# A New Logic-Based Approach for the Specification and Discovery of Semantic Web Services

**Omid Sharifi**

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the Degree of

Doctor of Philosophy
in
Computer Engineering

Eastern Mediterranean University
September 2014
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Elvan Yılmaz
Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Doctor of Philosophy in Computer Engineering.

Prof. Dr. Işık Aybay
Chair, Department of Computer Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Doctor of Philosophy in Computer Engineering.

Assoc. Prof. Dr. Zeki Bayram
Supervisor

Examining Committee

1. Prof. Dr. Rashad Aliyev
2. Prof. Dr. Yalçın Çebi
3. Prof. Dr. Can Özturan
4. Assoc. Prof. Dr. Zeki Bayram
5. Assoc. Prof. Dr. Alexander Chefranov

# 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, in this thesis 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 and matching engine using the FLORA-2 system. In order to be able to specify Semantic Web elements such as Web services, goals, ontologies, we define a sub-language of FLORA-2, which we call FLOG4SWS. The result is a practical, fully implemented matching engine and discovery agent 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.

Before going to the implementation of new language (FLOG4SWS) and logical inference based discovery agent we investigate the strong as well as weak aspects of WSML in order to guide us in the search for a better alternative. In our studies into the theory of F-Logic, we discovered a mistake in the unification algorithm for F-Logic molecules, and we present a corrected version of the algorithm in this thesis as well.

# ÖZ

Ağ hizmetlerini ve hedefler şeklinde belirtilmiş kullanıcı gereksinimlerini eşleştirmek, semantik ağ hizmetleri keşfinde yapılması kolay olmayan bir şeydir. Eşleştirmede en çok kullanılan, ancak tatminkar olmayan yaklaşım, küme tabanlı olandır. Bu yaklaşımda, hem kullanıcı, hem de ağ hizmeti istedikleri ve ihtiyaç duydukları nesneleri deklare ederler. Böylece, eşleştirme basit nesne kümeleri karşılaştırmasına dönüşür. Bu yaklaşım, kullanıcının ihtiyacı olan işlevsellik, veya hizmetin sunduğu işlevsellik hakkında hiç bir şey söylememesinden dolayı yetersizdir. Bu tezde, küme tabanlı eşleştirmeye alternatif olarak, FLORA-2 mantık sisteminde gerçekleştirildiği şekliyle F-Logic dilini kullanarak, hizmet yeteneklerini ve kullanıcı isteklerini mantık ifadeleri şeklinde belirtip, eşleştirmenin mantıksal çıkarım açısından ne anlama geldiğini açıkça tanımlayıp, FLORA-2 sistemini kullanan mantık tabanlı bir keşif ajanı ve eşleştirme makinesi gerçekleştiriyoruz. Ağ hizmetleri, hedefler ve ontolojiler gibi semantik ağ elemanlarını belirtebilmek için, FLOG4SWS adını verdiğimiz bir FLORA-2 alt dili tanımlıyoruz. Sonuç olarak ortaya çıkan gerçekleştirilmesi tamamlanmış, tamamen mantıksal çıkarıma dayalı, F-Logic ile olan yakın ilişkilerinden dolayı Web Service Modeling Ontology (WSMO) ve Web Service Modeling Language (WSML) diline doğrudan uyarlanabilen bir eşleştirme makinesi ve keşif ajanıdır.

Tezde ayrıca, yeni dil (FLOG4SWS) ve mantıksal çıkarım tabanlı keşif ajanının gerçekleştirmesine geçmeden once, bize daha iyi bir alternatifin yolunu göstermesi açısından, WSML'in kuvvetli ve zayıf yönlerini araştırdık. F-Logic kuramı araştırmalarımız esnasında, F-Logic moleküllerinin birleştirme algoritmasında bir hata keşfettik ve bu algoritmanın düzeltilmiş şeklini de sunuyoruz.

