Received: 27 November 2018 | Accepted: 18 January 2019
DOI: 10.1111/exsy.12387

WILEY Expert Systems

SPECIAL ISSUE PAPER

Frame Logic-based specification and discovery of semantic web services with application to medical appointments

Omid Sharifi[1] | Shahin Mehdipour Ataee[2] | Zeki Bayram[2]

[1]Computer and Software Eng ineering Department, Toros University, Mersin, Turkey
[2]Computer Engineering Department, Eastern Mediterranean University, Famagusta, North Cyprus

Correspondence
Zeki Bayram, Computer Engineering Department, Eastern Mediterranean University, Famagusta, North Cyprus, via Mersin 10, Turkey.
Email: zeki.bayram@emu.edu.tr

Present Address
Shahin Mehdipour Ataee, Computer Engineering Department, Final International University, Kyrenia, North Cyprus

Funding information
This research was carried out using the facilities available at the institutions of the authors, without utilizing any other source of funding.

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 services 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 an alternative, we use the Frame Logic as implemented in Flora-2 to specify web service capabilities and client requirements, including their preconditions, postconditions, and ontologies, implement a logic-based discovery agent using Flora-2, demonstrate its usefulness in a medical appointment making scenario, and show its efficiency both theoretically and by benchmarking. The result is an expressive yet concise representation scheme for semantic web services, and a practical, efficient, powerful, and fully implemented matching engine based purely on logical inference for web service discovery, with direct applicability to Web Service Modeling Ontology and Web Service Modeling Language, because both are based on Frame Logic.

KEYWORDS
discovery, frame logic, intelligent agent, logical inference, matching, medical appointment, semantic web services

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 semantic web service frameworks, we take Web Service Modeling Ontology (WSMO) (Fensel et al., 2007) 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 uses the same formalism for specifying both. WSMO itself is based on the Web Service Modeling Frame-work (WSMF) (Fensel & Bussler, 2002) and has four main components, namely, *Ontologies*, *Web Services*, *Goals*, and *Mediators*.

WSML (de Bruijn, 2008) represents a class of related languages used to describe ontologies, web services, and goals in conformance to the WSMO framework. It has a solid logic foundation, namely, Frame Logic (or F-Logic in short) (Kifer, Lausen, & Wu, 1995; Kifer & Lausen, 1989)

Abbreviations: F-Logic, Frame Logic; FLOG4SWS, Frame LOGic 4 Semantic Web Services; FOL, First Order Logic; WSML, Web Service Modeling Language; WSMO, Web Service Modeling Ontology; UDDI, Universal Description Discovery and Integration.

— 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 the study of de Bruijn (2008).

There are several discovery approaches used in WSML (Keller et al., 2004). 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 take 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* statements, as well as 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.

Different kinds of *match* have been defined for the set-based approach (Keller et al., 2004). 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* are a super set of the objects requested in the goal *G*.
- *Intersection Match* happens when the delivered items of the service *W* have 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 (a) allow relatively uncomplicated specification of web services and goals, (b) permit efficient execution of inference engines during the matching process, and (c) be powerful enough to effectively specify goals and web service capabilities.

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

1. We specify a sublanguage of F-Logic, which we call FLOG4SWS (Frame LOGic 4 Semantic Web Services) 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,
3. We implement a logical entailment-based (intelligent) matching agent using Flora-2, demonstrating not only the feasibility of our approach but also its practicality, and finally,
4. We benchmark the execution of the matching engine under different scenarios, with results showing that our approach is scalable, both in terms of the number of web services that are involved in each match and also in terms of the complexity of the logical expressions making up the pre and postconditions.

We chose F-Logic (Kifer et al., 1995; Bayram & Sharifi, 2016) as the starting point for the specification language of our matching agent, because 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. 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.

Because WSML is based directly on F-Logic (Kifer et al., 1995), our implementation, available for download at (Bayram, Sharifi, & Ataee, 2019), is a showcase for how WSML web services and goals can be represented in Flora-2 and how the Flora-2 engine can be used to perform logical entailment-based matching between goals and web services.

The remainder of this article is organized as follows. Our sublanguage of F-Logic used to describe ontologies, goals, and web services, as well as the matchmaker intelligent agent architecture is described in Section 2. In section 3, we have a sample ontology, goal, and web service specification that the matchmaker agent can use as inputs, demonstrating the power and convenience of using logic enhanced with frames. Highlights of the matchmaker agent implementation in Flora-2 are given in Section 4. Section 5 gives benchmarking results for the matcher, demonstrating the scalability of our approach. Section 6 describes other prominent matching approaches reported in literature and compares them with our approach. Section 7 is the conclusion and future research directions. Finally, the precise syntax of FLOG4SWS is given in Appendix A as an Extended Backus Normal Form grammar.

2 | THE INTELLIGENT SEMANTIC WEB SERVICE MATCHMAKER AGENT

In this section, we point out the logical components of web service and goal specifications that play a significant role in the matchmaking process, define a sublanguage of F-Logic (called FLOG4SWS) for specifying semantic web services, explain the precise relationship between FLOG4SWS and WSMO, state the proof commitments that must be shown true for a match between a goal and a web service to be successful, and finally present the overall matchmaker agent architecture.

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 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, because each ontology can be assigned a unique name space in the web environment.

2.2 | FLOG4SWS: A sublanguage 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 (a) efficient implementability of logical inference needed to verify the validity of the proof commitments derived from the specifications and (b) 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 (Gallier, 1985) 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.

In order to achieve this goal, we use a sublanguage of F-Logic in our web service and goal specifications. There is no explicit use of quantifiers: Formulas used in *goal.pre* are assumed to be ground, because the goal needs to be providing input data to the web service; variables in *ws.pre* that are not also in *ws.post* are assumed to be existentially quantified; all variables in *goal.post* are assumed to be existentially quantified as well. Variables in *ws.pre* that are also in *ws.post* are assumed to be universally quantified. Any variable in *ws.post* that does not already occur in *ws.pre* is assumed to be universally quantified, although this case is not expected to happen in realistic situations. *goal.pre* and *ws.post* are conjunctions of positive molecules and predicates; *ws.pre* and *goal.post* on the other hand are can involve conjunction, disjunction, and negation connectives.

Our choice of sublanguage 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 (given below) are verified with the same kind of efficiency that logic programmes are executed in a logic programming framework. Benchmarking results reported in Section 5, as well as the complexity analysis presented in Section 4.8 lend support to this claim and demonstrate that our logic-based matching is indeed scalable, both in terms of logical expression complexity, as well as the number of web services that are matched against.

Ontologies that are part of a semantic web service specification can contain any facts and rules that are allowed in F-Logic. Furthermore, the matchmaker agent verifies that no constraints specified in the ontologies are violated in any stage of the matching process. The precise syntax of FLOG4SWS is given in Appendix A as an Extended Backus Normal Form grammar, with additional explanations.

2.3 | Relationship of FLOG4SWS to WSMO

Web Service Modeling Ontology, or WSMO, as the name implies, is a high-level ontology spelling out the components of a semantic web service specification, without committing itself to any specific language. Using the Meta Object Facility MOF (Object Management Group, 2016), the main concepts of WSMO at the meta-level, taken from Fensel et al. (2007), are given in Listing 1 (some low-level concepts have been left out due to space considerations).

WSMO has four top-level concepts: *Ontology*, *goal*, *web service*, and *mediator*. *Nonfunctional property*, *imported ontology*, and *used mediator* attributes are common to the ontology, service, and goal concepts. An ontology instance can include concept definitions, relations, functions, instances, as well as axioms. A service instance has an embedded *capability* instance and an *interface* instance. A capability instance basically describes what the web service can deliver, and an interface instance specifies how it interacts with the requester, as well as other web services. The interaction with the requester is specified through a *choreography* specification, and the interaction with other web services in order to provide its functionality is specified through an *orchestration* specification.

