we can use its unconstrained effects \( δ^* \), as defined in section 5.2. The relaxation of a goal graph pattern can be performed on the dataset \( DS = \{ δ^* \} \), whose default graph is given by the unconstrained effects of a Web service operation \( σ \). Note that it is straightforward to extend algorithm 3 in order to find the set of Web service operations whose effects allows for partial fulfillment of a goal \( g \).

Although the goal relaxation over the unconstrained effects of Web service operations is technically feasible in a remote registry, it is preferable to let the agent relax the constraints of its goal according to some heuristics. Delegating goal relaxation to the agent allows for better optimization, and also more intelligent approaches.

6. Implementation and Evaluation

In this section we describe the prototype implementation of our agent and service registry. We name our agent SPARQLent (SPARQL Agent), because its algorithms and functioning leverage the computational model of SPARQL queries as described in the previous sections.

In our prototype implementation of registry and SPARQLent we used the Jena [9] platform for han-

\[\text{http://jena.sourceforge.net/}\]
duling Semantic Web languages, and ARQ \(^6\) for processing SPARQL.

The prototype service registry maintains internal data structures storing relevant information from service descriptions, and it computes for each service the corresponding unconstrained effects as described in Section 5.2. Such data structures allow for the implementation of Algorithm 3. In order to improve the performance of this algorithm, we exploit SPARQL features to directly query the registry dataset \(DS_\Psi = \{(u_0, \delta_0^\gamma), (u_1, \delta_1^\gamma), \ldots, (u_n, \delta_n^\gamma)\}\). More precisely, the answer to \(\text{findOperationsByGoal}(\gamma_\alpha)\) is computed with the SPARQL query

\[
\text{SELECT } u_i \text{ WHERE } \text{GRAPH } u_i \{ \gamma_\alpha \}
\]

which returns all services \(\sigma_i\) whose unconstrained effects \(\delta_i^\gamma\) allow for achievement of the agent goal \(\gamma_\alpha\).

A crucial implementation aspect for SPARQLent is related to the maintenance of its knowledge base, which, in our implementation, consists of an OWL-DL ABox. We observe that all the algorithms presented in the previous sections treat the agent’s knowledge base as an RDF graph. In general an RDF graph can “contain” implicit knowledge (knowledge derivable from the graph according to RDF and/or OWL semantics). It is important to have a strategy by which the implicit knowledge is made available for reasoning about services. As with many Semantic Web programming frameworks, our approach does not dictate a particular reasoning paradigm for deriving implicit knowledge, but allows for the adoption of any paradigm that can operate over an RDF knowledge base. In the SPARQLent implementation, this is accomplished by calling on the external OWL-DL reasoner Pellet \(^7\) [64] to derive the inferences and make them explicit. After the explicit inferences are returned from Pellet, they are simply asserted into SPARQLent’s KB. \(^8\)

Interestingly, recent versions of Pellet allow for incremental consistency checks: after having loaded an ontology, it is possible to make changes to an ABox, and verify the consistency with respect to the ontology. Additionally, one can use the Pellet explanation service, which allows for programmatically retrieving explanations of inconsistency. Thus, Pellet allow us not only to infer implicit knowledge, but also to ensure consistency of the agent knowledge base \(\delta_\alpha\) when merging it with an RDF graph \(\delta_e\) representing the effects of some service (see function \(\eta\) and related discussion in Section 2). Incidentally, we observe that our approach of removing a minimal number of assertions from \(\delta_e\) to ensure consistency when merging with a \(\delta_e\) has the drawback that it may progressively reduce the initial knowledge base. This is particularly evident when iteratively merging several RDF graphs. Although a more sophisticated implementation of belief revision is out of scope for our work, we observe that our approach and implementation completely decouples the SPARQLent core operations (discovery and relaxation) from belief revision, thus leaving room for future improvements in the latter without affecting the former.

Finally, we observe that our SPARQLent implementation provides both exact and relaxed service matchmaking. Relaxed discovery also includes in the result services whose preconditions are only partially satisfied by the agent knowledge base, and it uses the techniques described in Section 5.3.

