Noname manuscript No.
(will be inserted by the editor)

Matching Goals and Semantic Web Services in FLORA-2: A Logical Inference Based Discovery Agent

Omid Sharifi · **Zeki Bayram**

Received: date / Accepted: date

Abstract Matching web services and client requirements in the form of goals is a significant challenge in the discovery of semantic web services. The most common but unsatisfactory approach to matching is set-based, where both the client and web service declare what objects they require, and what objects they can provide. Matching then becomes the simple task of comparing sets of objects. This approach is inadequate because it says nothing about the functionality required by the client, or the functionality provided by the web service. As a viable alternative to the set-based approach, we use the F-Logic language as implemented in the Flora-2 logic system to specify web service capabilities and client requirements in the form of logic statements, clearly define what a match means in terms of logical inference, and implement a logic based discovery agent using the Flora-2 system. The result is a practical, fully implemented matching engine based purely on logical inference for web service discovery, with direct applicability to Web Service Modeling Ontology (WSMO) and Web Service Modeling Language (WSML), since F-Logic is intimately related to both.

Keywords Semantic Web Service Matching · Discovery · Intelligent Agent · F-Logic · Logical Inference

1 Introduction

Given a semantically rich enough description of web service capabilities and client requirements, semantic web service discovery tries to determine which web service

O. Sharifi
Department of Computer Engineering, Eastern Mediterranean University,
Famagusta, Northern Cyprus, via Mersin, Turkey
Tel.: +90-392-6301484
Fax: +90-392-6350711
E-mail: omid.sharifi@emu.edu.tr

Z. Bayram
E-mail: zeki.bayram@emu.edu.tr

is in a position to satisfy best the requirements of the client. Matching is the main operation performed during the discovery process, and takes two parameters: a formal description of what the requester desires, and a formal description of what a web service provides as a service. Its job is to decide if the web service can satisfy the requirements of the requester. At the same time, the web service may have some preconditions before it can be called, so the matching operation must also check whether the client and/or state of the world can satisfy these preconditions.

Among the existing several semantic web service frameworks, we take WSMO [9] as our starting point in our discussion of semantic web service discovery, as it allows us to formally distinguish between user requests (called "goals" is WSMO terminology) and web service specifications, and use the same formalism for specifying both. WSMO itself is based on the Web Service Modeling Frame-work (WSMF) [13] and has four main components, which are *Ontologies*, *Web Services*, *Goals* and *Mediators*.

Web Service Modeling Language (WSML) [6] is a language used to describe ontologies, semantic web services and goals in conformance to the WSMO framework. It has a solid logic foundation, namely F-Logic [2]- a powerful logic language with object modeling capabilities. WSML consists of five variants, which are *WSML-Core*, *WSML-DL*, *WSML-Flight*, *WSML-Rule* and *WSML-Full*. Each language variant provides different levels of logical expressiveness, as explained in detail in [6].

There are several discovery approaches used in WSML [32]. Keyword-based discovery is based on simple syntactic matching of goals and web services at the non-functional level. Set-based "lightweight" discovery works over simple semantic descriptions that takes into account the *postcondition* and *effect* of goals and web services. Set-based "heavyweight" discovery works over richer semantic descriptions by taking into account *precondition*, *assumption*, *postcondition* and *effect*, and the relationships between them. Heavyweight discovery based on WSML-Flight uses query containment reasoning tasks (where the results of one query are always guaranteed to be a subset of some other query) for the matchmaking. Set based heavyweight approach based on WSML first order logic (FOL) uses a theorem-prover for matchmaking [1].

Different kinds of *match* have been defined for the the set based approach. These are summarized below [32].

- *Exact-Match* happens when the items delivered by the web service $\mathcal{W}$ match perfectly the items specified in the goal $\mathcal{G}$. No irrelevant objects are returned by the service.
- *Subsumption-Match* happens when items returned by the service $\mathcal{W}$ is a subset of the objects requested in the goal $\mathcal{G}$.
- *Plugin-Match* happens when items delivered by the service $\mathcal{W}$ is a superset of the objects requested in the goal $\mathcal{G}$.
- *Intersection-Match* happens when the delivered items of the service $\mathcal{W}$ has a nonempty intersection with the set of relevant objects for the requester as specied in the goal $\mathcal{G}$.

The problem with the set based approach, no matter what underlying logic is used to represent sets of objects, is that even if the set of objects requested in the goal

are indeed returned by the service, there is no guarantee that the desired operation has been performed to obtain theses objects. Furthermore, what may be required by the execution of a service may not be some new objects, but rather a change in the relationship status of some already existing objects. Alternatively, some new objects may be desired by the goal, provided that certain relationships exist among these objects. We should also not overlook the fact that before the web service can be called, it is usually not sufficient to only have certain parameters supplied to the service: some other conditions may need to be true also, before the web service can be reliably called.

It is clear that the pure set based approach cannot answer these needs. What is needed is at least a first-order logic-based formalism and methodology that utilizes some form of inference. To perform a match between a request in the form of a goal and web service specifications, logical entailment should be carried out to determine whether the web service has all its requirements met before being called, and whether the service provided by the web service will satisfy the needs of the client, while maintaining the relationships between input ant output objects as required by the client. The choice of logical formalism for specifying goals and web services, on the other hand, must (i) allow relatively uncomplicated specification of web services and goals (ii) permit efficient execution of inference engines during the matching process, and (iii) be powerful enough to be effectively specify goals and web service capabilities.

Our work reported here achieves all these goals. We can summarize our contribution as follows:

1. We specify a sub-language of F-logic with implicit existential and universal quantifiers (depending on where the formula is used) that permits efficient goal-directed deduction, as in the case of Logic programming,
2. We clearly state the proof commitments (in terms of logical entailment) necessary for a successful match, and finally
3. We implement a logical entailment based (intelligent) matching agent using the FLORA-2 system, demonstrating not only the feasibility of our approach, but also its practicality.

We chose F-Logic [19] as the implementation language of our matching agent, since it is the underlying logical basis of WSML, and our semantic descriptions of ontologies, goals and web service capabilities can be easily translated into the syntax of WSML in a straightforward manner. In our implementation of the matching agent, we made use of the reification capability and transactional knowledge base update feature of Flora-2. Using F-Logic and its Flora-2 implementation allowed us to leverage the underlying inference capability of Flora-2, and to concentrate on the higher-level inference tasks that are relevant to matching, rather than the chores of implementing an inference engine from scratch.

Since WSML is based directly on F-Logic [19], our implementation is a showcase for how WSML web services and goals can be represented in Flora-2, and how the Flora-2 system can be used to perform logical entailment based matching between goals and web services.