Because the focal point of our work is logic-based discovery of semantic web services, FLOG4SWS does not include constructs for all the components of WSMO. Specifically, issues of mediation, non-functional properties, choreography, and orchestration are not addressed. On the other hand, components that are central to logic-based discovery are present: The ontology component is covered completely by leveraging the available constructs of F-Logic (concepts, relations, instances, and axioms), and capability specification of both goals and web services are given first-class treatment. In a sense, the current specification of FLOG4SWS has the core ingredients needed for logic-based specification and discovery of semantic web services but can be extended to include the other components as well.

```
Class ontology
 hasNonFunctionalProperty type nonFunctionalProperty
 importsOntology type ontology
 usesMediator type ooMediator
 hasConcept type concept
 hasRelation type relation
 hasFunction type function
 hasInstance type instance
 hasAxiom type axiom

Class service
 hasNonFunctionalProperty type nonFunctionalProperty
 importsOntology type ontology
 usesMediator type {ooMediator, wwMediator}
 hasCapability type capability multiplicity = single-valued
 hasInterface type interface

Class goal
 hasNonFunctionalProperty type nonFunctionalProperty
 importsOntology type ontology
 usesMediator type {ooMediator, ggMediator}
 requestsCapability type capability multiplicity = single-valued
 requestsInterface type interface

Class capability
 hasNonFunctionalProperty type nonFunctionalProperty
 importsOntology type ontology
 usesMediator type ooMediator
 hasPrecondition type axiom
 hasAssumption type axiom
 hasPost-condition type axiom
 hasEffect type axiom
```

LISTING 1 Concepts of WSMO in MOF notation (taken from Fensel et al. (2007))

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 goal.ont ws.ont 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 goal.ont ws.ont goal.pre* (*ws.pre* $\Rightarrow$ *ws.post*) $\models$ *goal.post*: If the conditions for the web service call are satisfied, then the requirements of the goal (i.e., its postcondition) 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* ⊨ *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).
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, because 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 the state of the common ontology and local ontologies of the goal and web services. We should note, however, that our matching engine, being a proof of concept, does not attempt to verify proof commitments 3 and 4, because their implementation would be practically identical to the verification of proof commitments 1 and 2.

2.5 | The intelligent matchmaker agent architecture

Figure 1 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. Because 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 and reports the result at the end. There is no "degree of match" computed, because we are interested only in web services that satisfy the requirements of a goal completely.

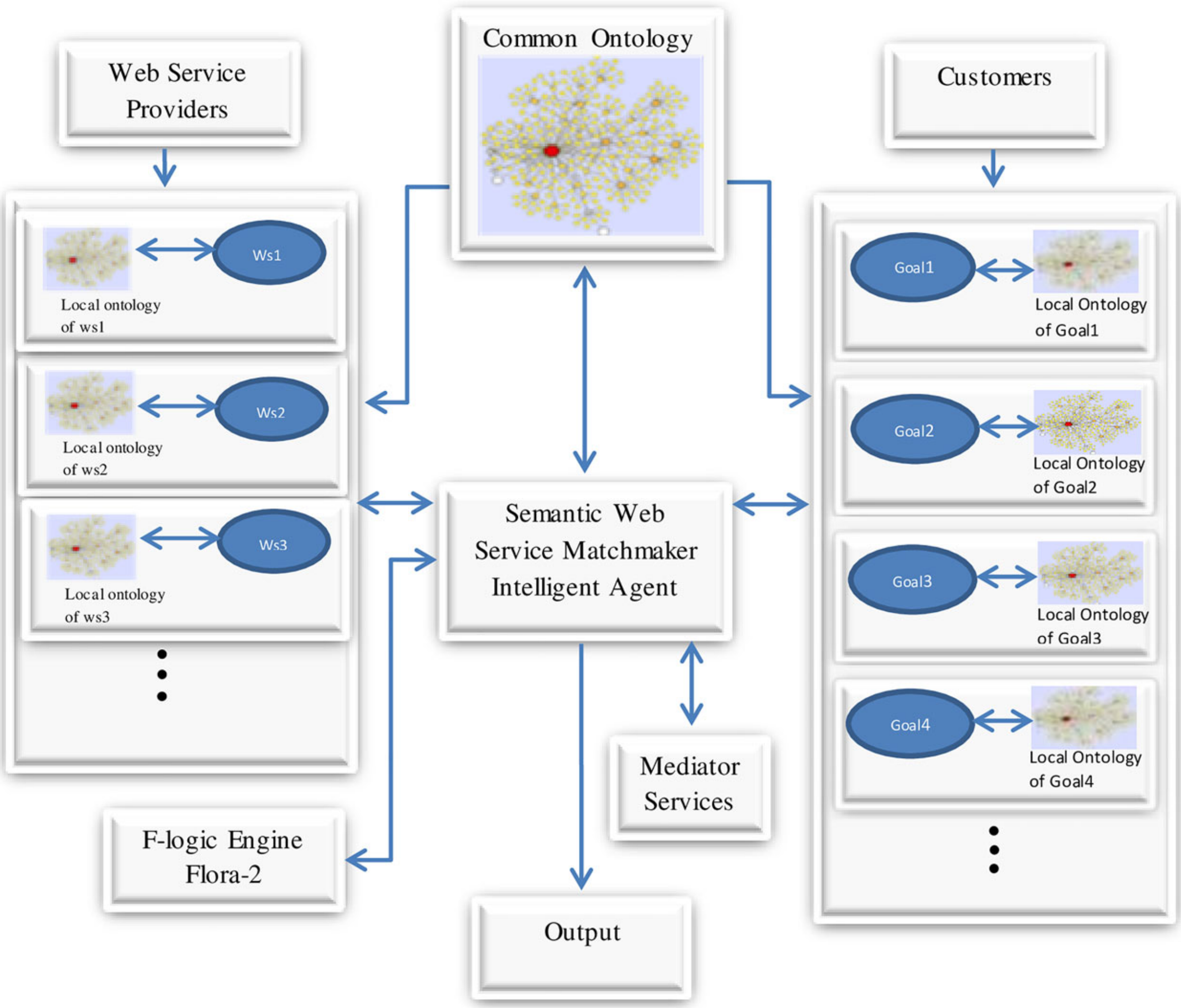

FIGURE 1 The proposed semantic web service matchmaker intelligent agent architecture

"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 equivalences 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 straightforward manner.

3 | SPECIFYING WEB SERVICES, GOALS, AND ONTOLOGIES IN FLORA-2

In this section, we give a representative example of (a) a web service specification for making appointments in hospitals, (b) a goal for consuming the appointment service, and (c) the common ontology used by the web service and goal. The logical expressions in the preconditions and postconditions contain a variety of constructs, such as reified objects, calls to predicates defined in the common ontology and local ontologies, as well as all the logical connectives, in order to give a sampling of what is available in FLOG4SWS.

3.1 | Sample web service specification

Listing 2 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*, *?From_date*, *?To_date*, and *?Age* 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 who is available in the requested date range. Note that the object identifier of the object of type *RequestAppointment* is an anonymous variable, because the web service is not concerned with the actual name of the input object, only its type and whether is has the required attributes. Also, observe how part of the web service precondition (`Doctor` instance and `hospital` predicate) depends on its local ontology, given in Listing 3, part of it (`lives_in_country` and `greater` predicates) depends on the common ontology, given in Listing 5, and the rest depend on the information provided by the goal through its precondition, given in Listing 4. 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. For simplicity, we modeled a date as a single integer.

The local ontology used by the web service specification, given in Listing 3, represents an abstraction of an actual local knowledge base that might be used by the web service that is being semantically described by the specification. This ontology has instances of the *Doctor* concept and facts (in the form of predicates) relating hospitals to their cities.

```
ws1[inputs->
    ${startW(ws1)[
         pre->( ${ ?_[ specialty->?DS,
                      patientName->?PN,
                      from_date -> ?From_date,
                         to_date -> ?To_date,
                      age->?Age]:RequestAppointment },

                   greater(?Age , 18),

                   lives_in_country(?PN, ?Country),

                   hospital(?HN, ?Country),

                   ${?doctor[ doctorName->?DN,
                              specialty->?DS,
                              hospitalName->?HN,
                              availableDate->?Date]:Doctor},

                   between(?Date, ?From_date, ?To_date)) ,

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

LISTING 2 Web service specification for making appointments

```
        Doctor [|  doctorName     =>  \symbol ,
                   specialty      =>  Specialty ,
                   hospitalName   =>  \symbol ,
                   availableDate  =>  \integer  |].

        doctor1 [  doctorName     ->  william ,
                   specialty      ->  ophthalmology ,
                   hospitalName   ->  IstanbulHospital ,
                   availableDate  ->{18,  19,  20,  21}  ]:Doctor.