# ACKNOWLEDGMENT

First of all I would like to express my special and deepest appreciation to my supervisor Assoc. Prof. Dr. Zeki Bayram for his advice and guidance over the past five years. During this period I have benefited from his intelligent effort to shape my thinking and working attitude. I would like to thank him one time more for encouraging my research; his advice on the research has been invaluable. I would like to thank members of my PhD committee, Prof. Dr. Rashad Aliyev and Assoc. Prof. Dr. Alexander Chefronov for their constructive and helpful comments, suggestions and career advice on this thesis. Also many thanks go in particular to my parents for their continuous support and prayers. Words can not express how grateful I am. I would also like to thank to my beloved wife, Maryam, thank you for supporting me for everything specially for encouraging me during this experience. To my beloved son Radvin, I would like to express my thanks for being such a good boy always cheering me up. I dedicate this thesis to my wife and our son for their support and help. Finally, I would like to thank my sister and brothers for their love and their thoughtful support.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| **ASM** | Abstract State Machines |
| **DCE** | Distributed Computing Environment |
| **EBNF** | Extended Backus Normal Form |
| **HTTP** | Hypertext Transfer Protocol |
| **OWL** | Web Ontology Language |
| **RDF** | Resource Description Framework |
| **RPC** | Remote Procedure Call |
| **SOAP** | Simple Object Access Protocol |
| **SW** | Semantic Web |
| **SWS** | Semantic Web Service |
| **W3C** | World Wide Web Consortium |
| **WSDL** | Web Service Description Language |
| **WSMF** | Web Service Modeling Framework |
| **WSML** | Web Service Modeling Language |
| **WSMO** | Web Service Modeling Ontology |
| **WSMT** | Web Services Modeling Toolkit |
| **WWW** | World Wide Web |
| **XML** | Extensible Markup Language |

# Chapter 1

# INTRODUCTION

Given a semantically rich enough description of Web service capabilities and client requirements, Semantic Web service (SWS) 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 [48] as our starting point in our discussion of Semantic Web service discovery, as it allows us to formally distinguish between user requests (called "goals" is WSMO terminology) and Web service specifications, and use the same formalism for specifying both. WSMO itself is based on the Web Service Modeling Frame-work (WSMF) [53] and has four main components, which are *Ontologies*, *Web Services*, *Goals* and *Mediators*.

Web Service Modeling Language (WSML) [45] 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 [61]- 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 [45].

There are several discovery approaches used in WSML [71]. 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 [92].

Different kinds of *match* have been defined for the the set based approach [71]. These are summarized below.

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

The problem with the set based approach, no matter what underlying logic is used to represent sets of objects, is that even if the set of objects requested in the goal are indeed returned by the service, there is no guarantee that the desired operation has been performed to obtain 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) be powerful enough to be effectively specify goals and Web service capabilities.

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

1. We specify a sub-language of F-logic with implicit existential and universal quantifiers (depending on where the formula is used) that permits efficient goal-directed deduction, as in the case of logic programming,

2. We clearly state the proof commitments (in terms of logical entailment) necessary for a successful match, and finally

3. We implement a logical entailment based (intelligent) matching agent using the FLORA-2 system, demonstrating not only the feasibility of our approach, but also its practicality.

We chose F-logic [74] 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 [74], 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.

We outline the structure of the thesis below.

Chapter 2 represents the context of the thesis and denotes the main definitions of the Semantic Web components, which are needed in specification, and discovery of Semantic Web services. For this, the chapter reviews the existing Web service technologies as well as the SWS languages and frameworks.

Chapter 3 presents in-depth the definition of WSMO, its formalism, the WSML, its execution environment and modeling toolkit Web Service Modeling Toolkit (WSMT) [30]. It contains an in-depth study of WSML and experimentation to identify its strong points and areas in which improvement would be beneficial.

Chapter 4 contains the main contribution of this thesis. In this chapter, we define a carefully selected sub-language of F-Logic for specifying Semantic Web services that permits efficient inferencing to be carried out, a precise notion of matching based on proof commitments, and proceed to describe an implementation of an intelligent matchmaking agent for the discovery of Semantic Web services.

Chapter 5 presents a correct version to the original unification algorithm for unifying F-Logic molecules that was discovered in our studies into the theory of F-Logic.

Finally, Chapter 6 summarizes the thesis and discusses the contributions, as well as future research directions.