The remainder of the paper is organized as follows. Our sublanguage of F-Logic used to describe ontologies, goals and web services, and the matchmaker intelligent agent architecture is described in section 2. The implementation of the matchmaker agent in Flora-2 is given in section 3. In section 4 we have a sample ontology, goal and web service specification that the matchmaker agent can use as inputs. Section 5 discusses the various problems that were encountered and how they were solved in the implementation. Section 6 describes several other matching approaches reported in literature, and compares them with our approach. Section 7 is the conclusion and future research directions. Finally, the appendix contains the our sublanguage of F-Logic to describe our implementation.

2 The Intelligent Semantic Web Service Matchmaker Agent

2.1 Logical Components of Web Service and Goal Specifications

In our model of web service and goal specification in F-logic, the desired computation by a requester can be specified in the form of inputs and relations on these inputs ($goal.pre$), as well as outputs and relations on these outputs ($goal.post$). The inputs and outputs may be required to be in a certain relationship as well after the computation. Furthermore, the requester may desire a new state of the world ($goal.effect$) once the execution of a web service is completed. On the service provider side, the capability provided by a web service is specified in the form of preconditions ($ws.pre$), which must be true before the webs service can be called, postconditions ($ws.post$), which the web service guarantees will be true once its execution is completed, assumptions about the the state of the world before the web service can be reliably called ($ws.assumption$), as well as the changes to the state of the world that are guaranteed to be in existence when the web service completes its execution ($ws.effect$). There is also the state of the world before the web service is called ($worldBefore$). Lastly, we assume the existence of a *common ontology* (co) that contains definitions of concepts, constraints and logic rules that can be used in the goal or web service description, as well as *local ontologies* of each goal and web service, containing definitions and logic rules (which we shall denote by $goal.ont$ and $ws.ont$) that are local to the goal or web service. We can safely assume there is no naming conflict between local ontologies and other ontologies, since each ontology can be assigned a unique namespace in the web environment.

2.2 Sub-language of F-logic used for Specification of Goals, Web Services and Ontologies

Although it would be desirable to use a specification language that is as powerful as possible, one has to be careful about (i) efficient implementability of logical inference needed to verify the validity of the proof commitments derived from the specifications, and (ii) understandability of the specifications. In the case of first order logic, unrestricted first order formulas can easily become very complicated, hard to understand, verify and test. Furthermore, *efficiently* proving logical entailment in full first

order logic (when a proof exists), even when using the resolution principle [14] or some of its derivatives, is not an easy proposition. What is needed is a compromise: a "clean" and concise subset of first order formulas that are easily understood, and can efficiently be used in determining the validity of certain implications that will guarantee a satisfactory match, but is at the same time expressive enough in the context of web service discovery.

We use a sub-language of F-logic in our web service and goal specifications. Formulas used for $goal.pre$ and $ws.post$ are implicitly universally quantified conjunctions of positive molecules and predicates. Formulas in $ws.pre$ and $goal.post$ on the other hand are implicitly existentially quantified logic statements that can involve conjunction, disjunction and negation connectives. In our implementation of matching, this allows us to verify the proof commitments (given in the next section) by temporarily by inserting antecedents of implications into the logic system database (as well as the common ontology and local ontologies), and use the Flora-2 logic system itself to prove the consequent, effectively establishing the logical entailment relation between the two.

Ontologies can contain any facts and rules that are allowed in F-Logic. Furthermore, using the reserved predicates $constraints$ and $constraint$ one can specify constraints. The matchmaker agent verifies that no constraints are violated in any stage of the matching process.

Our choice of sub-language allows us to have a powerful, yet practical logic based matching algorithm and strikes a fine balance between expressiveness and efficient implementation. The proof commitments are verified with the same kind of efficiency that logic programs are executed in a logic programming framework.

2.3 Syntax of the sub-language of F-logic used for Specification of Goals, Web Services and Ontologies

In figure 1 we have the Extended Backus Normal Form (EBNF) grammar for the sublanguage of F-logic we used for specifying goals, web services and ontologies. This grammar should be considered in conjunction with the full grammar of Flora-2 given in [25]. In Flora-2 "Rule" defines a rule with or without a right hand side. Left hand sides of rules are either inheritance relationships, membership relationships, or object declarations. "Body" stands for a rule body. No that we allow any valid Flora-2 rule in the ontology, since Flora-2 is used to represent knowledge that is common to goals and web services. A *constraint* rule belongs to our sublanguage, and should have on its left side the predicate "constraint" and the id the constraint as its parameter. Constraints are verified at various stages in the matchmaking process. A "Fact" is either a predicate or an object with an id that is an anonymous variable. The nonterminal "Term" is used to define both predicates and terms. Finally, an "atom" is any symbol without any internal structure.

```
Goal := GoalID '[inputs->${startG[pre->${' FactList '}' ','
        'post->{' Query '}' ']' '}' ']' '.'

WebService := WebserviceID '[inputs->${startW[pre->{' Query '}' ','
              'post->${' FactList '}' ']' '}' ']' '.'

Ontology := (Rule | ConstraintRule)*

ConstraintRule := 'constraint' '(' ConstraintID ')' ':-' Body '.'

FactList :=  Fact (',' Fact)*

Fact := '${' ObjectSpecification2 '}' | Term

ObjectSpecification2 := '?_' '[' SpecBody ']'

Query := Fact | QueryNegative | QueryConjunct | QueryDisjunct

QueryNegative := 'not' '(' Query ')'

QueryConjunct := 'and' '(' Query ',' Query ')'

QueryDisjunct :=  'or' '(' Query ',' Query ')'

ConstraintID := atom

GoalID := atom

WebserviceID := atom
```

Fig. 1 EBNF grammar for the sublanguage of F-logic used for specifying goals, web services and ontologies

2.4 Proof Commitments that must be Checked for Validity before a Match is Successful

The proof commitments or obligations (i.e. what must be proven before a match can succeed) required for our logical inference based matching are as follows:

1. $co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre \models ws.pre$: The precondition of the web service should be logically entailed by the common ontology, local ontologies of the goal and web service, and what is provided/guaranteed by the goal (i.e. $goal.pre$).
2. $co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre \bigwedge (ws.pre \Rightarrow ws.post) \models goal.post$: If the conditions for the web service call are satisfied, then the requirements of the goal should be satisfied. Note how we assume that the execution of the web service guarantees the validity of the implication $ws.pre \Rightarrow ws.post$.
3. $worldBefore \models ws.assumption$: The assumptions the web service makes about the world must be true before it can be called. These assumptions are independent of what the client supplies to the web service (e.g. flights canceled due to weather conditions etc.)