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

        doctor3 [  doctorName     ->  maryam ,
                   specialty      ->  gynecology  ,
                   hospitalName   ->  IstanbulHospital ,
                   availableDate  ->  {3,  4,  5,  6,18}  ]:Doctor.

        hospital(IstanbulHospital ,      Turkey).
        hospital(MontpellierHospital ,  France).
```

LISTING 3 Local ontology of the web service for making appointments

3.2 | Sample goal for consuming an appointment service

We have in Listing 4 a goal for consuming an appointment making service. The goal gives specific information about the desired appointment, such as the required specialty, hospital name, the age of the patient, and the desired date range. As part of the precondition, the fact that the patient `ashley` lives in `Istanbul` is given. This fact will be used to deduce that `ashley` lives in `Turkey` and can only make an appointment at a hospital in Turkey through the web service whose specification is given in Listing 2. The appointment requester is picky about the returned result: The hospital should not be `Bosphorus` or `Dardanelle`. This requirement is enforced in the postcondition of the goal.

We note how logic variables link the preconditions and postconditions of 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.

```
goal1[inputs->
        ${startG(goal1)[
                 pre->( ${  rqObj[  specialty->gynecology ,
                                    patientName->ashley ,
                                    age->28,
                                                from_date  ->  15,
                                    to_date  ->20  ]:RequestAppointment  },

                         ${lives_in_city(ashley ,Istanbul)}        ),

                 post->( ${?_[  appointmentDate->?Date ,
                                doctorName->?DoctorName ,
                            hospitalName->?HospitalName  ]:Appointment  },

                         \+  ((?HospitalName  =  Bosphorus)  ;
                          (?HospitalName  =  Dardanelle)  )  ]  }  ]:Goal.
```

LISTING 4 Goal for consuming the appointment making web service

3.3 | Common ontology

The common ontology, given in Listing 5, contains constraints that must hold true to have a valid knowledge base, the concepts `Appointment` and `RequestAppointment` whose instances are used in the preconditions and postconditions, the concept `Specialty` and its three instances `gynecology`, `ophthal-mology`, and `otolaryngology`, utility predicates `greater`, `less`, `is_equal` and `between`, factual information about which city is in which country through the predicate `city_country`, as well as a logic rule, which relates people to their countries, based on the city they live in, defined in the predicate `lives_in_country`.

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, or vice versa. 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.

```
RequestAppointment[|  specialty    => Specialty,
                      patientName  => \symbol,
                      age          => \integer,
                      from_date    => \integer,
                      to_date      => \integer   |].

Appointment[|  appointmentDate => \integer,
               doctorName      => \symbol,
               hospitalName    => \symbol  |].

Specialty[||].
gynecology:      Specialty.
ophthalmology:   Specialty.
otolaryngology:  Specialty.

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

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).

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

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

LISTING 5 Part of the common ontology used by goals and web services

4 | IMPLEMENTATION OF THE MATCHMAKER AGENT IN FLORA-2

In this section, we describe the implementation of the matchmaker for FLOG4SWS in some detail. First, we have an overview of Flora-2, followed by its specific features that were made use of in our implementation. Then, we describe the configuration (meta-data) file. This is followed by a description of the top level loop in the matcher and the specifics of proving the commitments that are needed for a successful match. Constraint checking, an important part of keeping the ontology in a "healthy" state, is explained next. Finally, we demonstrate how our choice of a sublanguage of F-Logic results in an efficient proof procedure, which is the main workhorse of the matching process.

4.1 | Overview of Flora-2

The proposed intelligent agent for semantic web service matching uses the Flora-2 reasoning engine. Flora-2 is 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, HiLog, and Transaction Logic (Bonner & Kifer, 1998, 1994) into tabled Prolog code

(Kifer, Yang, Wan, & Zhao, 2018). Flora-2 supports a programming language that is a dialect of F-Logic and has numerous extensions, such as 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 Wan, Grosof, Kifer, Fodor, and Liang (2009). 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, were inspired by Florid (May, 2000).

4.2 | 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 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 core parts of the implementation of our matcher in Flora-2. As mentioned before, the programme does not implement the proof commitments regarding the assumptions and effects of web services, but they are similar in essence to the implemented proof commitments for preconditions and postconditions and extending our program to deal with them is straightforward.

4.3 | Meta-data file

Information about the files in which the goal, web service, and the common ontology are stored should be specified in a meta-data file. An example meta-data file is given in Listing 6.

```
%WebServicesFile('C:\\Users\\bayram\\OneDrive\\ErgoCode\\Matcher\\goal1').
%GoalsFile('C:\\Users\\bayram\\OneDrive\\ErgoCode\\Matcher\\ws1').
%CommonOntology('C:\\Users\\bayram\\OneDrive\\ErgoCode\\Matcher\\CommonOntology').
```

LISTING 6 Meta-data file containing facts about goals, web services and the common ontology

4.4 | Entry point into the matcher

Listing 7 contains the predicate `%go` that is the entry point into the matcher. The percent sign in the predicate name means that the result of this predicate call should not be tabled. We preferred a nontabled implementation in order to avoid any possible complications caused by tabling.

```
1   %go(?Result):-
2
3       %initModules(),
4       %metaData(?MetaDataPath),
5       [?MetaDataPath>>UtilModule],
6
7       %WebServicesFile(?WSFile)@UtilModule,
8       [+?WSFile>>RingModule],
9
10      %GoalsFile(?GFile)@UtilModule,
11      [+?GFile>>RingModule],
12
13      %CommonOntology(?OntologyFile)@UtilModule,
14      [+?OntologyFile>>RingModule],
15
16      %run(?Result).
```

LISTING 7 Entry point into the matcher

In the body of the `%go` predicate definition, the two used modules are emptied through a call to `%initModules` (Line 3); meta-data is loaded into UtilModule (Lines 4 and 5); web services, goals, and common ontology are loaded into Ringmodule (Lines 7–11); and finally, the `%run` predicate is called (Line 16) with `?Result` as a parameter. The result of the matching operation is a list of (goal, web service) pairs that match each other and is returned in the `?Result` parameter. The module name RingModule was inspired by boxing matches that take place inside a ring.

4.5 | Pairing goals and web services for matching

In Listing 8, we have the definition for the `%run` predicate. Using the built-in `setof` construct of Flora-2, which implicitly backtracks on the goals on the right hand side of the "such that" (|) clause, each (goal, web service) pair is tested for a match, and the successful ones are returned in the `?SuccessfulMatches` variable. Matching is performed through the `%match_once predicate`.

```
%run(?SuccessfulMatches):-
    ?SuccessfulMatches =
        setof{ ?S | ?S=(?GoalName,?WSName),
        (?GoalName:Goal)@RingModule,
        (?WSName:WebService)@RingModule,
        %match_once(?GoalName,?WSName, RingModule) }.
```

LISTING 8 Pairing up goals and web services

4.6 | Proving the commitments for a successful match

In Listing 9, we have the code that attempts to prove the commitments specified in Section 2.4.