# Chapter 2

# THE WORLD WIDE WEB AND WEB SERVICES

This chapter presents the basic idea of Semantic Web and Semantic Web services matching, along with other World Wide Web (WWW) technologies that our research is based on.

## 2.1 The World Wide Web (WWW)

The WWW is a system of interlinked hypertext documents accessed via the Internet. For the first time it was proposed by Tim Berners-Lee in 1989 at CERN (European Organization for Nuclear Research) to use “HyperText ... to link and access information of various kinds as a Web of nodes in which the user can browse at will” [16]. The first invented Web browser in 1990 was used for displaying Hypertext Markup Language (HTML) documents. WWW changed the way information was published and broadcast, however, information was not dynamic, being updated only by the webmaster [21].

The early stage of the conceptual evolution of the WWW was Web 1.0, which only supported published HTML Web pages containing static information. Users were just able to receive the Web contents without interaction abilities with the websites to send and receive feedback. After Web 1.0, a new version of the WWW, Web 2.0, was suggested which allows users and websites to interact and communicate with each other [22]. Finally, the new generation of WWW is Web 3.0, the most important features of which are the Semantic Web and personalization [33]. It supports data transmission with the ability of unambiguous and shared meaning in computer systems. At the same time, it provides semantic interoperability that enables machine computable logic, knowledge discovery, inferencing and data federation between information systems [3].

## 2.2 Web Services and Web Service Technologies

The concept of Web services evolved from the Remote Procedure Call (RPC) mechanism in Distributed Computing Environments (DCE), a framework for software development that emerged in the early 1990s but mostly developed in the late 1990s [20]. A Web service is a method of communications between two electronic devices over WWW. It is a software function provided at a network address over the Web or the cloud; it is a service that is "always on" as in the concept of utility computing [23].

The World Wide Web Consortium (W3C) defines a "Web service" as a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically Web Services Description Language (WSDL) [24]). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards [29].

Usually Web service frameworks are composed of four main elements, namely communication protocols, service descriptions, service invocation, and service discovery. Description of Web services are made available by service providers to service consumers through WSDL. Service consumers obtain related Web services information through UDDI, and detailed information about specific Web services through their description in WSDL. Subsequently, interactions between service providers and consumers can be established using SOAP [1], or more recently REST [8]. More details about the three important components of Web service frameworks (WSDL, SOAP, and UDDI) are presented in the subsequent sections.

## 2.3 Web Service Description Language (WSDL)

WSDL is an XML-based language for describing the functionality offered by a Web service. A Web service specification in WSDL supplies a machine-readable description "of how the service can be called, what parameters it expects, and what data structures it returns" [26]. Essentially, a WSDL description provides a set of endpoints operating on messages including either procedure-oriented or document-oriented information. The described abstractly operations and messages are bounded to a message format and concrete network protocol to define an endpoint and the related endpoints constitute abstract endpoints where the abstract endpoints form services. WSDL provides the ability to define endpoints regardless of network protocols or message formats [24].

A defined Web service in WSDL includes various elements. “Type” is a container in order to define data type using some type system (such as XSD). “Message” is an abstract typed definition for data communication. “Operation” provides an abstract description of an action supported by the service. “Port Type” presents an abstract set of operations supported by one or more endpoints. “Binding” supplies a concrete protocol and data format specification for a particular port type. “Port” defines a single endpoint as a combination of a binding and a network address , service, a collection of related endpoints [27].”

## 2.4 Simple Object Access Protocol (SOAP)

SOAP is a lightweight protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on XML Information set for its message format, most notably Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission and forms the foundation layer of other Application Layer protocols [13, 14]. In summary SOAP is a communication protocol, which is used for communication between applications. Some important features of SOAP are platform independency, communication via Internet, language independency, simplicity, and extensibility [13].