We evaluated our approach in a real world use case in the e-Government domain, where we used a service registry and SPARQLent to solve the problem of automatically selecting assistance and welfare services for citizens [61]. We present here the experimental evaluation of our approach with respect to the OWL-S Services Retrieval Test Collection (OWLS-TC\(^9\)), a test collection for evaluating OWL-S service matchmaking algorithms. The latest version of OWLS-TC, which at the time of writing is version 3, includes 1007 Web service descriptions written in OWL-S 1.1, and 29 queries with corresponding relevance sets (true answers).

Unfortunately, the semantics of OWLS-TC service descriptions are only partially specified, because they give only the input/output types (by referring to concepts in various ontologies) without describing preconditions and effects. In our view, this may be taken as an indication that a fully adequate expression language for specifying preconditions and effects is currently missing. In order to use our approach with OWLS-TC we developed a \textit{Transformer}, which generates SPARQL graph patterns from OWLS-TC service and query descriptions. More precisely:

1. OWLS-TC service descriptions essentially consist of OWL-S atomic processes giving the semantics of inputs and outputs by referring to concepts in various ontologies. Given the list of input concepts

\(^6\) http://jena.sourceforge.net/ARQ/  
\(^7\) http://clarkparsia.com/pellet/  
\(^8\) Other strategies are possible. For example, some triple store systems transparently incorporate inferred triples in query answering.  
\(^9\) http://projects.semwebcentral.org/projects/owls-tc/
\(S_I = (I_1, I_2, \ldots, I_m)\), and the list of output concepts \(S_O = (O_1, O_2, \ldots, O_n)\) of an OWL-S atomic process, we characterize the service \(\sigma_i\), by building a pair of graph patterns \((\gamma_i, \tau_i)\) where:

- \(\gamma_i\) is a Basic Graph Pattern containing a triple pattern for every input concept \(I_i \in S_I\); such triple patterns have the form \(?x_i \text{ rdf:type } I_i\);
- \(\tau_i\) is a Template Pattern containing a triple pattern for every output concept \(O_i \in S_O\); such triple patterns have the form \(?:o_i \text{ rdf:type } O_i\), where \(?:o_i\) is a blank node.

Similarly to services, OWLS-TC queries consist of OWL-S atomic processes giving the semantics of inputs and outputs by referring to concepts in various ontologies. Given the list of input concepts \(Q_I = (I_1, I_2, \ldots, I_m)\), and the list of output concepts \(Q_O = (O_1, O_2, \ldots, O_n)\) of a query, we interpret its description in the following way:

- \(Q_O\) gives the list of required outputs, and therefore we use it to generate a Basic Graph Pattern \(\gamma_\text{q}\) representing the goal of our SPARQLent; \(\gamma_\text{q}\) contains a triple pattern for every output concept \(O_i \in Q_O\); such triple patterns have the form \(?:\text{out} \text{ rdf:type } O_i\), where \(?:\text{out}\) is a blank node;
- we use \(Q_I\) to build the content of the SPARQLent knowledge base: for each \(I_i \in Q_I\) we create a blank node of type \(I_i\).

We essentially say that the SPARQLent goal is structured so as to obtain instances of all the output concepts specified in the query, and to check that its knowledge base contains instances of the input concepts specified in the query.

The transformations described above allow us to (i) store the OWLS-TC service descriptions in our registry by computing the corresponding unconstrained effects, and (ii) use our SPARQLent to answer OWLS-TC queries. We observe that the transformation outlined above also takes into account the subsumption hierarchies corresponding to the various input and output concepts, thus externalizing implicit knowledge.

We have implemented the IMatchmakerPlugin interface, which allowed us to plug our code into the SME2 test tool\(^{10}\). The Semantic Web Service Matchmaker Evaluation Environment (SME2) is an open source tool for testing different semantic matchmakers in a consistent way. SME2 uses OWLS-TC to provide the matchmakers with service descriptions, and to compare their answers to the relevance sets of the various queries. SME2 gives several measures of the performance and effectiveness of a matchmaker:

- **Total execution time.** The time required by the matchmaker to parse all services descriptions and to answer all queries.
- **Memory usage.** Statistics on the memory usage during the total execution time.
- **Query response time.** The time required by the matchmaker to answer a single query (this does not take into account the time spent in the initialization phase for parsing service descriptions).
- **Precision, recall and fallout.** These are standard measures for evaluating the performance of information retrieval systems [2]. Given a query \(q\), and a set of items \(D\), let \(R_q\) be the relevance set of \(q\) (i.e., the set of relevant items for the query \(q\)), and let \(A_q\) be a computed answer set (i.e. a set of items returned as answer to \(q\)). We have that \(\text{precision} = \frac{|R_q \cap A_q|}{|A_q|}\), \(\text{recall} = \frac{|R_q \cap A_q|}{|R_q|}\), and \(\text{fallout} = \frac{|(D \setminus R_q) \cap A_q|}{|D \setminus R_q|}\). Intuitively, precision is the fraction of the answer set \(A_q\) that is relevant to the query, whereas recall is the fraction of the relevance set \(R_q\) that has been retrieved. Finally, fallout represents the fraction of non-relevant documents \((D \setminus R_q)\) that are retrieved.

SME2 computes macro-averaged values of precision and fallout over all queries, thus giving equal weight to every query. The macro-averaging is obtained by (i) measuring precision (fallout) for each query separately at standard recall levels (with \(\lambda\) being the number of levels), and then (ii) computing the mean value for the measures at each level \(i\) with \(0 \leq i < \lambda\). We observe that the computation of macro-averaged precision and recall requires ranking of the answer set \(A_q\); we adopted a subsumption-based ranking strategy as described in [48] to order the results returned by the SPARQLent discovery algorithm (a more sophisticated ranking strategy may improve the test results).

Figures 9(a) and 9(b) compare the results of our SPARQLent with some variants of OWLS-MX, an hybrid Semantic Web services matchmaker providing both traditional input/output subsumption and approximate matching based on information retrieval techniques [35,36]. We tested three variants of OWLS-MX:

- **OWLS-M0 (logic-based)** is a purely logic-based matchmaker that relies on subsumption reasoning.

\(^{10}\)http://projects.semwebcentral.org/projects/sme2/
- **OWLS-MX textSim only (Cos)** compares query descriptions and service descriptions using text similarity measures. It performs text similarity matching on concept descriptions: it considers inputs and outputs separately (looking at concatenated concept description strings of either inputs or outputs as a plain text document), and then averages the results to obtain an overall similarity value.

- **OWLS-M3 (MX2, hybrid, Cos)** is an hybrid matchmaker performing both logic-based subsumption reasoning and text similarity matching. It uses text similarity to avoid false positives arising, for example, from poor concept definitions or incomplete service descriptions.

Technically, the only fair comparison is between **OWLS-M0** and SPARQLent, which are both purely logic-based matchmakers. The other OWLS-MX variants exploit also text similarities in the services and/or concepts descriptions, thus detecting also aspects that escape the formal descriptions. For example, OWLS-TC service and query descriptions do not have any formal preconditions, but some of them have text comments that informally state preconditions: while SPARQLent and **OWLS-M0** cannot exploits such comments, the other variants of OWLS-MX can.

Nevertheless, the comparison among SPARQLent and OWLS-MX variants highlights some interesting results. Firstly, Figure 9(a) shows that SPARQLent has better precision than **OWLS-M0**: the same figure shows also that the two OWLS-MX variants using text similarity outperform SPARQLent. Figure 9(b) highlights that the fallout is substantially smaller for SPARQLent than for OWLS-MX variants: at the highest recall level, SPARQLent fallout is 0.035, compared to 0.232 for **OWLS-M3 (MX2, hybrid, Cos)**, 0.240 for **OWLS-MX textSim only (Cos)**, and 0.931 for **OWLS-M0 (logic-based)**.

Figure 10 illustrates the response times for the three matchmakers SPARQLent, **OWLS-M0 (logic-based)**, and **OWLS-M3 (MX2, hybrid, Cos)** when answering the 29 queries (Q01...Q29) of OWLS-TC; the last column (Avg) gives the average response time for the three matchmakers.

The experimental results show that SPARQLent is considerably faster than the OWLS-MX variants. A remarkable exception is query Q26, where SPARQLent is much slower than the others. This is due to the fact that query Q26 has no output (it searches for services that check the credit card account of an authorized person, and adds a given book to her/his shopping cart). The lack of outputs originates an empty goal for our SPARQLent, thus all services in the registry are returned, and the amount of time to process all of them is high. The average values show that SPARQLent is about 2.7 times faster than **OWLS-M0 (logic-based)** (860 ms versus 2321 ms), and about 3.7 times faster than **OWLS-M3 (MX2, hybrid, Cos)** (whose average query response time is 3200 ms).