The predicate `%match_once` (Lines 1 and 2) makes sure that the `%match` predicate is called only one time, and no backtracking is performed on it if it succeeds. `%match` retrieves the precondition and postcondition of the web service and goal using the meta-unification operator % (Lines 7–11). The goal precondition is inserted transactionally using the built-in `t_insert` predicate into the module where the match operation will take place (Line 13), and the web service precondition is proven, establishing the proof commitment *co goal.ont ws.ont goal.pre* ⊨ *ws.pre.*

The act of proving *ws.pre* binds variables in *ws.pre*, and if those variables occur in *ws.post*, they are bound in *ws.post* as well. Because the execution of the web service, by definition, guarantees the validity of the implication *ws.pre* ⇒ *ws.post*, *goal.post* is inserted into the module (Line 17), effectively implementing the modus ponens rule of logic. Then, *goal.post* is proven (Line 19), verifying the proof commitment *co goal.ont ws.ont goal.pre ws.pre* ⇒ *ws.post* ⊨ *goal.post*. The fact that insertions are made transactionally guarantees that they will be automatically deleted when another (goal, web service) pair is tried (upon backtracking) in the `%run` predicate.

Note that the predicate `%check_constraints` is called to check that the ontology itself is not already violating any constraints (Line 5) and after insertions into the *ring* module to verify that the new state does not violate any constraints either (Lines 14 and 18).

```
 1  %match_once(?GoalName, ?WSName, RingModule):-
 2                %match(?GoalName, ?WSName, RingModule),!.
 3
 4  %match(?Goal, ?WebService, ?Ring):-
 5     %check_constraints(?Ring, no_violation),
 6
 7     ?WebService[inputs->?TempW]@?Ring,
 8     ?TempW ~ ${startW(?_)[pre->?WsPre, post->?WsPost]}@?Ring,
 9
10     ?Goal[inputs->?TempG]@?Ring,
11     ?TempG ~ ${startG(?_)[pre->?GoalPre, post->?GoalPost]}@?Ring,
12
13     t_insert{?GoalPre}@?Ring,
14     %check_constraints(?Ring, no_violation),
15     %prove(?WsPre,?Ring),
16
17     t_insert{?WsPost}@?Ring,
18     %check_constraints(?Ring, no_violation),
19     %prove(?GoalPost,?Ring).
20
21  %prove((?X,?Y),?Ring):- !, %prove(?X,?Ring), %prove(?Y,?Ring).
22
23  %prove((?X;?Y), ?Ring ):- !, (%prove(?X, ?Ring ); %prove(?Y, ?Ring)).
24
25  %prove(\+(?X), ?Ring ):- !, \+ %prove(?X, ?Ring).
26
27  %prove( ?X, ?Ring ):- ?X@?Ring.
```

LISTING 9 Proving the commitments for a match

4.7 | Checking constraints

Listing 10 contains the code for checking constraint violations. When a constraint in an ontology (whether local or common) is violated, an error message is printed, and the constraint violation causes the match to fail. The `%check_constraints` predicate (Lines 1–4) takes in its first parameter the module where the matching operation is taking place, and returns in its second parameter either `violation` or `no_violation`, depending on whether a constraint has been violated or not. The `%check_constraints2` predicate (Lines 6–10) takes a list of constraint labels in its first parameter, the module where the matching operation is taking place in its third parameter, and using predicate `%check_constraint` returns in its second parameter a list of tagged constraint names, showing whether each constraint is violated or not. `succesful(c)` means constraint `c` was not violated, and `failure(c)` means it was violated. Finally, the predicate `%verify_results` (Lines 18–27) iterates over the result of `%check_constraints2`, both printing violated constraints, and also returning in its third parameter the result of the verification, that is, `violation` or `no_violation`.

```
1   %check_constraints(?Ring, ?Status):-
2      %constraints(?C)@?Ring,
3      %check_constraints2(?C,?R,?Ring),
4      %verify_results(?R, no_violation, ?Status).
5
6   %check_constraints2([],[],?_WsModule).
7
8   %check_constraints2([?H|?T],[?F|?R],?Ring):-
9      %check_constraint(?H,?F,?Ring),
10     %check_constraints2(?T,?R,?Ring).
11
12  %check_constraint(?Cons,successful(?Cons),?Ring):-
13     %constraint(?Cons)@?Ring.
14
15  %check_constraint(?Cons,failure(?Cons),?Ring):-
16     \naf %constraint(?Cons)@?Ring.
17
18  %verify_results([],?StatusIn, ?StatusOut):-
19     ?StatusOut = ?StatusIn.
20
21  %verify_results([successful(?_C)|?T], ?StatusIn, ?StatusOut):-
22     %verify_results(?T, ?StatusIn, ?StatusOut).
23
24  %verify_results([failure(?C)|?T], ?_StatusIn, ?StatusOut):-
25     write('*** Constraint violation!: ')@\prolog,
26     writeln(?C)@\prolog,
27     %verify_results(?T, violation, ?StatusOut).
```

LISTING 10 Checking constraints for violations

4.8 | Efficiency of the matching process

The main work of matching is done mainly through the predicate `%prove`, which takes two parameters. The first parameter is a logical expression ("goal" in the sense of Prolog) to be proved against a knowledge base, contained in the module given in the second parameter. The knowledge base consists only of positive facts or instances (or both), as well as definitions coming from local and common ontologies. The logical expression is allowed to have logical connectives *and* (in Flora-2 syntax, "`,`"), *or* (in Flora-2 syntax, "`;`") and *not* (in Flora-2 syntax, "`\+`"), facts and concept instances where all free variables are implicitly *existentially* quantified. The semantics of `\+` is "negation as finite failure."

The `%prove` predicate is called twice by the `%match` predicate: first to prove the web service precondition, given the initial knowledge base consisting of the facts and instances provided by the goal (in the WSMO sense) precondition, as well as the contents of the local and common ontologies, and the second time to prove the goal (in the WSMO sense) postcondition, against a possibly updated set of positive facts and instances due to the changes caused by the execution of web service.

Given a first parameter that is a simple fact (in the Prolog sense), the call to `%prove(..)` will be proven (or disproven) in one step if the knowledge base does not contain a definition of that predicate *with a body*. If its first parameter is an object with k attributes, due to the fact that in Flora-2, an object $ida_1 \rightarrow v_1, a_2 \rightarrow v_2, \ldots, a_k \rightarrow v_k$ is represented internally as a set of k predicates $a_1id, v_1,\ a_2id, v_2,\ \ldots,\ a_kid, v_k$, proving $ida_1 \rightarrow v_1, a_2 \rightarrow v_2, \ldots, a_k \rightarrow v_k$ means proving $a_1id, v_1,\ a_2id, v_2,\ \ldots,\ a_kid, v_k$, which requires at most k steps (we assume that attribute values are not computed through rule definitions but are given directly, as would be expected when the goal provides the data in its precondition).

If the first parameter is a logical expression containing connectives *and*, *or* and *not*, `%prove(..)` calls itself recursively (Lines 21–25 in Listing 9). A simple induction on the number of connectives n in the logic expression shows that in the worst case, there are $2n + 1$ calls to `%prove(..)`.

TABLE 1 Input parameters to the generator

#	Parameter	Identifier	Role
1	Number of goals	?howManyGoals	Determines how many goals should be created.
2	Number of web services	?howManyWebServices	Determines how many web services should be created.
3	Name of goals	?goalName	Determines the prefix of goal object names.
4	Name of web services	?wsName	Determines the prefix of web service object names.
5	Depth of logic tree	?Level	Determines the depth of the logic tree representing *ws.pre* and *goal.post* conditions.
6	Percentage of positive terms	?PosPercentage	Determines the probability of using the negation operator (\+) in front of sub-expressions in pre and post conditions. If ?PosPercentage is 100 means there is no negation operator at all.
7	Percentage of conjunctions	?ConjPercentage	Determines the probability of using conjunction (,) in the inner nodes of the logic tree. For example, if ?ConjPercentage is 70, that means 70% of inner nodes in the logic tree are expected to be (,) and 30% are expected to be (;) .
8	Number of conjunctions	?howManyConj	The number of facts/objects in *goal.pre* and *ws.post*. Should always equal $2^{?Level}$ due to the way goals and web services are constructed.
9	Number of attributes	?howManyAttributes	Determines how many attributes each object in pre and post–conditions has.

Even assuming that *each* call to `%prove(..)` tries to prove the existence of an object with k attributes, we have an upper bound of $k(2n - 1)$ proofs of simple facts for a call to `%prove(..)`. We thus conclude that the time complexity of `%prove(..)` is linear with respect to the number of logical connectives in the logical expression that is its first parameter.

