Two-phase Semantic Web Service Discovery Method for Finding Intersection Matches using Logic Programming

László Kovács, András Micsik, Péter Pallinger
MTA SZTAKI
Computer and Automation Research Institute
of the Hungarian Academy of Sciences
Department of Distributed Systems
{laszlo.kovacs, micsik, pallinger}@sztaki.hu

Abstract

Discovering Web Services based on logical matching of capabilities is a new requirement for Semantic Web Services which cannot be solved with traditional information retrieval (IR) techniques. Building fast and precise logical discovery engines is an ongoing challenge of the Semantic Web community. This paper presents the discovery engine implemented for the INFRAWEBS project which combines a traditional IR-based pre-filtering step and a logic-based matching implemented in Prolog. The logic-based step of discovery uses a novel technique based on Prolog-style unification of terms. This approach performs well in finding matches of intersection type, and it also provides possibilities to compare, rank and explain these matches.

1. Introduction

Using Semantic Web Services is often broken into the main steps of discovery, selection and execution. Discovering Web Services based on logical matching of capabilities is a new requirement for Semantic Web Services which cannot be solved with traditional information retrieval (IR) techniques. Building fast and precise logical discovery engines is an ongoing challenge of the Semantic Web community.

In this paper we present the discovery engine developed in the INFRAWEBS project of FP6. The INFRAWEBS project develops an ICT framework, which enables software and service providers to generate and establish open and extensible development platforms for Web Service applications. The INFRAWEBS project divides the life-cycle of Semantic Web Services in two different phases: Design Time and Runtime. During the Design Time phase various tools and editors support the creation of semantic descriptions for existing Web Services. The resulting ontologies, goals and Semantic Web Services are made accessible in a distributed registry. WSML [14] was chosen as the language for describing these semantic entities. The runtime phase involves discovery, selection and execution of the semantic web services. The runtime environment also collects quality of service data for semantic web services which are fed back to the phase of discovery and selection.

1.2. Discovery

The discovery engine in our scenario receives a WSML goal as input and it has to provide a list of matching Semantic Web Services possibly coupled with additional information that supports ranking and selection.

Discovery implementation has three steps: a pre-filtering step, a step for logical matching and a finalizing step to prepare the result.

The aim of the pre-filtering step is to narrow the list of candidates using traditional text-processing (keyword matching) algorithms.

The logical matching is performed on the precondition, assumptions, postcondition and effects of the goal and service.

In the final step the list of matching services are enhanced with QoS data based on past execution experience. QoS data is collected by another module of the framework, and can be used for service selection.

The rest of the paper describes the discovery component implemented for the INFRAWEBS project and compares it with related work.

2. Keyword-based discovery

Each semantic web service and goal capability definition may be seen as a structured text document. This gives us the possibility for a preliminary selection of Web Services using classical keyword-based discovery, but at the level of ontology concepts.

The WSMO deliverable on discovery [11] suggests the keyword-based approach to be used in conjunction with the keywords given by capability editors and listed as non-functional properties (metadata about web services). However, the correctness and quality of such natural language descriptions are hard to ensure and control. Empty or faulty descriptions make services inaccessible, even when their capabilities are defined correctly. Multilinguality of metadata is another problem of this solution.

Our approach is different from the WSMO idea, because here the axioms are indexed instead of the non-functional properties. The proper formulation of the axioms is needed for the correct use of Semantic Web Services, therefore the quality of the index data is implicitly ensured.

It is obvious that the result of keyword-based discovery may contain semantically non-matches (e.g. the capability for selling tickets everywhere *except* to Budapest). The key criterion is not to filter out any good semantic match in this phase. In [9] we show that by simple conditions on the ontology structure this can be guaranteed. For example, in case of a single homogeneous set of ontologies, the description of each relevant capability requires the use of certain concepts; therefore the goal and service capabilities must both contain these concepts.

The advantage is the very good response time compared to logic reasoning, while the disadvantage is that logically incorrect results are also collected. Therefore, this operation is ideal for a so-called pre-filtering, to reduce the number of web services for which the time-consuming logical matching has to be calculated.