Finally, Figure 11 illustrates the memory usage...
statistics for the three matchmakers SPARQLent, OWLS-M0 (logic-based), and OWLS-M3 (MX2, hybrid, Cos). The diagram clearly shows that SPARQLent requires less memory than the OWLS-MX variants. The graph corresponding to SPARQLent is more coarse grained than the others: this is due to the differences in the respective total execution times. Since SPARQLent is almost 12 times faster than the OWLS-MX matchmakers, the SME$^2$ testing tool collects only 7 sample values during its execution (compared to about 80 sample values for the OWLS-MX matchmakers). Although a precise explanation of the difference between SPARQLent and OWLS-MX performance is out of scope for this paper (and it would probably require an analysis of the actual implementations of the two matchmakers), we believe that the use of SPARQL (as opposed to pure Description Logic reasoning) during the matchmaking process allows for a substantial improvement in performance.

7. Related Work

The problem of action characterization and the related problem of action selection to meet a particular set of requirements have been investigated for several decades under various research headings, including deductive program synthesis, automatic programming, AI planning, e-science, Web services, Grid services, agent-based systems, and Semantic Web services. Because of space constraints, we can only mention examples from the last two of these areas. For a more extensive summary of related work, see [71].

The field of agent-based systems (ABS) includes a significant body of work on characterizing and reasoning about agent capabilities, which often are conceived as remotely invocable procedures (in
some ways, a forerunner of Web services). As in earlier work on AI planning, the common denominator of many approaches is the representation of preconditions and effects, often with additional information about the inputs and outputs of the operations that an agent provides. LARKS [67], for example, employs \textit{InConstraints} and \textit{OutConstraints}, expressed as Horn clauses referring to inputs and outputs, respectively, for this purpose. This approach, while flexible, requires special handling for these Horn clauses outside of the description logic framework that underlies LARKS’s ontology, using theta-subsumption. (Theta subsumption is one of five matching filters that can be configured for matching requested and provided capability descriptions.) Our approach, in contrast, remains within the representational framework defined by RDF and SPARQL, benefits from the additional expressiveness afforded by the use of SPARQL, and allows the developer to leverage the broad range of tools and libraries that have accompanied the standardization of these technologies.

BDI frameworks, such as exemplified by AgentSpeak and PRS, have already been mentioned in Section 4; these frameworks also rely on preconditions and postconditions. In PRS, pre- and post-conditions are logical formulas composed of terms, variables, function and predicate symbols, conjunction, disjunction, and negation. In AgentSpeak, a plan is formed by a triggering event, a context, and a sequence of basic actions. The triggering event denotes the purpose of the plan, and it ensures the reactivity of the agent. The context is essentially a conjunction of predicate symbols, and it represents the precondition: the plan is applicable only if the context is a logical consequence of the agent’s current beliefs. The last part of the plan gives a list of basic actions (beliefs updates, subgoals, actions) representing the plan’s effects. As noted earlier, an AgentSpeak agent has a set of predefined plans in a local library, whereas our agents include the ability to dynamically discover local or remote Web service operations allowing achievement of their goals.

Generally speaking, BDI agents lack built-in capabilities for lookahead planning and for creating new plans. The authors of [59,13] propose an approach based on hierarchical planning by leveraging the common aspects of BDI systems and Hierarchical Task Network planners. In [13] they introduce a hybrid planner that combines a classical planner and a hierarchical planner: the former synthesizes abstract plans bringing about a goal state, and the latter makes use of available domain knowledge to (possibly) find successful decompositions of potentially incorrect abstract plans (potential incorrectness of abstract plans is due to effects clobbering preconditions of later actions). The authors discuss also how to identify non-redundant maximally-abstract plans by improving a solution obtained by their hybrid planner. While this work focuses mainly on (hierarchical) planning in the context of BDI agents, we propose here a novel approach for the description and discovery of Semantic Web services, and we show how a BDI-like agent can take advantage of such services. Our goal-directed discovery algorithms (and the possible sequential composition of services outlined in Section 5.3) implement an approach similar to that of the classical planner used in [13], but we do not explore the use of hierarchical planning.