In the ontologies, one is free to define Flora-2 predicates and constraints on the state of the ontology, without any limitation. If such definitions are present, and calls are made to the definitions (implicitly in the case of constraints, explicitly inside a web service precondition or goal postcondition), it is natural that the above computation does not necessarily apply. In such a case, it is the responsibility of the web service author to make sure that the definitions included in the ontologies do not cause inefficiency.

5 | BENCHMARKING THE MATCHER

In this section, we present the results and analysis of benchmarking our matcher in terms of time. Benchmarking was performed on a Windows 7 64-bit 2.86GHz platform with 8GB of RAM using Ergo Suite version 1.2 (Solon). Ergo Suite is a commercial product based on the Flora-2 system, developed and marketed by Coherent Systems.[1] First, the methodology and test sets we used are explained. Then, the results are described and analysed to see if they fit the expectations. As can be seen in the sections below, the matching operation is linear with respect to both logical complexity, as well as the number of web services that are matched against, demonstrating the scalability of our approach.

5.1 | Parameters to the benchmark generator

The specific aim of the benchmarking effort has been to determine the scalability of our logic-based matching scheme in two dimensions: (a) complexity of logical expressions used in the pre and postconditions of goals and web services and (b) number of service descriptions against which a specific goal is matched. To achieve this aim, we developed a benchmark generator (Bayram et al., 2019), implemented in Flora-2, whose task is to produce sample goals and web services based on nine input parameters. These input parameters are summarized in Table 1. We define a configuration as a 9-tuple consisting of the parameters 1 to 9.

5.2 | Sample generated goals and web services for benchmarking

In Listing 11, we have a sample goal and in Listing 12 a sample web service generated for the configuration (1, 1, gg, ww, 2, 50, 50, 4, 2).

[1]http://coherentknowledge.com/

```
gg1[inputs->${startG(gg1)[
  pre->(
    ${rqObj4[a2->v_in,a1->v_in]:InConcept},
    ${rqObj3[a2->v_in,a1->v_in]:InConcept},
    ${rqObj2[a2->v_in,a1->v_in]:InConcept},
    ${rqObj1[a2->v_in,a1->v_in]:InConcept}),
  post->(
    (\+  (${respObj1[b2->v_out,b1->v_out]:OutConcept};
    (\+  ${respObj2[b2->v_out,b1->v_out]:OutConcept}))),
    ${respObj3[b2->v_out,b1->v_out]:OutConcept},
    (\+  ${respObj4[b2->v_out,b1->v_out]:OutConcept}))
]}]:Goal.
```

LISTING 11 Sample generated goal for configuration set {1, 1, gg, ww, 2, 50, 50, 4, 2}

```
ww1[inputs->${startW(ww1)[
    pre->(
      (((\+  ${rqObj1[a2->v_in,a1->v_in]:InConcept});
      ${rqObj2[a2->v_in,a1->v_in]:InConcept});
      (\+  ${rqObj3[a2->v_in,a1->v_in]:InConcept}),
      ${rqObj4[a2->v_in,a1->v_in]:InConcept})),
    post->(
      ${respObj4[b2->v_out,b1->v_out]:OutConcept},
      ${respObj3[b2->v_out,b1->v_out]:OutConcept},
      ${respObj2[b2->v_out,b1->v_out]:OutConcept},
      ${respObj1[b2->v_out,b1->v_out]:OutConcept})
    ]}]:WebService.
```

LISTING 12 Sample generated web service for configuration set {1, 1, gg, ww, 2, 50, 50, 4, 2}

5.3 | Code modifications to measure execution time

To measure the exact time needed by Flora-2 inference engine for finding a complete match against a set of web services, we modified the main predicate of the matcher as shown in Listing 13. At Line 12, we use XSB epoch_miliseconds predicate to get the current time of the system. At Line 14, the %run predicate is called to find the matching web services. At Line 16, again the current time of the system is obtained, and at Line 18, the difference between the two times is calculated in milliseconds and written into an output file (Lines 20–22).

```
1                %go(?Result):-
2                  %initModules(),
3                  %metaData(?MetaDataPath),
4                   [?MetaDataPath>>UtilModule],
5                  %WebServicesFile(?WSFile)@UtilModule,
6                   [+?WSFile>>RingModule],
7                  %GoalsFile(?GFile)@UtilModule,
8                   [+?GFile>>RingModule],
9                  %CommonOntology(?OntologyFile)@UtilModule,
10                [+?OntologyFile>>RingModule],
11
12                epoch_milliseconds(?S1,?MS1)@\prolog(machine),
13
14               %run(?Result),
15
16                epoch_milliseconds(?S2,?MS2)@\prolog(machine),
17
18               %duration(?S2,?MS2,?S1,?MS1,?DS,?DMS),
19
20                ?Filename = './out.txt',
21                ?Filename[open(append,?Stream)]@\io,
22                ?Stream[writeln([?DS,?DMS])]@\io,  ?Stream[close]@\io.
```

LISTING 13 Modified version of the main predicate used for benchmarking the match operation

5.4 | Test configurations and results

We defined sets of configurations to test the efficiency of the matcher for its scalability in terms of number of web services, as well as their complexity. Test configuration sets were divided into three categories.

The first category contains configurations, which are different only in the number of web services. Configurations sets 1 to 11 given in Table 2 are in the form of {1,X,2,100,100,4,4}, {1,X,2,100,4,3}, and {1,X,3,100,8,3} wherein X can be 50, 100, 500, and 1,000 (in last two sets).

The last column in Table 2 shows the average time of five independent runs of the configuration. It is expected that as the number of web services increases, the time of inferencing increases linearly. The results show that the expectation is true indeed. Figure 2 depicts the time trends of configurations 1 to 3, 4 to 7, and 8 to 11.

The second category contains the configuration sets, which are different in logical complexity of the pre and postconditions of both goal and web services. We consider the logical complexity of Flora-2 expressions used in pre and postconditions as the number of logical operators multiplied by the number of attributes in each frame (this is due to the fact that in Flora-2 a frame with *n* attributes is internally represented as conjunctions of *n* binary predicates). It is expected that by increasing the logical complexity, average time increases linearly. Table 3 gives the actual timing data obtained in the test runs, which are also depicted graphically in Figures 3 and 4. It can be clearly seen that the average inference time trend is again linear.

TABLE 2 Configuration sets and average time spent for matching

#	# of goals	# of web services	# of Level	%Pos.	%Conj.	# of Conj.	# of Attr.	Avg. time (ms)
1	1	50	2	100	100	4	4	182.8
2	1	100	2	100	100	4	4	336.8
3	1	500	2	100	100	4	4	1,741.2
4	1	50	2	100	100	4	3	144
5	1	100	2	100	100	4	3	281
6	1	500	2	100	100	4	3	1,477.2
7	1	1000	2	100	100	4	3	3,101
8	1	50	3	100	100	8	3	246.6
9	1	100	3	100	100	8	3	493
10	1	500	3	100	100	8	3	2,611.6
11	1	1000	3	100	100	8	3	5,765.8

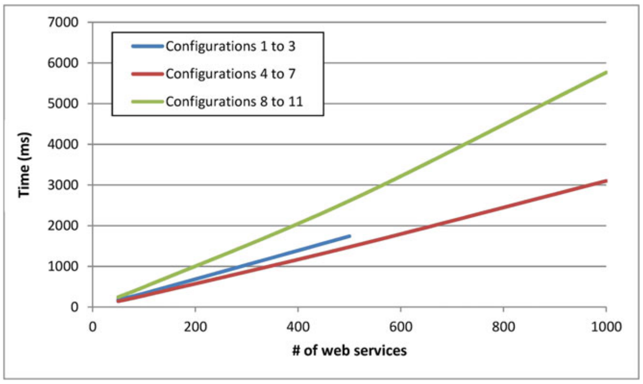

FIGURE 2 Running time trends of configurations 1 to 11 (different number of web services)