The Organisational Memory (OM) component of the INFRAWEBS framework is specialized on textual queries. Its main role is to index all available textual documents with respect to web services, including WSDL and WSML. With its indexes OM helps the creators of Semantic Web Services to find relevant or similar information for the semantic modelling of web services. In the INFRAWEBS framework the functionality of OM is used in the pre-filtering step, but another version is implemented using Apache Lucene (Apache’s text search engine library) as well.

3. Logical matching

The WSMO deliverable on discovery [11] defines semantic matching as:

$$W, G, O \models \exists x(g(x) \wedge ws(x))$$

where W is the definition of the web service, G is the definition of the goal, O is a set of ontologies to which both descriptions refer, g(x) and ws(x) are the first-order formulae describing the effects of the goal and web service respectively. In this case only the desired and offered outcomes are matched, and the meaning of the match is: there exists an outcome offered by the service which is requested by the user. This definition can be further elaborated into various types of matches:

- *Exact match*: the possible outcomes of the service and the goal are equivalent; the service does exactly what the user desires.
- *Subsumption match*: each possible outcome of the service is accepted by the goal, i.e. all service offers are acceptable by the user, though there might be desires not covered by the service.
- *Plugin match*: each possible outcome requested by the goal can be produced by the service, i.e. all user requests can be satisfied by the service, although the service may provide additional, non-matching outcomes as well.
- *Intersection match*: there is at least one service outcome accepted by the user (goal).

This section details the practical problems of semantic matching. Theoretically both Description Logic and Logic Programming (the most popular and available variations of logic in this area) are capable to find all the types of matches listed above. However, the assumption based on our real-life scenarios is that the majority of matches will be of type intersection.

A method for finding intersection matches using Description Logic is described in [18], but its application in case of long and complex capability descriptions is problematic. For example, it requires that unrelated concepts are defined as disjoint pair wise. Overall, the matching process needs complex DL modelling which is hard to maintain. Using Description Logic it is decidable whether there is a common solution for the goal and the service, but it cannot tell what the solution is. More details on our experiments with using Description Logic for service matching are published in [10].

Logic Programming can tell what the common solution is if it is able to find it. Let's take two very simple example conditions:

```
Goal:
  Ticket from Wien to Graz
  Ticket date 06.06.2006.

Webservice:
  Ticket from X to Y
     where X and Y are in Austria
  Ticket issued by Austrian Air
```

This example roughly corresponds to the WSMO Virtual Travel Agency use case [17], and also agrees with the INFRAWEBS project use cases [6].

For humans this seems a perfect match. If we model the outcome with a set of tickets according to the set-based modelling approach [11], it is clear that there can be tickets fulfilling all conditions. But in the world of logic, there is no implication in either direction between the goal and web service. The goal is specialized in details of desired service, while the web service is specialised in details of service delivery.

A further problem is that both goal and service descriptions may vary in the level of detail. For example, sometimes the destination or the date is omitted, sometimes more conditions are given, for example to have a business class ticket.