4. $worldBefore \bigwedge (ws.assumption \Rightarrow ws.effect) \models goal.effect$: The state of the world after the web service is called, as required by the goal, should be guaranteed. Again note how we assume that the web service execution guarantees the validity of the implication $ws.assumption \Rightarrow ws.effect$.

In the WSMO framework, assumptions and effects need not be checked, since they are supposed to represent real world conditions. However, even if they represent real world conditions, there is no reason why real world conditions could not have some internal representation in the computer domain that reflects the state of the world. We thus assume that the state of the world is represented in some externally accessible, global knowledge base. Then we can interpret "effects" as changes to the state of this global knowledge base. Assumptions can also be checked against this global knowledge base. Preconditions/postconditions of a web service or goal, on the other hand, refer to relationships among objects interchanged between the goal and web services.

2.5 Dealing with State Change and Non-monotonicity: Simulating Non-monotonicity inside First-order logic

Of special interest is the way the computation of a web service is treated above: as an implication from its precondition to its postcondition ($ws.pre \Rightarrow ws.post$), and an implication from its assumption to its effect ($ws.assumption \Rightarrow ws.effect$). In the presence of state change caused by the execution of the web service, however, this can be problematic. What was true in the precondition of the web service may suddenly become false after the webservice is executed due to the state change caused by the execution itself. So the entailments given in the proof commitments above should *not* be proven using inference rules for first order logic alone (F-logic can be mapped to first-order logic and is thus equivalent to it [18]) *if* they involve change of the internal state of objects, *or if* some fact known to be true (false) before the web service is called becomes false (true) after the web service is called.

While it is true that we cannot directly reason about non-monotonic changes in the state of the world, or internal states of objects, using first order logic alone, and that more specialized logics, such as transaction logic [4] or temporal logic [22], are required to reason with non-monotonic changes caused by the execution of the web service, we can get around this problem by writing our specifications that use only monotonic changes to the state, but *simulate* non-monotonicity. For state changes inside an object, we can return information about *scheduled* activity regarding changes to the object. For example, the postcondition of the web service can contain a predicate $scheduledUpdate(?x,4)$ instead of $?x = 4$ for some variable that was supposed to have some other value in the precondition. The goal postcondition then can query the $scheduledUpdate(?x,4)$ predicate. Similarly, an object can be marked for deletion, e.g. $scheduledDelete(someObject)$ may be part of the post condition instead of the object actually being deleted. After all, in the web service discovery phase, we are not concerned about real changes actually happening, but rather the knowledge that such changes will take place if the web service is called.

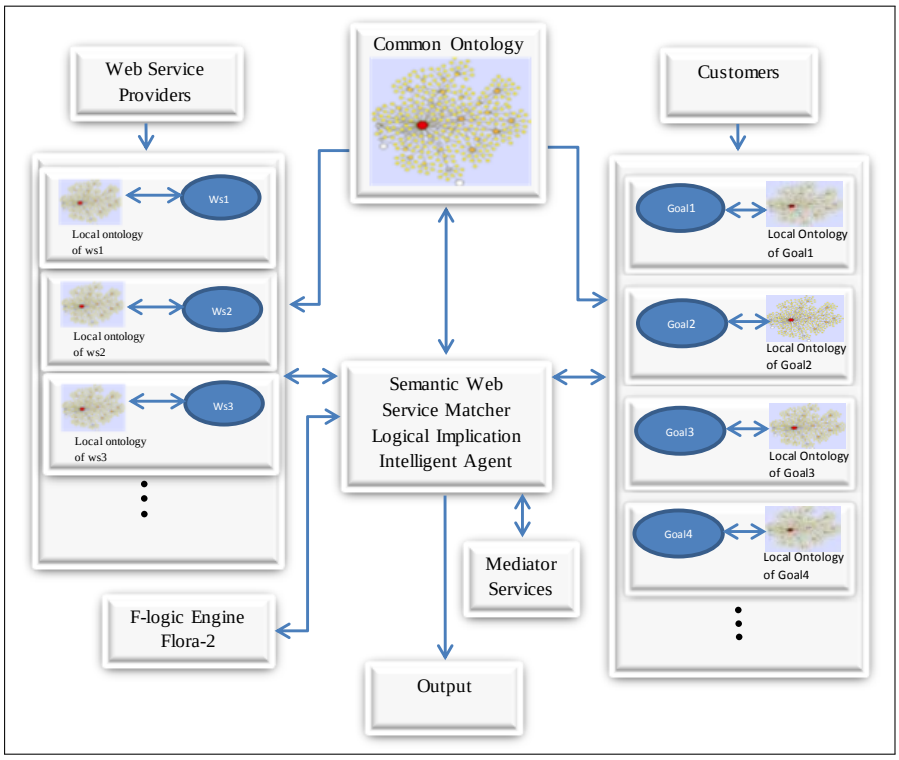

Fig. 2 The proposed semantic web service matchmaker intelligent agent architecture

2.6 The Intelligent Matchmaker Agent Architecture

Figure 2 depicts the architecture of our intelligent matchmaker agent. At the center of the figure sits the agent itself. It has access to web service specifications, goal specifications, the common ontology, and mediation services. Since it is implemented in Flora-2, it has access to all the underlying functionality of the Flora-2 engine as well. The agent verifies proof commitments one by one for each goal-web service pair, keeping track of both the successful matches and the unsuccessful ones, and reports the result at the end. There is no "degree of match" computed, since we are interested only in web services that satisfy the requirements of a goal completely.

"Mediation" is a broad term used to describe transformations that ensure compatibility among components of a system. At the simplest level, mediation can be carried out between different terminologies, so that equvalencies between terms are established (e.g. "car" is the same thing as "automobile" etc.). Although our architecture includes a mediation component, in our actual implementation we concentrated on the logical reasoning part, and left the mediation part out altogether. The mediation component can be incorporated into the implementation later on in a straight-forward manner.

3 Implementation of the Matchmaker Agent in Flora-2

3.1 Overview of Flora-2

The proposed intelligent agent for semantic web service matching uses the Flora-2 reasoning engine. Flora-2 is considered as a comprehensive object-based knowledge representation and reasoning platform. The implementation of Flora-2 is based on a set of run-time libraries and a compiler to translate a unified language of F-logic [9], HiLog [4], and Transaction Logic [5,4] into tabled Prolog code [25]. Basically, Flora-2 supports a programing language that is a dialect of F-logic including numerous extensions, that involves a natural way to do meta-programming in the style of HiLog, logical updates in the style of Transaction Logic, and a form of defeasible reasoning described in [33]. Flora-2 provides strong support for modular software development through its unique feature of dynamic modules. Some important extensions, such as the versatile syntax of Florid path expressions, are borrowed from Florid [24].

