A New Logic-Based Approach for the Matching of Goals and Semantic Web Services: 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 services · Matching · Discovery · Intelligent Agent · F-Logic · Logical Inference

O. Sharifi
Department of Computer Engineering, Toros University,
Mersin, Turkey
Tel.: +90-324-3253300
Fax: +90-324-3253301
E-mail: omid.sharifi@toros.edu.tr

Z. Bayram
Department of Computer Engineering, Eastern Mediterranean University,
Gazimagusa, Turkey
Tel.: +90-392-6302840
Fax: +90-392-6350711
E-mail: zeki.bayram@emu.edu.tr
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 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” in WSMO terminology) and Web service specifications, and use the same formalism for specifying both. WSMO itself is based on the Web Service Modeling Framework (WSMF) [13] and has four main components, which are Ontologies, Web Services, Goals and Mediators.

Web Service Modeling Language (WSML) [5] is a language used to describe ontologies, Semantic Web Services (SWS) and goals in conformance to the WSMO framework. It has a solid logic foundation, namely F-Logic [16]- 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 [5].

There are several discovery approaches used in WSML [20]. 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 [34].

Different kinds of match have been defined for the the set based approach [20]. These are summarized below.

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

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 these 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 and 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) 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 [22,1] as the starting point for the specification 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 [22], 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, in addition the section denotes the performance statistics of the proposed method. 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. Finally section 7 proposes the conclusion and future research directions.

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 Web 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 FLOG4SWS: A 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. Even 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) temporarily by inserting antecedents of implications into the logic system database (as well as the common ontology [8] 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 Fig. 1 we have the Extended Backus Normal Form (EBNF) grammar for the sub-language 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 [26]. 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. Note 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 of 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.
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. \( co \land \text{goal.ont} \land \text{ws.ont} \land \text{goal.pre} \models \text{ws.pre} \): 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. \( \text{goal.pre} \)).

2. \( co \land \text{goal.ont} \land \text{ws.ont} \land \text{goal.pre} \land (\text{ws.pre} \Rightarrow \text{ws.post}) \models \text{goal.post} \): 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 in \( \text{ws.pre} \Rightarrow \text{ws.post} \).

3. \( \text{worldBefore} \models \text{ws.assumption} \): 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 ⋀ (ws.assumption ⇒ ws.effect) |= 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 \(ws.assumption ⇒ ws.effect\).

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 \((ws.pre ⇒ ws.post)\), and an implication from its assumption to its effect \((ws.assumption ⇒ ws.effect)\). 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 Web service 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 [21]) 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 [3] or temporal logic [15]. So that they 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 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.
2.6 The Intelligent Matchmaker Agent Architecture

Fig. 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 equivalencies 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 [4, 3] into tabled Prolog code [26]. Basically, FLORA-2 supports a programming 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 [37]. 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 [25].

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 also 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 post-conditions, and extending our program to deal with them is straightforward.

3.3 The Top-level Matcher Loop

Fig. 3 has predicates that contain information about the files containing goals, Web services and the common ontology. Each WebService predicate contains information about a specific Web service, a Goal predicate contains information about a specific goal, and CommonOntology 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 (?).

Fig. 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 service matches which goal. run2 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 Fig. 4, the current module is forced to erase its contents by calling erasemodule function, 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 Fig. 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 \[ co \land \text{goal.ont} \land \text{ws.ont} \land \text{goal.pre} \models \text{ws.pre} \] is verified, and in line 8, the proof commitment \[ co \land \text{goal.ont} \land \text{ws.ont} \land \text{goal.pre} \land (\text{ws.pre} \Rightarrow \text{ws.post}) \models \text{goal.post} \] 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 \[ co \land \text{goal.ont} \land \text{ws.ont} \land \text{goal.pre} \models \text{ws.pre} \] (i.e. \[ co \land \text{goal.ont} \land \text{ws.ont} \land \text{goal.pre} \]) 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 \[ co \land \text{goal.ont} \land \text{ws.ont} \land \text{goal.pre} \land (\text{ws.pre} \Rightarrow \text{ws.post}) \models \text{goal.post} \], however, since FLORA-2 does not allow temporary insertion of rules (of the form \( \text{lhs} : \neg \text{rhs} \)) in a
Matching Goals and SWS: Logical Inference Based Discovery Agent

module. What we needed was the ability to insert into the module \( ws.post \) : \( \neg ws.pre \) (\( ws.post \) 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 \( \text{prove}/3 \) 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 Fig. 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 \( ws.post : \neg ws.pre \) was inserted into the module.

```prolog
run (? Result): –
    WebServices (?WSS),
    Goals (?GS).
run2 ([], ?,[]): –
    true, !. // goals finished
    true, !,
    WebServices (?WSS),
run2 ([? Goal|? RestGoals], [?WS]? RestWebServices],
    [[? Goal, ’MATCHES’, ?WS]? RestResult]): –
    main (?Goal, ?WS), !,
run2 ([? Goal|? RestGoals], [?WS]? RestWebServices],
    [[? Goal, ’DOES NOT MATCH’, ?WS]? RestResult]): –
    true, !,
    WebService (?WS, ?, ?WsModule),
    erasemodule {?WsModule},
main (?GoalName, ?WsName): –
    WebService (?WsName, ?WsPath, ?WsModule),
    Goal (?GoalName, ?GoalPath),
    CommonOntology (?OntologyPath),
    [+’C:\Users\OMID\Desktop\test\matcher’>>?WsModule],
    [+?WsPath>>?WsModule],
    [+?OntologyPath>>?WsModule],
    [+?GoalPath>>?WsModule],
    matchh (?GoalName, ?WsName, ?WsModule)@?WsModule,
    ?FoundWs=?WsName,
    erasemodule {?WsModule}, !.
```