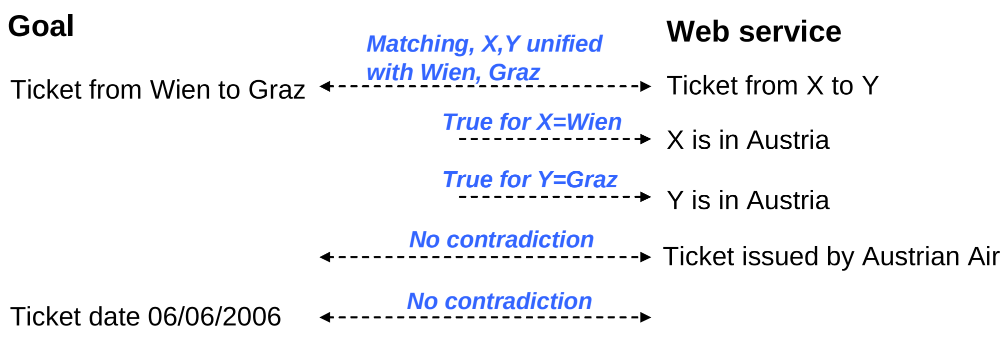

Figure 1: Schematic example of matching with unification

3.1. Semantic matching using Logic Programming

Our solution applies the unification facility of Prolog engines. If we find matching terms within the goal and the web service, we can use this information to decide on the matching. A schematic explanation of unification can be found in Figure 1. The goal in the figure describes a request for a ticket from Wien to Graz for a given date, while the service advertises Austrian Air flight tickets within Austria. The ticket departure and destination facts are unified in the goal and the web service, with the consequence that X becomes Wien and Y becomes Graz. After that, whether X and Y are in Austria can be decided. The last two facts have no correspondence on the other side, therefore they cannot generate any contradiction, and they can be silently ignored.

In order to reach a comparable list of terms some kind of normalized form is needed, of which the Disjunctive Normal Form (DNF) was the most suitable, as it represents the set of simple capabilities (desires or effects). The DNF consists of clause sets in the form of *(C_{11} and C_{12} and …) or… (C_{m1} and C_{m2} and …)*, where C_{ij} are atomic terms. We call C_{ij} a clause, and *(C_{11} and C_{12} and … C_{1n})* a clause set. A DNF is true if at least one clause set is true. A DNF clause set is true if all its clauses are true. This means that a clause set provides a complete solution if all its clauses are true.

The pre-processing steps needed to create the infrastructure for matching:

- Ontologies are converted to Prolog. Special predicates are used to represent subconcept, attribute and attribute type relationships within concepts.
  - o Web services are converted to DNF with the internal steps of:
  - o Replacing logical constructs such as implication or equivalence with an equivalent form using only conjunction, disjunction and negation,
  - o Conversion into Negation Normal Form (NNF),
  - o Elimination of *forall* and *exists* constructs, skolemization,

    - Moving disjunctions to the outermost level to reach DNF.

- DNF clause sets are converted to Prolog. Membership molecules of WSML are converted to *type/2* predicates and *hasValue* molecules are represented with *attr/3* predicates.

An example of generated Prolog clauses is given for a service providing flights from Innsbruck to Wien:

```
type(V_FlightPrefs,
flightBookingPreferences),
attr(V_FlightPrefs, start, V_Start),
=(V_Start, innsbruckAirport),
attr(V_FlightPrefs, end, V_End)
=(V_End, wienAirport),
attr(V_FlightPrefs, class, V_Class),
type(V_Buyer, buyer),
attr(V_Buyer, contactInformation,
V_BuyerContact),
attr(V_BuyerContact, emailaddress,
V_BuyerEmail),
type(V_BuyerEmail, string)
```

When the system is initialized, the following steps are needed for discovery with a given goal:

- The goal is converted first to DNF and then to Prolog the same way as web services, the result is a Prolog list of clause sets for each condition in the goal capability,
- The matching algorithm is run: an attempt is made to match each web service with the goal,
- The result is a list of matching services which is ranked before it is sent to the user.

As part of the matching algorithm, in order to match the postconditions of the web service and the goal we need to find a matching between the DNF clause sets representing postconditions. The steps to be performed for each pair of clause sets (one from the goal and one from the web service) are:

- Unification of clauses in the clause sets: two clauses are unified if they have the same signature, then corresponding variables in the two clauses are unified,
- Labelling each clause: matched, failed or ignored,
- Decision of match or failure,
- Generation of matching result (with lists of failed, ignored and matched facts attached).