3.2 How We Use Flora-2

Our implementation of the semantic web service matchmaker makes use of many of the unique features of Flora-2. Goals and web service specifications are represented as objects with reified internal parts. Modules and the transactional logic capabilities of Flora-2 are used to temporarily insert facts into the database in an isolated environment and verify the proof commitments for each goal-web service pair. Higher order features are used for verifying the proof commitments.

In the following sections, we give the implementation for our matcher in Flora-2. The program is instructive, and its relative conciseness allows us to present it in full in the main body of the paper. Note that the program does not implement the proof commitments regarding the assumptions and effects of web services, but they are similar in essence to implemented proof commitments for preconditions and postconditions, and extending our program to deal with them is straightforward.

3.3 The Top-level Matcher Loop

Figure 3 has predicates that contain information about the files containing goals, web services and the common ontology. Due to a syntactic glitch in the implemetnation of Flora-2, the *cut (!)* operator is not allowed directly after the implication $(:-)$ operator, so we had to use the *true* predicate before the *cut* operator. Each *Webservice* predicate contains information about a specific web service, a *Goal* predicate contains information about a specific goal, and $CommonOntolgy$ contains information about the common ontology. The predicates *WebServices* and *Goals* describe which goals and web services should be included in the match operation.

Note that unlike Prolog, in Flora-2 predicates *can* start with capital letters, and variables *must* start with the question mark (?).

Figure 4 contains the top-level predicates of the matcher. *run* retrieves the names of web services and goals, and returns a list which has information about which web

```
1
2  WebService(ws1, 'C:\Users/OMID/Desktop/test/ws1', W1):-true,!.
3  WebService(ws2, 'C:\Users/OMID/Desktop/test/ws2', W2):-true,!.
4  WebService(ws3, 'C:\Users/OMID/Desktop/test/ws3', W3):-true,!.
5
6  Goal(goal1,'C:\Users/OMID/Desktop/test/goal1'):-true,!.
7  Goal(goal2,'C:\Users/OMID/Desktop/test/goal2'):-true,!.
8  Goal(goal3,'C:\Users/OMID/Desktop/test/goal3'):-true,!.
9  Goal(goal4,'C:\Users/OMID/Desktop/test/goal4'):-true,!.
10
11 CommonOntology('C:\Users/OMID/Desktop/test/CommonOntology'):-true,!.
12
13 WebServices([ws1, ws2, ws3]).
14 Goals([goal1, goal2, goal3, goal4]).
```

Fig. 3 Facts about goals, web services and common ontology

service matches which goal. *run*2 pairs goals and web services, and checks by calling *main* whether a goal matches a web service. *main* takes a goal name and a web service name, loads the goal and web service data, local ontologies of the web service and goal (they are in the same file as the web service or goal respectively), as well as the common ontology data into a module that is unique to the current web service, and calls *matchh* which does the proof commitment verification for the goal and web service.

In lines 24 and 37 of Figure 4, the current module is forced to empty its contents by loading a file called "empty" (which naturally contains nothing inside) into it, so that the same module can be used to test whether some other goal matches the web service that "owns" the module.

3.4 Proving the Commitments for a Successful Match

In Figure 5 we have the code that tries to prove the commitments specified in section 2. The predicate *matchh* retrieves the precondition of goal and inserts it temporarily into the module that was given to it. Then *prove* is called, which verifies the proof commitments. Specifically, in line 6, the proof commitment $[co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre \models ws.pre]$ is verified, and in line 8, the proof commitment $[co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre \bigwedge (ws.pre \Rightarrow ws.post) \models goal.post]$ is verified.

Note that the predicate *check_constraints* is called at various points where actions can lead to violation of constraints.

The *prove*/1 predicate takes only one parameter, i.e. only the precondition of a web service, and tries to prove it in the module associated with the web service. The assumptions of the proof commitment $[co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre \models ws.pre]$ (i.e. $co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre$) have already been temporarily inserted into the module, and all that is left is to verify the proof commitment $ws.pre$.

We could not use the exact same approach for the second proof commitment $[co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre \bigwedge (ws.pre \Rightarrow ws.post) \models goal.post]$, however, since Flora-2 does not allow temporary insertion of rules (of the form $lhs :- rhs$) in a

```
1
2  run(?Result):-
3      WebServices(?WSS),
4      Goals(?GS),
5      run2(?GS, ?WSS, ?Result).
6
7  run2([],?_,[]):-
8      true,!. // goals finished
9
10 run2([?Goal|?RestGoals],[], ?Result):-
11     true,!,
12     WebServices(?WSS),
13     run2(?RestGoals,?WSS,?Result).
14
15 run2([?Goal|?RestGoals],[?WS|?RestWebServices],
16      [[?Goal,' MATCHES ', ?WS]|?RestResult]):-
17         main(?Goal,?WS), !,
18         run2([?Goal|?RestGoals],?RestWebServices, ?RestResult).
19
20 run2([?Goal|?RestGoals],[?WS|?RestWebServices],
21      [[?Goal,' DOES NOT MATCH ', ?WS]|?RestResult]):-
22         true,!,
23         WebService(?WS, ?, ?WsModule),
24         ['C:\Users/OMID/Desktop/test/empty'>>?WsModule],
25         run2([?Goal|?RestGoals],?RestWebServices, ?RestResult).
26
27 main(?GoalName,?WsName):-
28   WebService(?WsName, ?WsPath, ?WsModule),
29   Goal(?GoalName, ?GoalPath),
30   CommonOntology(?OntologyPath),
31   [+'C:\Users/OMID/Desktop/test/matcher'>>?WsModule],
32   [+?WsPath>>?WsModule],
33   [+?OntologyPath>>?WsModule],
34   [+?GoalPath>>?WsModule],
35   matchh(?GoalName, ?WsName, ?WsModule)@?WsModule,
36   ?FoundWs=?WsName,
37   ['C:\Users/OMID/Desktop/test/empty'>>?WsModule],!.
```

Fig. 4 Matching agent top-level predicates.