Fig. 4: Matching agent top-level predicates.
    check_constraints, 
    ?WebService [ inputs->$ { startW [ pre->?WsPre, post->?WsPost ] } ], 
    t_insert [ ?GoalPre ] @ ?WsModule, 
    prove (?WsPre), 
    check_constraints, 
    prove (?GoalPost, ?WsPost, ?WsPre), 
    check_constraints.

prove (and (?X, ?Y)):− !, prove (?X), prove (?Y).
prove (or (?X, ?Y)):− !, prove (?X); prove (?Y).
prove (not (?X)):− !, naf (prove (?X)).

prove (?GoalPost, ?WsPost, ?WsPre):−
    ?GoalPost = ?WsPost, !, prove (?WsPre).
prove (and (?X, ?Y), ?WsPost, ?WsPre):− !,
prove (or (?X, ?Y), ?WsPost, ?WsPre):− !,
    prove (?X, ?WsPost, ?WsPre); prove (?Y, ?WsPost, ?WsPre).
prove (not (?X), ?WsPost, ?WsPre):− !,
    naf (prove (?X, ?WsPost, ?WsPre)).

Fig. 5: Proving the commitments for a match

3.5 Checking Constraints

Fig. 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 can be 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 consuming the appointment service, and (iii) the common ontology used by the Web service and goal.
Fig. 6: Checking constraints for violations

4.1 Sample Web Service Specification

Fig. 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 Fig. 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 Fig. 8 a goal for consuming an appointment making service. The goal gives specific information about the appointment, such as the required specialty, hos-
Fig. 7: Web service specification for making appointments
Fig. 8: Goal for consuming the appointment making Web service

pital name, and age of the patient. The appointment date field is left empty, which 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 Fig. 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 Fig. 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.frr

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).

constraint (validDay): -
  naf (?d{ day->?v } : Calendar, ?v > 31).

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

constraint (validYear): -

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


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

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

Fig. 9: Common ontology used by 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

Fig. 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 [31].

4.5 Performance Statistics for FLOG4SWS

Table 1 depicts the time it took to match goal2 against two different sets of Web service specifications, one where it matched all the Web services, and another where it matched none of the Web services. Figures 11 and 12 depict the same information graphically. As can be seen, there is a linear relationship between the number of Web
Table 1: Matching time of one goal with different number of Web services in FLOG4SWS

<table>
<thead>
<tr>
<th>Number of Web services</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal (matched)</td>
<td>0.407</td>
<td>0.860</td>
<td>1.437</td>
<td>1.828</td>
<td>2.203</td>
<td>2.734</td>
<td>3.265</td>
<td>3.703</td>
<td>4.891</td>
</tr>
<tr>
<td>Goal (not matched)</td>
<td>0.312</td>
<td>0.500</td>
<td>0.766</td>
<td>0.890</td>
<td>1.312</td>
<td>1.453</td>
<td>1.813</td>
<td>2.000</td>
<td>2.750</td>
</tr>
</tbody>
</table>

services, and the time it took to perform the match in either case. The graph for the matching case can be described by the formula $t = 0.52s - 0.22$ and the graph for the non-matching case can be described by the formula $t = 0.28s - 0.098$ ($t$=time in seconds, $s$=number of Web services), as found by the curve-fitting function of Matlab.
5 Discussion

Although we found FLORA-2 to be a very powerful logic system with frames, 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 Fig. 13 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 (the dollar sign $ is used to reify a component).

```
// query:  pred1(f[b->1,c->?y]).
// Ans:  ?y unbound
5
a[b->$c[d->?x]]. k->$e[f->?x]]).
// query:  a[b->$c[d->1]]. k->$e[f->?x]]).
// Ans:  ?x unbound.

// query:  pred3(f[b->1,c->?x]).
// Ans: NO
// query:  pred3($f[b->1,c->?x])).
// Ans:  ?x=1

id12($o5[a->?x]].$o6[b->?x]]).
// query:  id12 ($o5[a->1],o6[b->?x])
// Ans:  ?x=1

project2 ($a2[b2->?z]].?z).
// query:  project2 (?x,?y).
// Ans:  ?x = $a2[b2 -> ?_h2209]
// ?y = ?_h2209
// query:  project2 (a2[b2->1],?x)@ml.
// Ans: NO
// query:  project2 ($a2[b2->1],?z).
// Ans: ?z = 1
Fig. 13: 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 [20] 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 F-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. 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 commitments 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 [20]. Incidentally, we do use the transactional logic capabilities of FLORA-2 in our implementation, but only as a tool.

In [28,35,32,33,19], 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 [27]. 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 [2] 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 [36] 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.

[30] 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 [33] 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 relationships. An implementation of a matcher using the OWL-S/UDDI matchmaker is given in [6].

A matchmaking algorithm based on bipartite graphs for Semantic Web services specified in OWL-S is presented in [18]. 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 non-monotonic 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. Similarly, in [17] 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 F-logic specifications.

In [10] authors implement in F-Logic a matching mechanism that relies on Web service-goal mediators. They use 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.

The WSMO-MX service matchmaker, described in [23], 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 “approximative” 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 [29] 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 logic
which is expressive enough to practically specify goal requirements and Web service capabilities is used.

In other related work, [7] 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 [24], a graded relevance scale for Semantic Web services 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