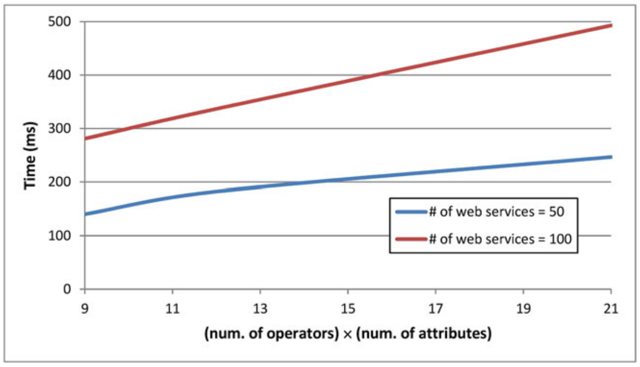

FIGURE 3 Graph of matching time versus logical complexity (50 web services and 100 web services)

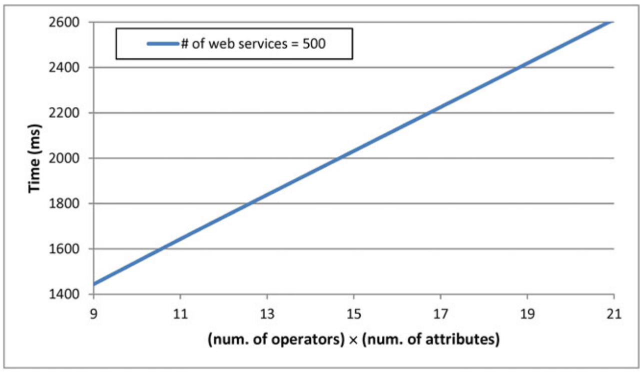

FIGURE 4 Graph of matching time versus logical complexity (500 web services)

TABLE 3 Average matching time for different logical complexities

		# of web services		
		50	**100**	**500**
# of operators × # of attributes	**9**	140.4ms	281ms	1,444.2ms
	12	182.8ms	336.8ms	1,741.2ms
	21	246.6ms	493ms	2,611.6ms

TABLE 4 Average matching time at different conjunctive and positive percentages (200 web services)

	Positive %				
Conjunctive %	**0%**	**25%**	**50%**	**75%**	**100%**
0%	720.8	780	620.8	539.8	480.6
25%	592.8	570.8	639.6	1067	627
50%	502.4	496	556.4	798.8	686.8
75%	436.8	449.2	486.8	568	777
100%	368	386.8	383.8	443	986

The last configuration sets are different only in positive and conjunction percentages (Table 4). The number of attributes is fixed at 3.

Figure 5 shows the timing of the configuration sets as a 3-D surface. It can be seen that the time dimension has local peaks when the conjunctive percentage is either 25% or 100%, and the positive percentage is either 75% or 100%, respectively. We consider this an interesting observation, without any implications with respect to scalability.

6 | RELATED WORK ON MATCHING

In previous work (Ataee & Bayram, 2015), we presented a way of representing web service and goal specifications in Flora-2 by using the built-in `if-then-else` construct, as well as the `insert` command of Flora-2. We also described simple matcher that can check whether a single goal specification matches a single web service specification. No consistency checking of the ontologies was available. In this work, we have a much more comprehensive and systematic approach, which can deal with multiple web service and goal descriptions, has checking for ontology consistency, does not make explicit use of the `insert` command or the `if-then-else` construct of Flora-2 in the goal and service specifications, and has been shown to be efficient both analytically and through benchmarking.

Roman, Kopeckỳ, Vitvar, Domingue, and Fensel (2015) describe WSMO-Lite, a lightweight ontology of web service semantics based on a bottom-up specification called SAWSDL, which is the first SWS standard produced by the World Wide Web Consortium in 2007. WSMO-lite makes use of four semantic aspects of services, that is, function, behaviour, information model, and nonfunctional properties as the basis for semantic automation. Information semantics are represented using domain ontologies (as in WSMO). Functional semantics are represented with capabilities (preconditions and effects) and/or functionality classifications (not available in WSMO) using some classification ontology. Nonfunctional semantics are represented using an ontology (in WSMO, this is a component of lower level constructs, not the web service as a whole). Of special interest is the way they represent behavioural semantics through annotating the service operations with functional descriptions, that is, capabilities and/or functionality classifications, providing the means to schedule calls to operations. These roughly correspond to choreography specifications of WSMO, where rule activations during choreography execution are mapped to operation calls but are more explicit and straightforward. They also include support for combining RESTful services with WSDL-based ones in a single semantic framework through two HTML microformats that mirror WSDL and SAWSDL in the documentation of RESTful services.

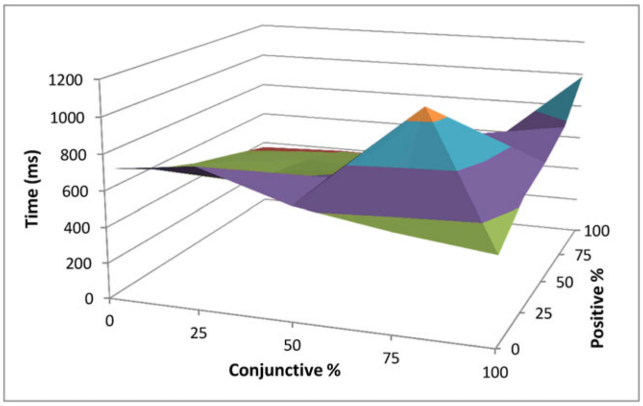

FIGURE 5 Running time trends of configurations, which are different only in conjunction and positive percentages

WSMO-lite is a fundamentally bottom-up approach, aiming for adoption in industrial web service settings. FLOG4SWS, on the other hand, represents a top-down, purely logic-based approach that has the potential to be developed further to incorporate aspects of SWS specification, which it does not currently have. The behavioural semantics of WSMO-lite for example would fit very nicely into our logical framework in the context of choreography specification.

Baklouti, Gargouri, and Jmaiel (2015) propose an extension of OWL-S in order to improve description and discovery of web services in linguistic information systems. Their idea is to integrate nonfunctional linguistic properties and their relationships inside the advertisements and semantically annotate some elements of the linguistic web service descriptions. Being based on OWL-S, highly specialized and restricted to only non-functional properties means that their work is not directly comparable with ours.

Semantic Web Service Discovery using natural language processing techniques is proposed by Sangers, Frasincar, Hogenboom, and Chepegin (2013). Queries posed in natural language are analyzed and matched against web service descriptions specified using WSML (specific type not stated), and then a search using various algorithms is performed. Again their work is not directly comparable to ours, but the two are complementary, since their Semantic Web Service Discovery engine can be adapted to accept FLOG4WS as the "back end" instead of WSML.

An application of semantic web services using OWL-S, as well as a hybrid matching algorithm used for service discovery is presented by Zapater, Escrivá, García, and Durá (2015). The application area is "allowing travelers to find road traffic information web service (WSs) that best fits their requirements." Their matching algorithm is set-based and not directly comparable with ours.

Kifer, Lara, Polleres, and Zhao (2004) propose a discovery method using proof commitments in transactional logic. Because they place no restrictions on the logic statements that can take part in the preconditions and postconditions of web service capabilities or goal specifications; however, it can be expected that a very heavy burden is placed on the reasoning engine. 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. 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 hold between them. Consequently, we have been able to build an actual, practical implementation for our matching agent.

In Paolucci, Kawamura, Payne, and Sycara (2002b); Sycara, Paolucci, Ankolekar, and Srinivasan (2003); Srinivasan, Paolucci, and Sycara (2005, 2004); Kawamura et al. (2003), 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 similar work, a way to map DAML-S service profiles into UDDI records and using the encoded information to perform semantic matching is described by Paolucci, Kawamura, Payne, and Sycara (2002a). 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. Because 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 by Bener, Ozadali, and Ilhan (2009) 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.

Sycara, Klusch, Widoff, and Lu (1999) 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," and "outconstraints." Their matchmaking algorithm determines the relationship among two semantic descriptions by computing the respective subsumption relationship and is again set-based.