module. What we needed was the ability to insert into the module $ws.post :- ws.pre$ ($ws.post$ is a set-valued attribute, and logical variables get instantiated to each member of the set one at a time according to the semantics of Flora-2). To get around this restriction, we devised the $prove/3$ predicate which takes three parameters (postcondition of a goal, postcondition of a web service, and precondition of the same web service). In lines 16-17 of Figure 5, the objective of using the implication ($ws.pre \Rightarrow ws.post$) is achieved operationally by unifying the postcondition of the goal with the postcondition of the web service, and then proving the precondition of the web service, thereby implementing backward reasoning manually, and achieving the same result as if $ws.post :- ws.pre$ was inserted into the module.

```
1  matchh(?Goal, ?WebService, ?WsModule):-
2      check_constraints,
3      ?WebService[inputs->${startW[pre->?WsPre, post->?WsPost]}],
4      ?Goal[inputs->${startG[pre->?GoalPre, post->?GoalPost]}],
5      t_insert{?GoalPre}@?WsModule,
6      prove(?WsPre),
7      check_constraints,
8      prove(?GoalPost, ?WsPost, ?WsPre),
9      check_constraints.
10
11 prove(and(?X,?Y)):- !, prove(?X), prove(?Y).
12 prove(or(?X,?Y)):- !,  prove(?X); prove(?Y).
13 prove(not(?X)):-  !, naf(prove(?X)).
14 prove( ?X):- ?X@?WsModule.
15
16 prove(?GoalPost, ?WsPost, ?WsPre):-
17     ?GoalPost = ?WsPost,!, prove(?WsPre).
18
19 prove(and(?X,?Y), ?WsPost, ?WsPre):- !,
20   prove(?X, ?WsPost, ?WsPre), prove(?Y, ?WsPost, ?WsPre).
21
22 prove(or(?X,?Y), ?WsPost, ?WsPre):- !,
23     prove(?X, ?WsPost, ?WsPre); prove(?Y, ?WsPost, ?WsPre).
24
25 prove(not(?X), ?WsPost, ?WsPre):-  !,
26     naf(prove(?X,?WsPost, ?WsPre)).
27
28 prove( ?X, ?WsPost, ?WsPre):- ?X@?WsModule.
```

Fig. 5 Proving the commitments for a match

3.5 Checking Constraints

Figure 6 contains the code for checking constraint violations. When a constraint in an ontology (whether local or common) is violated, an error message is printed, although no action is currently taken to invalidate the match. Furthermore, successful constraint checks are also shown in the output.

This behavior can easily be changed and a match made to fail in case of a constraint violation by removing the clause in lines 22-24.

4 Specifying Web Services, Goals and Ontologies in Flora-2

In this section, we give a representative example of (i) a web service specification for making appointments in hospitals, (ii) a goal for consume the appointment service, and (iii) the common ontology used by the web service and goal.

```
check_constraints:-
    constraints(?C),
    check_constraints(?C,?R),
    verify_results(?R).

check_constraints([],[]).

check_constraints([?H|?T],[?F|?R]):-
    check_constraint(?H,?F),
    check_constraints(?T,?R).

check_constraint(?Cons,successful(?Cons)):- constraint(?Cons).

check_constraint(?Cons,failure(?Cons)):-
    naf constraint(?Cons).

verify_results([]).
verify_results([successful(?C)|?T]):-
    writeln(successful(?C))@_plg,
    verify_results(?T).

verify_results([failure(?C)|?T]):-
     writeln(failure(?C))@_plg,
     verify_results(?T).
```

Fig. 6 Checking constraints for violations

4.1 Sample Web Service Specification

Figure 7 depicts the specification of a web service for making doctor appointments. The precondition requires an object of concept *RequestAppointment* to be provided by the goal, containing the request information. *?DS,?PN,?Date,?HN* and *?X* are logic variables that will be bound to the corresponding values that should be provided by the requester. Before the match is successful, it must be verified that the age of the patient is more than 18 (for some reason!), there is a hospital in the same country that the patient lives in, and there is a doctor with the correct specialization area in that hospital.

The postcondition of the web service specification makes available an *Appointment* object containing information about the appointment date, doctor name,patient name and hospital name.

The local ontology used by the web service specification, also given in Figure 7, represents an abstraction of an actual local database that might be used by the web service that is being semantically described by the specification.

4.2 Sample Goal for Consuming an Appointment Service

We have in Figure 8 a goal for consuming an appointment making service. The goal gives specific information about the appointment, such as the required specialty, hospital name, and age of the patient. The appointment date field is left empty, which

```
ws2[inputs->
    ${startW[
            pre->{and(
                    ${?-[
                          specialty->?DS,
                          patientName->?PN,
                          appointmentDate->?Date,
                          hospitalName->?HN,
                          age->?X]:RequestAppointment
                      },
                        and(
                            and(greater(?X, 18),
                                and(
                                    lives_in_country(?PN, ?Country),
                                    hospital(?HN, ?Country)
                                )
                            ),
                            ${?doctor[
                                      doctorName->?DN,
                                      specialty->?DS,
                                      hospitalName->?HN,
                                      availableDate->?Date
                                     ]:Doctor
                              }
                            )
                        )
                    },

            post->${
                    ?-[
                        appointmentDate->?Date,
                        doctorName->?DN,
                        patientName->?PN,
                        hospitalName->?HN
                       ]:Appointment
                    }
            ]
    }
   ].

doctor1[
        doctorName->robert,
        specialty->opthalmology,
        hospitalName->MontpellierHospital,
        availableDate->{20, 21, 22, 23, 24, 25}
       ]:Doctor.

doctor2[
        doctorName->omid,
        specialty->otolaryngology ,
        hospitalName->MontpellierHospital,
        availableDate->{13, 14, 15, 16}
       ]:Doctor.

doctor3[
        doctorName->martin,
        specialty->gynecology ,
        hospitalName->MontpellierHospital,
        availableDate->{3, 4, 5, 6}
       ]:Doctor.

hospital(MontpellierHospital, France).
```

Fig. 7 Web service specification for making appointments

```
goal2[inputs->
        ${startG[
                pre->${
                        ?_[
                            specialty->opthalmology,
                            patientName->philip,
                            appointmentDate->?_,
                            hospitalName->MontpellierHospital,
                            age->22
                           ]:RequestAppointment,
                        lives_in_city(philip,Paris)
                       },

                post->{
                        and(
                            ${reqApp[
                                      appointmentDate->?Date,
                                      doctorName->?DN,
                                      patientName->philip,
                                      hospitalName->MontpellierHospital
                                     ]:Appointment
                              },
                            or(greater(?Date, 19), less(?Date, 23))
                           )
                       }
                ]
         }
    ].
```

Fig. 8 Goal for consuming the appointment making web service

might mean that any date is fine. However, in the postcondition of the goal, the actual date returned by the web service is checked to be in the range 20 to 22, so any other date will cause a mismatch.