The pseudo-code shows a simplified version of this matching algorithm:

```
% both inputs are lists of clauses
match(Goal, Webservice) :-
 % perform possible unifications
 unify(Goal, Webservice, UnifiedClauses),
 % classify each clause/term in goal as
 % matched/failed/ignored
 checkClauses(Goal, UnifiedClauses,
      MatchedInGoal, IgnoredInGoal,
      FailedInGoal),
 % classify each clause in service as
 % matched/failed/ignored
 checkClauses(Webservice, UnifiedClauses,
      MatchedInService, IgnoredInService,
      FailedInService),
 % decide on matching
 isMatching(MatchedInGoal, IgnoredInGoal,
            FailedInGoal),
 isMatching(MatchedInService,
      IgnoredInService, FailedInService).

unify([G|Goal],Webservice,M2) :-
  (if member(G, Webservice) then
                M2=[G|M1] else M2=M1),
  unify(Goal,Webservice,M1).

unify([],_,[]).

checkClauses([Clause|ClauseSet],
         UnifiedClauses,
         Matched2, Ignored2, Failed2) :-
 check(Clause, UnifiedClauses, Status),
 (if Status=m then
     Matched2=[Clause|Matched2] else
     Matched2=Matched1),
 (if Status=i then
     Ignored2=[Clause|Ignored2] else
     Ignored2=Ignored1),
 (if Status=f then
     Failed2=[Clause|Failed2] else
     Failed2=Failed1),
 checkClauses(ClauseSet, UnifiedClauses,
          Matched1, Ignored1, Failed1).

checkClauses([],_,[],[],[]).
```

The detailed explanation of the algorithm is given below. First, all possible unifications are made by the *unify* predicate. Second, the algorithm has to examine all clauses in the clause set with the effects of unifications made. If the clause yields true value after the unification, it is labelled as matched. If the clause is not true, but all its variables are bound, it is labelled as failed. False clauses with free variables mean that the condition they represent is not specified in the other clause set, therefore they are labelled as ignored.

Finally, the algorithm has to decide whether the goal and the web service match each other. If there is a failed clause, it means a disagreement of the goal and service, so there is no match. If there are only matched and ignored clauses, then a little heuristics is needed to decide about the match. As a primary rough heuristics we say there is a match if the number of matched clauses is greater than the number of ignored clauses.

In fact this is a problem of WSML capabilities: it is very hard to tell which variables represent essential, basic service issues and which variables are just to help describe conditions. Figure 1 shows two facts at the bottom which can be safely ignored. It can also happen that the service and the goal are totally different except they share the conditions of payment or confirmation (for example). In this case the matching result will not contain failed facts, just a lot of ignored clauses (the essential service requested) and a lot of matched clauses (the not so important buying conditions). If there are no failed facts, the decision of matching cannot be completely sure. Conventions are suggested to by-pass this shortcoming. One such agreement can be to define a basic service outcome or service class in each postcondition or effect.

The list of matching services should be ranked to provide guidance for the user in selection. The usual solution is to rank the list by some kind of quality aspect or by the categories of exact, subsume, plugin and intersection match. The algorithm introduced here provides the new possibility of ranking based on the number of ignored clauses. Clauses can be ignored on both the service and the goal side. Clauses ignored on the goal side means certain conditions are not defined in the service (e.g. departure date, comfort seats), so a lower number of ignored facts mean more precise fulfilment of user desires. Clauses ignored on the service side means something additional which is not specified in the goal (e.g. more travel information, company data, etc.).

The judgement of these is highly subjective, but a lower number of ignored facts may also mean a more accurate match here. In our implementation we rank by the sum of ignored facts in both service and goal. Another possible approach is to rank first by the number of ignored facts in the goal, and then by the number of ignored facts in the service.

Modelling user preferences and added value in discovery is an emerging new problem in this area. The presented matching algorithm can handle the added value in requests and offers as demonstrated in the following example. The goal postcondition declares that the flight is business class. If the service says nothing about the class, then this requirement is ignored by the matching algorithm. If the service offers only economy class, then the matching fails. If the service declares that it can provide both economy and business class tickets, then the corresponding fact is matched in the goal, and therefore the service gets a higher position in the ranked result list.

The presented algorithm can thus handle user preferences and provide a ranking based on it, if there is a way to differentiate between clauses modelling core postcondition and clauses modelling user preferences in the goal. Currently, we see the only solution to use axioms completely separated from the goal to express user preferences.

For matching preconditions and assumptions we can use a simpler matching mechanism as expressions have to be fully enforced there: a clause is either matched or failed. Facts from the goal are inserted into the knowledge base, and the clause sets (in DNF) of the service precondition and assumption are checked one-by-one. In this way not only the failure is detected but also the failing clauses can be identified and presented to the user.