Rambold, Kasinger, Lautenbacher, and Bauer (2009) provide a high-level survey and ranking of web service discovery methods but do not give any details regarding the specific matching algorithm used by the surveyed methods.

Srinivasan et al. (2004) 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 by Celik and Elci (2006).

A matchmaking algorithm based on bipartite graphs for semantic web services specified in OWL-S is presented by Ilhan and Bener (2007). The approach carries out semantic similarity assignment using subsumption, properties, similarity distance annotations, and WordNet. No logical inference is carried out, except for subsumption.

Di Noia, Di Sciascio, and Donini (2007) formalize the matching problem in general using Description Logic (DL), and devise "Concept Abduction" and "Concept Contraction" as nonmonotonic inferences in DL for modelling matchmaking in a logical framework. They also give algorithms for semantic matchmaking based on the devised inferences as well. Similarly, Grimm, Motik, and Preist (2006) use a framework for annotating web services using DL , and they show how to realise service discovery by matching semantic service descriptions, applying DL inferencing. DL is a very limited form of logic, compared with the first-order logic, and matching based on DL specifications is not directly comparable with our work, which deals with F-Logic specifications.

Della Valle and Cerizza (2005) 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 by Klusch and Kaufer (2009), 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 research has started to shift towards "more scalable and approximative rather than computationally expensive logic-based reasoning with impractical assumptions" Paydar and Kahani (2014). 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, Chabeb, Tata, and Ozanne (2010) describe 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 by Fan, Ren, and Xiong (2005). Küster and König-Ries (2008) propose a graded relevance scale for semantic web services matchmaking as measurements to evaluate SWS matchmakers based on such graded relevance scales.

7 | CONCLUSION AND FUTURE RESEARCH DIRECTIONS

Using a sublanguage of F-Logic to specify ontologies, web services and goals, and Flora-2 as the implementation tool, we 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, reification and module facilities of Flora-2, as well as its built-in inference engine. Because 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 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. Furthermore, we demonstrated the scalability of our approach analytically and also by benchmarking the matching operation under various scenarios.

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.

Our sublanguage of F-Logic used for specifying web services and goals in logical terms is open to further development. For future work, we envisage the addition of a choreography component based on abstract state machines, as well as a mediation component to FLOG4SWS and incorporating these into the matchmaker without affecting the efficiency of the proof procedure employed in it. Given the purely logical specification of web services and goals, a logical inference-based composition of web services to achieve the functionality required by a goal looks like a reasonable proposition as well. Yet, another direction in which this research can be advanced is the definition of 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. Such an implementation could have the potential to make WSML a viable alternative in industry for semantic web service specification.

ACKNOWLEDGEMENTS

We express our appreciation to Coherent Knowledge, LLC[2] for donating a free license of Ergo Suite version 1.1, which is a commercial product compatible with Flora-2, for use in this project.

CONFLICT OF INTEREST

None.

ORCID

Zeki Bayram https://orcid.org/0000-0003-4878-357X

REFERENCES

Ataee, S. M., & Bayram, Z. (2015). A novel concise specification and efficient F-Logic based matching of semantic web services in flora-2. In *Information Sciences and Systems 2015 - 30th International Symposium on Computer and Information Sciences, ISCIS 2015*, London, UK, pp. 191–198. https://doi.org/10.1007/978-3-319-22635-4_17

Baklouti, N., Gargouri, B., & Jmaiel, M. (2015). Semantic-based approach to improve the description and the discovery of linguistic web services. *Engineering Applications of Artificial Intelligence*, *46*, 154–165.

Bayram, Z., & Sharifi, O. (2016). Unifying F-logic molecules: A rectification to the original unification algorithm. *Journal of Logic and Computation*, *26*, 1043–1049.

Bayram, Z., Sharifi, O., & Ataee, S. (2019). FLOG4SWS-v2 semantic matcher for web services and goals using Flora-2. https://sourceforge.net/projects/flog4sws-v2/. Accessed: 2019-01-24.

Bener, A. B., Ozadali, V., & Ilhan, E. S. (2009). Semantic matchmaker with precondition and effect matching using SWRL. *Expert Systems with Applications*, *36*, 9371–9377.

Bonner, A. J., & Kifer, M. (1994). An overview of transaction logic. *Theoretical Computer Science*, *133*, 205–265.

Bonner, A., & Kifer, M. (1998). A logic for programming database transactions. In *Logics for Databases and Information Systems (The Book Grew Out of the Dagstuhl Seminar 9529: Role of Logics in Information Systems, Dagstuhl, Wadern, Germany, 1995)*, pp. 117–166.

Celik, D., & Elci, A. (2006). 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*, *2*, IEEE, Chicago, USA, pp. 273–278.

Chabeb, Y., Tata, S., & Ozanne, A. (2010). Yasa-m: A semantic web service matchmaker. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, IEEE, Perth, Australia, pp. 966–973.

de Bruijn, J (2008). WSML language reference, deliverable d16. 1v1. 0.

Della Valle, E., & Cerizza, D. (2005). Cocoon glue: A prototype of WSMO discovery engine for the healthcare field. In *Proceedings of the 2nd WSMO Implementation Workshop WIW* Innsbruck, Austria, pp. 1–12.

Di Noia, T., Di Sciascio, E., & Donini, F. M. (2007). Semantic matchmaking as non-monotonic reasoning: A description logic approach. *Journal of Artificial Intelligence Research*, *29*, 269–307.

Fan, J., Ren, B., & Xiong, L.-R. (2005). An approach to web service discovery based on the semantics, *Fuzzy systems and knowledge discovery* (pp. 1103–1106): Springer: Changsha, China.

Fensel, D., & Bussler, C. (2002). The web service modeling framework WSMF. *Electronic Commerce Research and Applications*, *1*, 113–137.

Fensel, D., Lausen, H., Polleres, A., de Bruijn, J., Stollberg, M., Roman, D., & Dominigue, J. (2007). *Enabling semantic web services: The web service modeling ontology*. Heidelberg: Springer.

Gallier, J. H. (1985). *Logic for computer science: Foundations of automatic theorem proving*. New York: Harper & Row Publishers, Inc.

Grimm, S., Motik, B., & Preist, C. (2006). Matching semantic service descriptions with local closed-world reasoning, *The semantic web: Research and applications* (pp. 575–589). Heidelberg: Springer.

Ilhan, E., & Bener, A. (2007). Improved service ranking and scoring: Semantic advanced matchmaker (SAM) architecture. In *Evaluation of Novel Approaches to Software Engineering (ENASE 2007)*, Barcelona, Spain, pp. 95–102.

Kawamura, T., De Blasio, J.-A., Hasegawa, T., Paolucci, M., & Sycara, K. (2003). Preliminary report of public experiment of semantic service matchmaker with UDDI business registry, *Service-oriented computing-ICSOC 2003* (pp. 208–224). Heidelberg: Springer.

Keller, U., Lara, R., Polleres, A., Toma, I., Kifer, M., & Fensel, D. (2004). WSMO web service discovery. http://www.wsmo.org/2004/d5/d5.1/v0.1/. Accessed: 2019-01-24.

Kifer, M., Lara, R., Polleres, A., & Zhao, C. (2004). A logical framework for web service discovery. In *ISWC 2004 Semantic Web Services Workshop, CEUR Workshop Proceedings*, Hiroshima, Japan.

Kifer, M., & Lausen, G. (1989). F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *ACM SIGMOD Record*, *18*, ACM, Poland, pp. 134–146.

Kifer, M., Lausen, G., & Wu, J. (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the ACM (JACM)*, *42*, 741–843.

[2]http://www.coherentknowledge.com/

Kifer, M., Yang, G., Wan, H., & Zhao, C. (2018). Ergolite (a.k.a. flora-2): User's manual. http://flora.sourceforge.net/docs/floraManual.pdf. Accessed: 2019-01-24.

Klusch, M., & Kaufer, F. (2009). WSMO-MX: A hybrid semantic web service matchmaker. *Web Intelligence and Agent Systems*, *7*, 23–42.