As part of the precondition, the *fact* that the patient (philip) lives in Paris is given. This fact will be used to deduce that philip lives in France and can only make an appointment at a hospital in France through using the web service whose specification is given in Figure 7.

We note how logic variables link the preconditions and postconditions of both goals and web services. In fact, the usual scenario is that information "flows from" the precondition of the goal "into" the precondition of the web service, then "into" the postcondition of the web service, and finally "into" the postcondition of the goal.

4.3 Common Ontology

The common ontology, given in Figure 9, contains constraints that must hold true to have a valid knowledge base, utility predicates (such as the *appointment* predicate that converts *Appointment* objects into a relation), factual information about which city is in which country, as well as a rule which relates people to their countries, based on the city they live in.

```
                                //CommonOntology.flr

constraint(noPatientClash):-
   \+ (( appointment(?pat, ?doc1, ?dt),
         appointment(?pat, ?doc2, ?dt),
         ?doc1 != ?doc2)).

constraints([noPatientClash, noDoctorClash]).

constraint(noDoctorClash):-
   \+ ((appointment(?pat1,?doc,?dt),
        appointment(?pat2,?doc,?dt),
        ?pat1 != ?pat2)).

constraint(appointmentWhenDoctorWorks):-
   \+ ((appointment(?dt,?doc,?pat),
        ?dt[
             nameOfDay->?nod,
             hour->?h
            ]:DateTime,
        \+ worksOn(?doc,?nod,?h))).

?d[dateDoctorFree->?dt]:Doctor:-
   free(?dt,?d).

free(?dt,?d):-
   ?d[dateDoctorFree->?dt]:Doctor.

constaint(validDay):-
   naf (?d[day->?v]:Calendar, ?v > 31).

constraint(validMonth):-
   naf (?m[month->?v]:Calendar, ?v > 12).

constraint(validYear):-
   naf (?y[year->?v]:Calendar, ?v < 2013).

appointment(?patient,?doctor,?dt):-
   ?a[
      patient->?patient,
      doctor->?doctor,
      dateTime->?dt
     ]:Appointment.

greater(?X, ?Y):- ?X > ?Y.
less(?X, ?Y):- ?X < ?Y.
is_equal(?X, ?Y):- ?X = ?Y.

lives_in_country(?Man, ?Country):-
   lives_in_city(?Man, ?City),
   city_country(?City, ?Country).

city_country(London, England).
city_country(Istanbul, Turkey).
city_country(Paris, France).
```

Fig. 9 Common ontology used by goals and web services

```
?Result = [
           [goal1, ' DOES NOT MATCH ', ws1],
           [goal1, ' DOES NOT MATCH ', ws2],
           [goal1, ' MATCHES ', ws3],
           [goal2, ' DOES NOT MATCH ', ws1],
           [goal2, ' MATCHES ', ws2],
           [goal2, ' DOES NOT MATCH ', ws3],
           [goal3, ' MATCHES ', ws1],
           [goal3, ' DOES NOT MATCH ', ws2],
           [goal3, ' DOES NOT MATCH ', ws3],
           [goal4, ' DOES NOT MATCH ', ws1],
           [goal4, ' DOES NOT MATCH ', ws2],
           [goal4, ' DOES NOT MATCH ', ws3]
          ]
```

Fig. 10 Result returned by the matchmaker on some goals and web services

Constraints are checked by the constraint handling component of the matcher, as already explained. The *noPatientClash* constraint makes sure that a patient is not given appointments in the same time slot to different doctors. The *noDoctorClash* guarantees that a doctor will not see two patients in the same time slot. The *appointmentWhenDoctorWorks* constraint checks that a doctor is given a patient only during his working hours. The constraints *validDay*, *validMonth* and *validYear* have obvious meaning. The predicate *appointment* generates a relation from *Appointment* objects.

Obviously, *where* information will be placed (local or common ontology) is a design decision, and some information placed in the common ontology here could have been put inside the local ontologies of web services. However, it is common sense to place rules that can be used by more than one web service specification, or by both goals and web service specifications, inside an "outside" ontology that can act as a common ontology.

4.4 Running the Matchmaker Agent on a Set of Goals and Web Service Specifications

Figure 10 contains the output of the matcher when used to match one set of goals against a set of web service specifications. Due to space limitations, the goals and web service specifications are not included in the paper, but are available for downloading at [https://sourceforge.net/projects/sws-goal-matchmaker/files/].

5 Discussion

Although we found Flora-2 to be a very powerful logic system, its behavior with respect to logical variables inside objects was somewhat different from our expectations, and we had to experiment with reification at different levels (including reification of the components of objects) to make the system behave as we desired. In Figure 11 we give some examples demonstrating the behavior of Flora-2 with respect

to unification, reification and logic variables. The result of the query and its result are given in the comments below the code.

```
1
2  pred1( f[b->?x,c->?x] ).
3  // query:  pred1(f[b->1,c->?y]).
4  // Ans:    ?y unbound
5
6  a[b->${c[d->?x]}, k->${e[f->?x]}].
7  // query: a[b->${c[d->1]},k->${e[f->?x]}].
8  // Ans:   ?x unbound.
9
10 pred3( ${f[b->?x,c->?x]}).
11 // query: pred3(f[b->1,c->?x]).
12 // Ans: NO
13 // query: pred3(${f[b->1,c->?x]}).
14 // Ans: ?x=1
15
16 id12(${o5[a->?x]},${o6[b->?x]}).
17 // query: id12(${o5[a->1]},${o6[b->?z]})
18 // Ans: ?z=1
19
20 project2(${a2[b2->?z]},?z).
21 // query: project2(?x,?y).
22 // Ans: ?x = ${a2[b2 -> ?_h2209]}
23 //      ?y = ?_h2209
24 // query: project2(a2[b2->1],?z)@ml.
25 // Ans: No
26 // query: project2(${a2[b2->1]},?z).
27 // Ans: ?z = 1
```

Fig. 11 Behavior of Flora-2 with respect to unification, reification and logic variables

6 Related Work on Matching

Matching request specifications against semantic web service capability has recently been an active area of research. Below we review some of the most relevant work in this area.