3.2. Implementation and Performance

The matching algorithm was implemented in Prolog, and was tested with SWI-Prolog. The discovery engine is written in Java, which initializes the Prolog implementation, and feeds the WSML descriptions for ontologies and web services into it. Then, for each discovery request only the goal is converted into Prolog and the matching algorithm is run.

Further software components used in the implementation are wsmo4j for parsing WSML files and Interprolog for making Prolog queries from Java.

The matching algorithm has the following costs for each step (where goal has L clauses and service has M clauses):

- Unification of clauses: at most $L*M$ operations,
- Checking each clause: $L+M$ operations,
- Decision of match or failure: constant number of operations.

So the matching of one goal clause set with one service clause set takes approximately $(L+1)*(M+1)$ operations. If we take the natural assumption that the number of clauses has an upper limit m in the system (an upper limit for L and M), and the maximal number of clause sets representing a service or a goal has an upper limit c, we get the estimation that the order of complexity of the discovery algorithm is linear with respect to the number of services in the system.

The performance tests were based on the project use case; a frequent flyer system offering bonus flights, hotel stays and car rentals. The test environment contained 18 ontologies, 25 different web services and

10 different goals. In order to get more services for the tests, multiple copies were generated from each service.

The Prolog matcher on a 1.6 GHz P4 desktop PC provided the following response times:

250 services:	1.82s	(.00728s/ws)
1000 services:	7.47s	(.00747s/ws)
4000 services:	32.94s	(.00823s/ws)
8000 services:	65.63s	(.00820s/ws)

In the first experiment the number of services was 25, and the per service discovery time (the total time needed for discovery measured in the Java VM divided by the number of services checked for matching) varied between 12 and 103 milliseconds, depending on the complexity of the goal capability. In our next experiment the number of web services was increased to 250 in the same environment. The per service discovery time in the second experiment varied between 12 and 129 milliseconds.

Overall, the response time perceived by the user is between 3 and 25 seconds. This is acceptable for the project as the pre-filtering step is assumed to reduce the number of services in this second step of discovery to the magnitude of 2-300 services.

The following conclusions can be drawn from the response times:

- The native Prolog implementation needs only a couple of milliseconds to match a web service and a goal. This is scalable until the magnitude of 1000 services (result within 5 seconds), which is satisfying if we consider the pre-filtering step as well.
- There is a bottleneck in the communication between Java and Prolog. We will experiment with other possible communication methods.
- Little effort was spent on code optimization, and we think the speed of discovery can still be increased.

Currently there is little information available about the performance of other discovery implementations. We think the proposed and implemented solution is comparable in effectiveness and correctness to the few other approaches. The solution can be used generally, and it is also able to provide some explanation of the match or failure, which can be presented to the user.

In order to provide an interface for testing and experimenting, a web-based test bed has been implemented for the discovery component. This test bed includes not only the software but example data as well.

The test bed is available as an online service of SZTAKI, it can be reached from the demonstrations page of the INFRAWEBS project website: http://www.infrawebs.eu.

Related work

The early experiments to implement matching algorithms with Logic Programming are mostly done using Flora-2, a reasoner supporting F-logic [19][20][21]. The syntax of the Flora-2 language is very close to WSML, which alleviates language conversion.

In [19] and [20] the matching is based on the discovery model in Transaction Logic [11]. After precondition matching, the service postcondition is assumed to be true (inserted into the knowledge base), and the fulfilment of goal postcondition is checked. Finally, the inserted facts are retracted from the knowledge base, and the matching of next service can be started. The matching is done in the traditional way, checking the truth value of the goal postcondition. This leads us to the matching problem detailed in section 3, namely, the postcondition of a matching service does not imply the goal postcondition. This solution works if goals are prepared by parameterizing goal templates, but it does not support custom goals and the discovery functionality needed for service composition.

There are several proof-of-concept experiments for SWS discovery using WSMO, but relatively few complete implementations. Della Valle et al. have created a complete discovery solution based on mediators [21]. It is used in Glue, a lightweight implementation of their suggested execution framework for Semantic Web Services. Glue uses pre-defined goals (goal templates), which can be parameterized by the user for her actual request. A wgMediator [14] is used to find matching web services for a specific goal. Therefore, the (simplified) steps of discovery are:

- Parameterize a pre-defined goal,
- Find the wgMediator for that pre-defined goal,
- The wgMediator returns matching services for the goal.