## 2.5 Universal Description, Discovery and Integration Protocol (UDDI)

UDDI is a platform-independent, Extensible Markup Language (XML)-based registry where businesses world wide can list themselves on the Internet. It provides a mechanism to register and locate Web service applications to enable companies to find each other on the Web and make their systems interoperable for e-commerce [19, 17]. In other words, UDDI enables businesses to publish service listings using name, location, product, or the Web services they offer. In addition, it enables businesses to discover each other, and determines the procedure of services or software applications interaction over the Internet [19]. UDDI was originally aimed as a core Web service standard to be investigated by SOAP messages in order to access to WSDL documents presenting the protocol bindings and message formats needed to interact with the Web services listed in its directory [19, 31].

## 2.6 Web Services Discovery

Web service discovery is the process of locating Web services that can be retrieved to fulfill some users' requests [78]. Software systems gain access to the published Web services over the Internet using standard protocols [28]. In this area, two types of service discovery are used namely, static and dynamic. In static discovery, the details of services implementation are bound at design time and service retrieval is performed on a service registry. In dynamic discovery, the service implementation details are left unbound at design time, therefore the details determined could be postponed to run time. In the dynamic approach, the Web service requesters determine their preferences in order to invoke the application to infer/reason desired Web services [7].

## 2.7 Semantic Web Services

SWS (Semantic Web Services), like conventional Web services, are the server end of a client-server system for machine-to-machine interaction via the WWW. Semantic services are a component of the Semantic Web because they use markup which makes data machine-readable in a detailed and sophisticated way where human-readable HTML is not simply understood by computer programs [11].

In fact, the SWS goal is to overcome the deficiencies of the conventional technologies of Web service, particularly for the service detection and usability analysis. It provides rich semantic annotation ability for Web service descriptions so that inference-based techniques for automating the detection and usage of Web services are provided [82, 53]. A wide range of research topics were developed on SWS technologies in order to increase the discovery and reasoning techniques to detect proper Web services. Also composition ability was developed to combine several Web services to answer a more complex task, and mediation was developed to solve the heterogeneity problem between the requester and the provider. Therefore, SWS method develops descriptions of Web services in order to use ontologies instead of

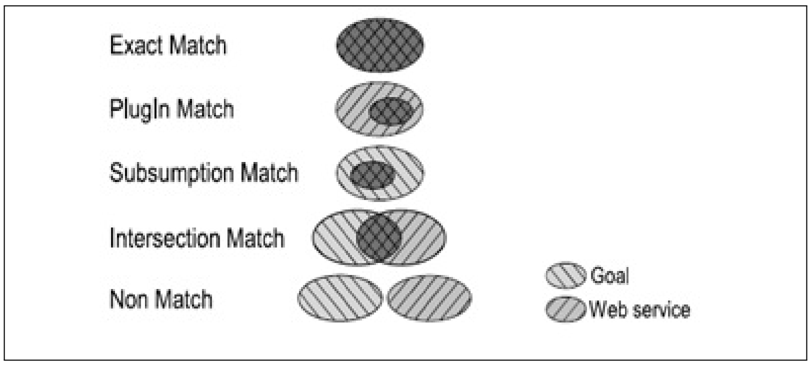


Figure 2.1: Matchmaking concepts in semantic discovery

XML, apply functional along with non-functional information, define interface for consumption to support automated compatibility, and apply aggregation Web services to increase its functionality by combining several other Web services [62].

## 2.8 Web Services Matching

Matching is a very important process for discovery and retrieval of Web services. Matching approaches help Web service discovery engines to match abstracted goal description with semantic annotations of Web services.

In this area, matching goals and Web services is most important task in the Web services discovery. Mostly in the matching process logical relationships between the semantic capability descriptions of Web services and goals are considered. Different degrees of logical relationships represent various levels of matching, which can be categorized in five levels such as Exact Match, plugin Match, Subsumption Match, intersection Match, and Non Match. Depending on the degree of the relationship, suitable Web services are represented as answers. Figure 2.1 demonstrates the five matching notions [55].