As already mentioned in the introduction, authors in [32] describe in detail several set-based approaches to discovery, but they also propose a discovery method using proof commitments in transactional logic. Their proposal however is so general that its efficient implementation is practically impossible. They place no restrictions on the logic statements that can take part in the preconditions and postconditions of web service capabilities or goal specifications, making the use a full-fledged transaction logic reasoner necessary to prove their commitments. Furthermore, their proof commitments involve an existential quantifier for web service specifications, so that discovery of a suitable web service is delegated to the deduction process (i.e. computation carried out for the satisfaction of the proof commitment) completely. In our

case, our proof commitments are logical entailments in first-order logic, with well-defined restrictions on goal and web service specifications in order to have efficient goal-directed proofs of the proof commitments. Specifically, our web service preconditions and goal postconditions are implicitly existentially quantified statements involving conjunction, disjunction and negation operators, and act like queries in the logic programming Flora-2 (or any other logic programming language). Goal preconditions and web service postconditions are implicitly universally quantified, contain only positive statements and involve only the conjunction operator. These restrictions allow us to efficiently check the proof commitments, much like queries are answered in a top-down manner in logic programming languages. To make our matching agent even more efficient, our proof commitments involve only one goal and one web service: individual goals and web services are checked iteratively to see if the proof commitment holds between them. Consequently, we have been able to build an actual, practical implementation for our matching agent, whereas our literature search has failed to uncover a real transaction-logic based matcher that checks the proof commitments specified in [32]. Incidentally, we do use the transactional logic capabilities of Flora-2 in our implementation, but only as a tool.

In [27,31,29,30,17], the authors describe a matching algorithm for automatic dynamic discovery, selection and interoperation of web services based on DAML-S. They show a representation for service capabilities in the Profile section of a DAML-S description and a way to semantically match advertisements and requests. In related work, a way to map DAML-S service profiles into UDDI records and using the encoded information to perform semantic matching is described in [26]. The actual matching is performed by the "Matching Engine" component, which makes use of the "DAML=OIL Reasoner" to compute the level of the match. Since the approach used in DAML-S is fundamentally set-based, it suffers from the same drawbacks as other set-based methods of discovery.

The matchmaking method described in [3] assigns matchmaking scores to condition expressions in OWL-S documents written in SWRL. It uses a reasoner to determine subsumption relationships and compute scores for each advertisement. This approach is also a set-based and does not specify any proof commitments explicitly.

The authors in [23] consider matchmaking between service provider agents and service requester agents. *Middle agents* perform the matchmaking between the requester and service provider agents. They present the agent description language LARKS, and discuss the matchmaking process using LARKS. A specification in LARKS is a frame which includes slots "context," "input," "output," "inconstraints," "outconstraints." Their matchmaking algorithm determines the relationship among two semantic descriptions by computing the respective subsumption relationship, and is again set-based.

[28] is a high-level survey and ranking of web service discovery methods and does not give any details regarding the specific matching algorithm used by the surveyed methods.

In [30] the authors propose OWL-S/UDDI matchmaker. They embed OWL-S profile information inside UDDI advertisements before performing a match. Their algorithm matches the outputs/inputs of the request against the inputs/outputs of the published advertisements. The match between the inputs or outputs depends on the

relationship between the OWL concepts to which the objects belong. So the proposal is a set-based approach using OWL-S. Any reasoning done is for determining subsumption realtionships. An implementation of a matcher using the OWL-S/UDDI matchmaker is given in [7].

A matchmaking algorithm based on bipartite graphs for semantic web services specifies in OWL-S is presented in [16]. The approach carries out semantic similarity assignment using subsumption, properties, similarity distance annotations and WordNet. No logical inference is carried out, except for subsumption.

In [11], the authors formalize the matching problem in general using Description Logics, and devise "Concept Abduction" and "Concept Contraction" as nonmonotonic inferences in Description Logics for modeling matchmaking in a logical framework. They also give algorithms for semantic matchmaking based on the devised inferences as well. Similary, in [15] a framework for annotating Web Services using description logics (DLs) is used and it is shown how to realise service discovery by matching semantic service descriptions, applying DL inferencing. Description Logics is a very limited form of logic, compared to the first order logic, and matching based on DL specifications is not directly comparable to our work, which deals with first order logic specifications.

In [10] authors implemented in F-Logic a matching mechanism that relies on web service-goal mediators . They used Flora-2 in the matching procedure to evaluate the similarity rules embedded in the description of each mediator and return references to the discovered Web services and the degree of matching (exact, subsumed, plug-in and intersection). It is clear that their approach is strictly set-based and does not involve logical inference in first-order logic.

The WSMO-MX service matchmaker, described in [20], uses different matching filters to retrieve Semantic Web services written in a dialect of WSML-Rule. WSMO-MX recursively computes "logic-based and syntactic similarity-based matching degrees and returns a ranked set of services that are semantically relevant to a given query. The matching filters perform ontology-based type matching, logical constraint matching, and syntactic matching." The proposed system is "aproximative" and does not guarantee with 100% certainty the suitability of the discovered services to satisfy the needs of the requester. This is in line with the authors' belief that Semantic Web research has started to shift towards "more scalable and approximative rather than computationally expensive logic-based reasoning with impractical assumptions." Our work disproves this belief: it *is* possible to have logic-based reasoning that is not prohibitively expensive, provided that an appropriate subset of first order logic which is expressive enough to practically specify goal requirements and web service capabilities is used.

In other related work, [8] describes a new algorithm for matching web services in YASA4WSDL. The matching algorithm consists of three variants based on three different semantic matching degree aggregations. Their method uses an algorithm using extended semantic annotation, based on Web Service standards. The critical problems in Web Service discovery such as how to locate Web Services and how to select the best one from large numbers of functionally similar Web Services are explained in [12]. In [21], a graded relevance scale for semantic web Services (SWS)

matchmaking is proposed as measurements to evaluate SWS matchmakers based on such graded relevance scales.

7 Conclusion and Future Research Directions

Using a sub-language of F-logic to specify ontologies, web services and goals, and Flora-2 as the implementation tool, we have built an intelligent matchmaker agent for matching semantic web services and goals using a purely logical inference based approach. We specified explicitly in terms of logical entailment the proof commitments that must be verified before a match between a goal and web service can succeed. Our sublanguage of F-logic has implicit existential and universal quantifiers (depending on where the formula is used) that permits efficient goal-directed deduction as in the case of logic programming, allows relatively uncomplicated specification of web services and goals, and is powerful enough to effectively specify goals and web service capabilities as desired. We explained in some detail our implementation of the matchmaker agent, which makes use of the higher-order capabilities, transactional logic extensions, reification and module facilities of Flora-2, as well as its built-in inference engine. Since ontologies are part of the matchmaking process, and integrity of knowledge contained in the ontologies must be guaranteed, our matchmaker has a constraint verification part as well. We also illustrated the use of our sublanguage of F-logic in the specification of web services, goals and ontologies through an appointment making scenario, where the goal is to make a doctor appointment for a patient.