The advantages of this approach are:

- Efficient solution which is also easy to implement,
- Straightforward support for mediation between heterogeneous ontologies using ooMediators and ggMediators.

The disadvantages of this approach are:

- Goals can only be created based on pre-defined goals. If the user's desire does not match any of the pre-defined goals, she has no choice to find matching services,
- The maintenance of pre-defined goals needs continuous effort from the operators of the discovery engine. If composite goals are supported, the number of pre-defined composite goals to be created and maintained may reach a huge number.

OWL-S is probably the most known approach currently for Semantic Web Services. In OWL-S a service can be specified with inputs and outputs, but also with pre- and postconditions. The typical discovery solutions using OWL-S [22][23] match the user's input with the service input and the service output with the user's desired outcome. This means that all concepts required by the service as input need to match with a concept provided as user input, and all concepts required by the user as output match with a concept in the service output. The schematic process of such discovery contains the following steps:

- Locate services with matching input and output
    - For each input concept of the service, find a matching input concept provided by the user
    - For each output concept required by the user, find a matching output concept of the service

The advantages of this approach are:

- Simple matching algorithm, required reasoning is minimal
- Fast implementations are possible using indexed search

The disadvantage of this approach is that matching is based solely on inputs and outputs, and the effects of service execution are ignored, so there is no guarantee that the matching service has the desired effect. (As an example: a service with an input of type document and an output of type document might perform rather different tasks on the document.)

In [24] a peer-to-peer environment is described for WSMO-based semantic discovery. The system (supported by the DIP project) contains several novelties, such as decentralized discovery and QoS-based matching. Here we concentrate on the method of discovery. The paper provides minimal information about the logic-based matching applied in the system. Logic-based matching is preceded by a filtering step, when irrelevant services are filtered out, and the logic-based matching is only applied on a small set of services. The rationale is similar to that of the INFRAWEBS project: logic-based matching is expensive, and it is needless to run the expensive algorithms on services which have no chance to match with the goal. The concepts are classified into concept groups, and services and goals are associated with concept groups based on concepts mentioned in the capability. Keys are calculated for both services and goals which summarize the concept groups used in the capability descriptions.

According to the authors the necessary condition for a service to match with a goal is that its description at least must contain all concepts related to those specified in the goal. We saw in the previous sections that this assumption for matching is not necessarily true. For example, a goal may contain date constraints while services usually say nothing about dates. The creators of the system admit that for partial matchmaking some concepts has to be discarded from the goal in the filtering phase, and GUI tools would help users in doing this. As a comparison of this approach and INFRAWEBS discovery we outline the following:

- Both projects use a filtering phase before the logic-based matching, but the filtering rules used in INFRAWEBS are less restrictive.
- QoS data is used to define criteria for matching in this work, while INFRAWEBS uses QoS data for ranking the list of matching services.

Conclusion

In this paper we have presented the implementation of the Discovery Component of the SWS environment created by the INFRAWEBS project.

The matching algorithm selected for discovery combines a traditional IR-based pre-filtering step and a logic-based matching implemented in Prolog. The logic-based step of discovery uses a novel technique based on Prolog-style unification of terms. This approach is able to find intersection and other types of service matches, and also provides possibilities to compare, rank and explain service matches (or non-matches). The performance of the solution is satisfactory within the project scenario. In the lack of performance details of other solutions it is hard to compare the presented implementation with other discovery engines.

A test bed is available online for the demonstration of the approach. It can be reached from the demonstrations page of the INFRAWEBS project website.

Acknowledgement The presented work was supported by the INFRAWEBS project funded by the European Commission.