Küster, U., & König-Ries, B. (2008). Evaluating semantic web service matchmaking effectiveness based on graded relevance. In *Proceedings of the Second International Conference on Service Matchmaking and Resource Retrieval in the Semantic Web*, SMRR'08, *416*, CEUR-WS.org, Aachen, Germany, Germany, pp. 32–46. http://dl.acm.org/citation.cfm?id=2889945.2889949

May, W. (2000). How to write F-Logic programs in Florid. Institut für Informatik, Universität Freiburg. http://dbis.informatik.uni-freiburg.de/content/projects/Florid/florid_tutorial.pdf. Accessed: 2019-02-15.

Object Management Group (2016). Meta Object Facility (MOF) Core. http://www.omg.org/spec/MOF/. Accessed: 2019-01-24.

Paolucci, M., Kawamura, T., Payne, T. R., & Sycara, K. (2002a). Importing the semantic web in UDDI, *Web services, e-business, and the semantic web* (pp. 225–236). London, UK: Springer.

Paolucci, M., Kawamura, T., Payne, T. R., & Sycara, K. P. (2002b). Semantic matching of web services capabilities. In *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, Sardinia, Italy, pp. 333–347. https://doi.org/10.1007/3-540-48005-6_26

Paydar, S., & Kahani, M. (2014). A semantic web enabled approach to reuse functional requirements models in web engineering. *Automated Software Engineering*, *22*, 1–48.

Rambold, M., Kasinger, H., Lautenbacher, F., & Bauer, B. (2009). Towards autonomic service discovery a survey and comparison. In *2009 IEEE International Conference on Services Computing*, Bangalore, India, pp. 192–201.

Roman, D., Kopeckỳ, J., Vitvar, T., Domingue, J., & Fensel, D. (2015). WSMO-Lite and hRESTS: Lightweight semantic annotations for web services and RESTful APIs. *Web Semantics: Science, Services and Agents on the World Wide Web*, *31*, 39–58.

Sangers, J., Frasincar, F., Hogenboom, F., & Chepegin, V. (2013). Semantic web service discovery using natural language processing techniques. *Expert Systems with Applications*, *40*, 4660–4671.

Srinivasan, N., Paolucci, M., & Sycara, K. P. (2004). An efficient algorithm for OWL-S based semantic search in UDDI. In *Semantic Web Services and Web Process Composition, First International Workshop, SWSWPC 2004, San Diego, CA, USA, July 6, 2004, Revised Selected Papers*, San Diego, CA, USA, pp. 96–110. https://doi.org/10.1007/978-3-540-30581-1_9

Srinivasan, N., Paolucci, M., & Sycara, K. (2005). An efficient algorithm for OWL-S based semantic search in UDDI. In Cardoso, J., & Sheth, A. (Eds.), *Swswpc'04: Proceedings of the First International Conference on semantic Web Services and Web Process Composition*. Berlin, Heidelberg: Springer-Verlag, pp. 96-110.

Sycara, K. P., Klusch, M., Widoff, S., & Lu, J. (1999). Dynamic service matchmaking among agents in open information environments. *Sigmod Record*, *28*, 47–53.

Sycara, K., Paolucci, M., Ankolekar, A., & Srinivasan, N. (2003). Automated discovery, interaction and composition of semantic web services. *Web Semantics: Science, Services and Agents on the World Wide Web*, *1*, 27–46.

Wan, H., Grosof, B., Kifer, M., Fodor, P., & Liang, S. (2009). Logic programming with defaults and argumentation theories, *Logic programming* (pp. 432–448). Heidelberg: Springer.

Zapater, J. J. S., Escrivá, D. M. L., García, F. R. S., & Durá, J. J. M. (2015). Semantic web service discovery system for road traffic information services. *Expert Systems with Applications*, *42*, 3833–3842.

AUTHOR BIOGRAPHIES

O. Sharifi received his Ph.D. degree from the Department of Computer Engineering, Eastern Mediterranean University, North Cyprus in 2014. Currently, he is an assistant professor in the Department of Computer and Software engineering, Toros University, Turkey. His current research interests include biometrics, multimodal fusion, and semantic web services.

S. Mehdipour Ataee received his BS and MS degrees in Software Engineering from Mazandaran University of Science and Technology in 2006 and Qazvin Azad University in 2010, respectively. He received his Ph.D. in Computer Engineering from Eastern Mediterranean University in 2018. He is currently an assistant professor in the Faculty of Engineering at Final International University (North Cyprus). His current research interests include semantic web reasoning and process mining.

Z. Bayram received his BS degree in Economics from the Wharton School of the University of Pennsylvania (USA) in 1987, MS, and Ph.D. degrees in Computer and Information Sciences from the University of Alabama at Birmingham (USA) in 1988 and 1993, respectively. He is currently an associate professor at the Computer Engineering Department of Eastern Mediterranean University (North Cyprus). His current research interest include automated deduction, expert systems, Frame Logic, constraint logic programming, and semantic web services.

How to cite this article: Sharifi O, Ataee SM, Bayram Z. Frame Logic-based specification and discovery of semantic web services with application to medical appointments. *Expert Systems*. 2019;e12387. https://doi.org/10.1111/exsy.12387

APPENDIX A: SYNTAX OF THE SUBLANGUAGE OF F-LOGIC USED FOR SPECIFICATION OF GOALS, WEB SERVICES, AND ONTOLOGIES

In Listing 14, 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 Kifer et al. (2018). 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, because 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 object identifier. The nonterminal "Term" is used to define both predicates and terms in the sense of Prolog.

Note the difference between facts and queries: inside facts, object identifiers must be ground terms, whereas inside queries, they must be variables. The reason is that the provider of the fact knows the name of the fact, whereas the query does not care about the actual name of the provided object, but rather its contents and the concept it belongs to.

Also note that objects *always* have to be reified (i.e., enclosed inside `${}`). On the other hand, predicates that are inside queries (web service precondition and goal postcondition) need not be reified. Inside a goal precondition and web service postcondition, they *do* need to be reified, just like objects.

The logical connective *and* is represented by a comma (,), *or* is represented by a semi-colon (;), and *not* is represented by `\+` as in Flora-2.

```
Goal ::= ID '[inputs->${startG[ pre ->' Fact ',
                                    post->' TopLevelQuery
                                 ']
                           }
].'
WebService ::= ID '[inputs->${startW[  pre->' TopLevelQuery ',
                                         post->' Fact
                                      ']
                                 }
                  ].'
Ontology ::= { AnyValidFlora2Definition | ConstraintRule }
ConstraintRule ::= 'constraint(' ID ') :-' Body '.'
Fact ::= Simple_Fact | Fact_List
Simpe_Fact ::= '${' ObjectSpecification1 '}' | '${' Term '}'
Fact_List ::=  '(' Simpe_Fact {',' Simple_Fact } ')'
ObjectSpecification1 ::= ID '[' [ SpecBody ] ']' [':' Concept]
TopLevelQuery ::= SimpleQuery | '(' ComplexQuery ')'
SimpleQuery ::= '${'ObjectSpecification2 '}' | Term
ComplexQuery ::=    ( SimpleQuery |  NegationQuery )
                    { ',' ComplexQuery | ';' ComplexQuery }
                  | '(' ComplexQuery ')'
NegationQuery ::=    \+ '(' SimpleQuery ')'
                   | \+ '(' NegationQuery ')'
                   | \+ '(' '(' ComplexQuery ')' ')'
ObjectSpecification2 ::= LogicVar '['[SpecBody]']' [':'Concept]
SpecBody ::=  ID '->' Value { ',' ID '->' Value }
Value ::= LogicVar | ID | Number | Term
Number ::= Any numeric value
LogicVar ::= Any logic variable of Flora-2
ID ::= Any zero-ary function symbol in the sense of Prolog
Term ::= Any term in the sense of Prolog
Body ::= Any allowed rule body in Flora-2
AnyValidFlora2Definition ::= Any valid single Flora-2 definition
Concept ::= Any valid concept name in Flora-2
```

LISTING 14 EBNF grammar for FLOG4SWS, the sub-language of F-Logic used for specifying goals, web services and ontologies