ABS has explored a variety of styles of matchmaking. For example, in the Open Agent Architecture (OAA) [10], the basic capability description is a logic programming predicate structure (which may be partially instantiated), and matchmaking is based on unification of goals with these predicate structures. In addition, both goals and capabilities declarations may be accompanied by a variety of parameters that modify the behavior of the matchmaking routines. Although our approach does not make use of unification, it achieves greater flexibility by building on SPARQL.

As noted earlier, these same problems have been the focus of research on Semantic Web Services (SWS). The first challenge in SWS has been the enrichment of service descriptions. OWL-S, the pioneering effort in this field, introduced the expression of preconditions and effects in a Semantic Web-compatible manner. The Semantic Web Services Framework (SWSF) [3] builds out from OWL-S by including some additional concepts (especially in the area of messaging); employing first-order logic, which is more expressive than OWL; and drawing on the axiomatization of processes embodied in the Process Specification Language (PSL). WSMO [57] shares many of the same objectives and approaches as OWL-S and SWSF. As noted earlier, WSMO distinguishes two types of preconditions (called \textit{assumptions and preconditions}), and two types of postconditions (called \textit{postconditions and effects}).

Based on the formal model [57] for services and goals defined by WSMO, [32] builds a conceptual model for the automatic location of services that includes the following steps: (i) goal discovery (reuse
of predefined goals), (ii) goal refinement (refinement of the discovered goals based on the given requester desire), (iii) service discovery (the discovery of relevant abstract services), and (iv) service contracting (the contracting of concrete services to fulfill a requester goal). The fourth step (service contracting) takes into account preconditions and postconditions, which are defined respectively by a predicate on inputs, and a predicate on inputs and outputs. [66] extends the approach of [32] by differentiating two notions of goals: the goal template (a generic objective description that is defined at design time) and the goal instance (a concrete client request that is created at runtime by instantiating a goal template). The approach proposes a two phased discovery model: at design-time the system determines usable Web services for goal templates, and saves them for later use; at run-time a goal instance is used in considering only those Web services identified by the corresponding goal template, thus reducing the number of matchmaking operations. By comparison, our work does not require goal templates, and it can effectively distribute the discovery operation between the agent and a services registry (see algorithm 2). On the other hand, the design-time phase of [66] does not rely on the availability of instance data, and thus may be applicable in some settings where our approach is not.

IRS-III [15] employs WSMO’s conceptual model to provide a broker-based approach in which a client sends a request expressing a desired outcome or goal and a broker discovers potentially relevant Web services, selects the Web services that best fit the incoming request, mediates any mismatches at the conceptual level, and invokes the selected Web services. In IRS-III, as in our approach, service selection relies on instance data and is based on preconditions and assumptions, but also considers other elements including input types and non-functional properties.

Several approaches for semantic service discovery using OWL-S have been proposed. [40] proposes a software framework for matchmaking, and an algorithm based on subsumption reasoning. During the matchmaking process, a service profile and a service request are considered to match when all the outputs of the request goal are matched against all, or a subset of service output, and as well all the inputs of the service are matched against all, or a subset of users’ goal. Different degrees of matching were identified: (i) exact match: the outputs, respectively the inputs being matched are exactly the same, (ii) plug-in match: the output of the service subsumes the output of the request, (iii) subsumes match: the output of the request subsumes the output of the service and (iv) fail: no matching services were found for the request goal. The DAML-S/OWL-S matchmaker [65] allows creating an OWL-S Profile of a Web service and publishing the Web service to a UDDI registry using the OWL-S profile to UDDI mapping described in [47]. Compared with the previous approach presented in [40], other degrees of matching are considered as well: i.e. intersection match (in this case the intersection of request R and advertisement A is satisfiable) and disjoint match (none of the matches previous presented). The strength of the match is decreasing from the Exact Match to Disjoint Match. Most work on semantic service discovery using OWL-S takes into account only matching of input and output types (possibly with subsumption), but does not take into consideration preconditions and postconditions. A possible reason for this is OWL-S’s openness with respect to the means of expressing preconditions and postconditions. Our work shows how SPARQL can effectively be used for this purpose, enabling uniform characterization not only of service preconditions and postconditions, but also of agents’ goals. Additionally, we provide a SPARQL-based mechanism to progressively relax conditions, which is useful to identify partial matches.