References

[1] Alejandro López, Jesús Gorroñogoitia. Deliverable D11.1.2 INFRAWEBS Software Development Process.
[2] Alejandro López, Jesús Gorroñogoitia. DS2 Activity definition. INFRAWEBS Document.
[3] Alejandro López. DD2 Component interface description. INFRAWEBS Document.
[4] Chris Preist. A conceptual architecture for semantic web services. In Proceedings of the International Semantic Web Conference 2004 (ISWC 2004), November 2004.
[5] Web Services Glossary. W3C Working Group Note 11 February 2004. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/
[6] O. López, E. Riceputi, A. López, C. Pezuela, Y. Gorroñogoitia. INFRAWEBS Deliverable 10.1-2-3-4.1 Requirement Profile 1 & Knowledge Objects. 2005 Nov 10.
[7] Ettore Riceputi. Deliverable D10.5-6-7.2 Requirement Profile 2 & Knowledge Objects
[8] J. Scicluna, T. Haselwanter, A. Polleres. INFRAWEBS Deliverable D7.1-2.1 Reduced Rule Base, QoS Metrics, Running SWS-E and QoSBroker. 2005 Aug 15
[9] Cs. Fülöp, L. Kovács, A. Micsik, Z. Tóth, G. Agre, A. Polleres. INFRAWEBS Deliverable D6.1-2.1: Specification & Realisation User-Interface Agent, Discovery Agent. Nov 2005.
[10] A. Micsik, L. Kovács, Z. Tóth, P. Pallinger, J. Scicluna. INFRAWEBS Deliverable D6.2.2 – Specification & Realisation of the Discovery Component. Aug 2006.
[11] D5.1v0.1 WSMO Web Service Discovery. WSML Working Draft 12 11 2004, http://www.wsmo.org/2004/d5/d5.1/v0.1/20041112/
[12] Ruben Lara: KnowledgeWeb Deliverable 2.4.2 Semantics for Web Service Discovery and Composition.
[13] A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, Logics for Databases and Information Systems, chapter 5, pages 117-166. Kluwer Academic Publishers, March 1998.
[14] D16.1v0.3 The Web Service Modeling Language WSML. WSML Working Draft 5 October 2005, http://www.wsmo.org/TR/d16/d16.1/
[15] Web Service Modeling Ontology Primer. W3C Member Submission 3 June 2005, http://www.w3.org/Submission/WSMO-primer/
[16] WSMX Deliverable D10 v0.2, Semantic Web Service Discovery. WSMX Working Draft October 3, 2005
[17] D3.3 v0.1 WSMO Use Case "Virtual Travel Agency". WSMO Working Draft 19 November 2004, http://www.wsmo.org/2004/d3/d3.3/v0.1/
[18] Lei Li and Ian Horrocks. A software framework for matchmaking based on semantic web technology. In Proceedings of the 12th International Conference on the World Wide Web, Budapest, Hungary, May 2003.
[19] Michael Kifer, Ruben Lara, Axel Polleres, Chang Zhao, Uwe Keller, Holger Lausen and Dieter Fensel. A Logical Framework for Web Service Discovery. Proceedings of the ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications, Hiroshima, Japan, November 8, 2004. CEUR Workshop Proceedings, ISSN 1613-0073, Vol-119.
[20] WSML Deliverable D5.1 v0.1 Inferencing Support for Semantic Web Services: Proof Obligations. WSML Working Draft – August 2, 2004
[21] Emanuele Della Valle, Dario Cerizza, Irene Celino. The mediators centric approach to Automatic Web Service Discovery of Glue. First International Workshop on Mediation in Semantic Web Services: MEDIATE 2005, Amsterdam, Netherlands, December 12 2005
[22] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, Katia Sycara. Semantic Matching of Web Services Capabilities. International Semantic Web Conference 2002. In: Lecture Notes in Computer Science, Volume 2342, Jan 2002, Page 333
[23] Michael C. Jaeger, Gregor Rojec-Goldmann, Gero Mühl, Christoph Liebetruth, Kurt Geihs. Ranked Matching for Service Descriptions using OWL-S. In Kommunikation in verteilten Systemen (KiVS 2005), Informatik Aktuell, Kaiserslautern, Germany, February 2005. Springer.
[24] Le-Hung Vu, Manfred Hauswirth, Fabio Porto, Karl Aberer. A Search Engine for QoS-enabled Discovery of Semantic Web Services. International Journal of Business Process Integration and Management, 2006, to be published.