Our approach stands out among all other approaches to semantic web service matchmaking due to its purely logical basis, unambiguous definition of what a match means in terms of proof commitments, and efficient implementation made possible through diligent selection of a sublanguage of F-logic for specifying goals, web services and ontologies.

For future work, we are planning to use our logical framework for composing different web services to achieve the needs of the client, in case a single web service cannot do so. Our plans also include defining a variant of WSML that maps directly to our sublanguage of F-logic that we use for specifying goals, web services and ontologies, and implementation of a translator from this variant of WSML to our sublanguage. We believe such an implementation will be an important step in making WSML a viable alternative in industry for semantic web service specification.

References

1. General information on the sti discovery working group. http://wiki.wsmx.org/index.php?title=Discovery
2. How to write f-logic-programs. http://www.semafora-systems.com/documents/tutorial_flogic.pdf
3. Bener, A.B., Ozadali, V., Ilhan, E.S.: Semantic matchmaker with precondition and effect matching using swrl. Expert Systems with Applications **36**(5), 9371–9377 (2009)
4. Bonner, A.J., Kifer, M.: An overview of transaction logic. Theoretical Computer Science **133**(2), 205–265 (1994)
5. Bonner, A.J., Kifer, M.: A logic for programming database transactions. In: Logics for databases and information systems, pp. 117–166. Springer (1998)

6. de Bruijn, J.: Wsml language reference, deliverable d16. 1v1. 0. WSML Final Draft pp. 08–08 (2008)
7. Celik, D., Elci, A.: Discovery and scoring of semantic web services based on client requirement (s) through a semantic search agent. In: Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International, vol. 2, pp. 273–278. IEEE (2006)
8. Chabeb, Y., Tata, S., Ozanne, A.: Yasa-m: A semantic web service matchmaker. In: Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on, pp. 966–973. IEEE (2010)
9. De Bruijn, J., Bussler, C., Domingue, J., Fensel, D., Hepp, M., Kifer, M., König-Ries, B., Kopecky, J., Lara, R., Oren, E., et al.: Web service modeling ontology (wsmo). Interface **5**, 1 (2006)
10. Della Valle, E., Cerizza, D.: Cocoon glue: a prototype of wsmo discovery engine for the healthcare field. In: Proceedings of 2nd WSMO Implementation Workshop WIW, vol. 2005 (2005)
11. Di Noia, T., Di Sciascio, E., Donini, F.M.: Semantic matchmaking as non-monotonic reasoning: A description logic approach. Journal of Artificial Intelligence Research **29**(1), 269–307 (2007)
12. Fan, J., Ren, B., Xiong, L.R.: An approach to web service discovery based on the semantics. In: Fuzzy Systems and Knowledge Discovery, pp. 1103–1106. Springer (2005)
13. Fensel, D., Bussler, C.: The web service modeling framework wsmf. Electronic Commerce Research and Applications **1**(2), 113–137 (2002)
14. Gallier, J.H.: Logic for computer science: foundations of automatic theorem proving. Harper & Row Publishers, Inc. (1985)
15. Grimm, S., Motik, B., Preist, C.: Matching semantic service descriptions with local closed-world reasoning. In: The Semantic Web: Research and Applications, pp. 575–589. Springer (2006)
16. Ilhan, E., Bener, A.: Improved service ranking and scoring: Semantic advanced matchmaker (sam) architecture. Evaluation of Novel Approaches to Software Engineering (ENASE 2007), Barcelona (2007)
17. Kawamura, T., De Blasio, J.A., Hasegawa, T., Paolucci, M., Sycara, K.: Preliminary report of public experiment of semantic service matchmaker with uddi business registry. In: Service-Oriented Computing-ICSOC 2003, pp. 208–224. Springer (2003)
18. Kifer, M., Lausen, G.: F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. In: ACM SIGMOD Record, vol. 18, pp. 134–146. ACM (1989)
19. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM (JACM) **42**(4), 741–843 (1995)
20. Klusch, M., Kaufer, F.: Wsmo-mx: A hybrid semantic web service matchmaker. Web Intelligence and Agent Systems **7**(1), 23–42 (2009)
21. Küster, U., König-Ries, B.: Evaluating semantic web service matchmaking effectiveness based on graded relevance. In: The 7th International Semantic Web Conference, p. 35 (2008)
22. Logic, T.: Temporal logic. Stanford Encyclopedia of Philosophy (2002)
23. Lu, J.: Dynamic service matchmaking among agents in open information environments* katia sycara, matthias klusch, seth widoff the robotics institute, carnegie mellon university, pittsburgh, usa.{katia, klusch, swidoff}@ cs. cmu. edu (1999)
24. May, W.: How to write f-logic programs in florid. Tech. rep., Technical report, Institut für Informatik, Universität Freiburg (2000)
25. Michael Kiefer Guizhen Yang, H.W.C.Z.: Flora-2: Users' manual (version 0.99.3). Electronically available from flora.sourceforge.net/docs/floraManual.pdf (2013)
26. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.: Importing the semantic web in uddi. In: Web Services, E-Business, and the Semantic Web, pp. 225–236. Springer (2002)
27. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.: Semantic matching of web services capabilities. In: The Semantic WebSWC 2002, pp. 333–347. Springer (2002)
28. Rambold, M., Kasinger, H., Lautenbacher, F., Bauer, B.: Towards autonomic service discovery a survey and comparison. In: Services Computing, 2009. SCC'09. IEEE International Conference on, pp. 192–201. IEEE (2009)
29. Srinivasan, N., Paolucci, M., Sycara, K.: Adding owl-s to uddi, implementation and throughput. proceeding of Semantic Web Service and Web Process Composition 2004 (2004)
30. Srinivasan, N., Paolucci, M., Sycara, K.: An efficient algorithm for owl-s based semantic search in uddi. In: Semantic Web Services and Web Process Composition, pp. 96–110. Springer (2005)
31. Sycara, K., Paolucci, M., Ankolekar, A., Srinivasan, N.: Automated discovery, interaction and composition of semantic web services. Web Semantics: Science, Services and Agents on the World Wide Web **1**(1), 27–46 (2003)
32. U., K., R., L., A., P., M., K., D., F.: D5.1 v0.1 wsmo web service discovery (2004)
33. Wan, H., Grosof, B., Kifer, M., Fodor, P., Liang, S.: Logic programming with defaults and argumentation theories. In: Logic Programming, pp. 432–448. Springer (2009)