It is worth noting that much of the prior work does not assume the availability of instance data (ABox), as we have done. As noted earlier, we rely on instance data for checking the truth of services’ preconditions against the current world state. In contrast, other approaches may compare preconditions at a conceptual level via equivalence or implication using a theorem prover, possibly at a cost of greater algorithmic complexity. There is a tradeoff associated with the reliance on instance data about the current world state. On the one hand, this reliance, as we have described it, constrains the final selection of a service to be done in temporal proximity to its invocation (i.e., “just before” its invocation), or at least requires an agent to ensure that the relevant Abox content does not change between selection and invocation. On the other hand, checking preconditions against instance data allows for a broader range of conditions to be checked, and in a more concrete fashion. (For example, a precondition could check whether the user’s bank account balance is currently large enough to pay for the use of a particular service.)
A different approach to approximate matchmaking is taken by the OWLS-MX system [35,36]. This system is a hybrid Semantic Web services matchmaker combining traditional inputs/output subsumption with approximate matching based on similarity computations. Such similarity computations exploit implicit semantics by analysing patterns or relative frequencies of terms in service descriptions as computed by techniques from data mining, linguistics, or content-based information retrieval. We observe also that a similar matchmaker (WSMO-MX) exists also for the WSMO formalism [37]. Although our work on relaxing graph patterns also addresses the area of approximate services matchmaking, it relies entirely on logical techniques. Section 6 describe some comparative results of our implementation and some OWLS-MX variants.

We regard the idea of using information retrieval techniques for computing approximate concept matching as particularly attractive, especially from the perspective of the recently proposed iSPARQL [34], a proposed extension of SPARQL to enable approximate triple pattern matching based on custom similarity functions. iSPARQL has already been applied to approximate matchmaking of OWL-S services profiles [33]: the iMatcher (imprecise Matcher) uses iSPARQL as a general purpose matching algorithm to identify OWL-S services profiles (advertisements) that have a high degree of similarity with an OWL-S request profile (query). This approach essentially achieves the same objectives as the OWLS-MX matchmaker. Since iSPARQL is an extension of SPARQL, it perfectly fits within our approach, and we regard iSPARQL as a promising means by which to extend our work to enable imprecise services matchmaking:

- the definition of imprecise goals: the graph pattern \( \gamma_g \) representing the agent goal can be expressed using iSPARQL, thus enabling the definition of approximate matches;
- the definition of imprecise preconditions: the graph pattern \( \gamma_i \) representing the preconditions of a Web service can use iSPARQL to enable the definition of less stringent constraints.

The combination of our approach to graph pattern relaxation and iSPARQL’s approximate matching of triple patterns promises to be an interesting research direction for blending logic-based and similarity-based service discovery.

Finally, we observe that our approach is widely applicable. SPARQL can be adopted as expression language not only by OWL-S, but also by other semantic Web services frameworks: for example, [30] describes the high-level integration of our approach with SAWSDL. Additionally, [61] exploits OWL-S and SPARQL to describe assistance services in the e-Government domain. Although such services are not Web services, the flexibility of the OWL-S profile can accommodate their description, including the definition of their eligibility criteria (i.e. preconditions) and effects in SPARQL. Our approach can therefore be adapted to enable the automatic discovery of services for citizens, by matching the eligibility criteria of assistance services with RDF-based citizens’ profiles.

8. Conclusions

We have shown how SPARQL can be used for describing and discovering Web service operations, and given a conceptual framework for formalizing and understanding this approach. To make this possible, we propose the use of SPARQL to express the preconditions and postconditions of services, as well as the goals of agents. Once this has been done, SPARQL query evaluation can be used as the basis for service discovery, by checking the truth of a precondition, constructing the postcondition that results from the executing a service, and determining whether a service execution with those results will satisfy the goal of an agent. This approach also allows for SPARQL features to be leveraged in optimizing the discovery algorithm, and in creating relaxed forms of preconditions and goals for use in discovery.

As noted earlier, this is a general approach that can be used with any SWS framework based on knowledge representation using RDF or OWL. OWL-S is one such framework with which this approach fits neatly, and indeed this approach contributes in two significant ways to OWL-S practices. First, we believe that this approach establishes SPARQL as a compelling answer to the question of which language to use with OWL-S for expressing preconditions and effects. The relationship of SPARQL to RDF and OWL knowledge bases is already well understood and well-defined, and SPARQL provides both a high degree of expressiveness and of flexibility. Second, the use of SPARQL makes available valuable building blocks for constructing OWL-S tools. Tools and agents for OWL-S are normally implemented atop generic RDF or OWL components. Most such components
knowledge bases, editing tools, code libraries, reasoning environments, and so forth – already have well integrated support for SPARQL. These existing SPARQL implementations, in turn, can be leveraged in the construction of OWL-S editing, discovery, planning, and enactment components that handle preconditions and effects expressions. For these reasons, we recommend that SPARQL be regarded as the default language for expressing preconditions and effects with OWL-S.

8.1. Future Directions

Looking ahead, we see promising research directions in these areas:

- **Non-functional aspects of services.** In addition to preconditions and postconditions, SWS researchers have also experimented with the use of additional kinds of information in discovery, such as quality of service, response time, and other kinds of performance characterization. These aspects of service characterization have often been referred to as the “non-functional” aspects, in distinction to the characterization of inputs, outputs, preconditions, and postconditions, which are called the “functional” aspects. While the consideration of non-functional aspects is out of scope for this article, we believe that it would be worthwhile to explore the use of SPARQL in evaluating these aspects, and in creating a single approach to discovery that considers both functional and non-functional aspects.

- **Ontology mediation.** In this paper, we have made the simplifying assumption that all service descriptions and goals (for use within a given community of agents) are expressed in terms of a single, common, shared set of ontologies. Thus, we have not concerned ourselves with the need to mediate between different ontologies used by different agents, or used by different service providers. Mediation capabilities, however, are recognized as central to the broader success of the Semantic Web, including SWS. We believe that the same SPARQL mechanisms that make SPARQL valuable for use in expressing preconditions and postconditions also can be leveraged in providing mediation capabilities. In particular, a CONSTRUCT query can be viewed as an if-then rule: if the WHERE part of the query succeeds, the matching pattern of triples can be translated into the triples specified by the CONSTRUCT part of the query. (In this usage, the properties and classes mentioned in the WHERE part belong to the source ontology of the mediation, whereas those mentioned in the CONSTRUCT part belong to the target ontology.) However, a naive application of this idea would be limited to simple forms of mediation. This is because, in general, mediation requires forward chaining of rules. To address this issue, it would be interesting to explore whether a collection of CONSTRUCT queries could be applied iteratively until a fixpoint is reached.

- **Conditional effects in OWL-S.** In this paper, we have dealt with preconditions and postconditions. OWL-S additionally has elements known as conditional effects. It should be straightforward to extend our approach to handle the evaluation of conditional effects.

- **Minimal sufficient conditions.** In some systems, other kinds of reasoning are done about preconditions, in addition to evaluating their truth against a knowledge base. For example, some systems need to determine the minimal set of assertions that would need to be added to a knowledge base to make a given precondition true; this set of assertions is sometimes called the minimal sufficient condition for use of a service. (The minimal sufficient condition tells an agent what it needs to make true to enable the use of the service.) It would be interesting to consider how best to determine minimal sufficient conditions when preconditions and postconditions are expressed in SPARQL.

- **RESTful Web services.** Services based on the representational state transfer (REST) [19] paradigm enable the implementation of a lightweight service oriented architecture, and have recently pervaded the Web due to their simplicity and flexibility. Some work has been done to bring semantics to RESTful services following the SAWSDL approach [63], and embedding semantic annotation of input and output types within the HTML pages describing the services themselves. It should be straightforward to extend our approach to RESTful Web services, thus giving a formal definition of their preconditions and effects, and enabling software agents to autonomously interact with RESTful Web services (possibly on behalf of human beings). In fact, current interaction with RESTful Web services typically involves a user filling a form and submitting it to the service (using HTTP POST). If such a form were accompanied
References


[40] L. Li, I. Horrocks, A software framework for matchmaking based on semantic web technology (2003). URL citeseer.ist.psu.edu/li03software.html


URL http://jena.hpl.hp.com/~afs/SPARQL-Update.html


URL http://dx.doi.org/10.1016/j.websem.2007.03.004