## 2.9 The Semantic Web and Ontologies

The Semantic Web (SW) is the extension of the current Web (WWW) that for the first time was introduced by Tim Berners-Lee . Figure 2.2 represents "Semantic Web layer cake" including core elements of overall structure of the Semantic Web, as proposed by Tim Berners-Lee. The Semantic Web enables people to share content beyond the boundaries of applications and websites. By including the semantic content in the Web pages, Semantic Web converts the current Web, dominated by unstructured and

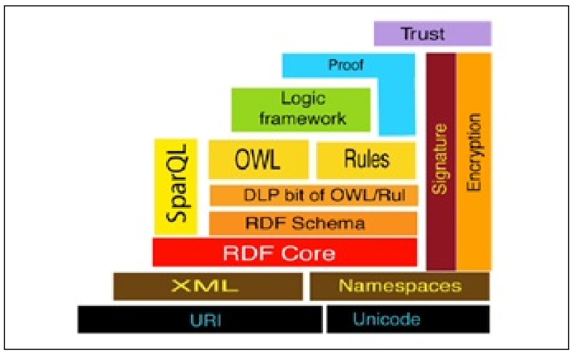


Figure 2.2: Layer cake of the Semantic Web according to Tim Berners-Lee

semi-structured documents into a “Web of data” [10, 12].

XML documents are not able to convey the meaning of data contained in the XML documents. It is necessary for the parties to have agreement on the exact syntactical format (expressed in XML Schema) in exchanging of data. In fact, Semantic Web enables the representation and exchange of information in a meaningful way, facilitating automated processing of descriptions on the Web.

In Semantic Web area, annotations state links between information resources on the Web. Information resources are connected to formal terminologies called ontologies. Ontologies provide machine understanding of information by way of the links between the information resources and the terms in the ontologies. In addition, they enable interoperation between information resources through links to the same ontology or links between ontologies. Concisely it can be stated that ontology is a formal explicit specification of a shared conceptualization [55].

## 2.10 Semantic Web Service Technologies

Consideration of some techniques in SWS such as discovery to detect proper Web services for a requested task, composition to combine several Web services in order to fulfill more complex tasks, mediation to deal with heterogeneities, and automated execution of Web services can represent most important means to detect services and usability analysis phase. Discovery involves the detection of Web services that are suitable for a requested goal. In this area, usually the base of Web service discovery is on matching abstracted goal descriptions with semantic annotations of Web services where this process is done on an ontological level. Composition is related with combination of several Web services to achieve a more complex functionality to deal with solving a client request at the time it is needed. The resources needed to process the request may be heterogeneous, so that mediation can handle and solve the heterogeneities, which may hamper the interoperability between a requester and a provider. Automated execution provides ability to execute Web services automatically in way to minimize the need for human intervention after all the necessary Web services have been discovered, composed, and mediated successfully [93, 70].

In Semantic Web services area some languages and frameworks were defined to implement and develop Semantic Web Services. Here follows some of these frameworks and languages [12]. The explanation of some the frameworks and languages are represented in the following section.

Semantic Web languages:

- Ontology Inference Layer (OIL)
- DARPA Agent Markup Language (DAML)
- DAML+OIL
- Web Ontology Language (OWL)
- Resource Description Framework (RDF)
- Web Services Modeling Language (WSML)
- Web Services Semantics (WSDL-S)
- SAWSDL
- Rule Based Service Level Agreements (RBSLA based on RuleML)

Semantic Web Service frameworks:

- WSMF
- OWL-S
- QuASAR
- WSMO
- IRS-III
- METEOR-S
- HALEY
- BioMOBY (Bioinformatics)

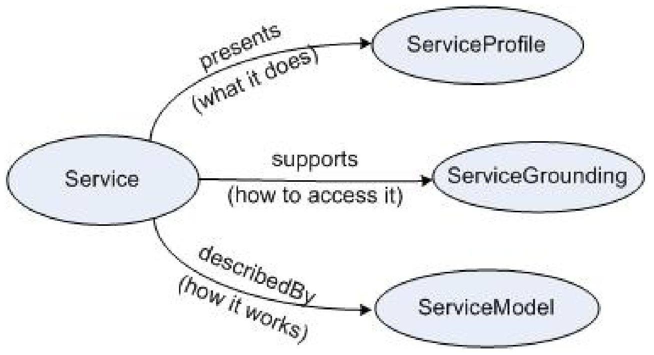


Figure 2.3: Conceptual models of OWL-S (taken from [87])

- SSWAP

## 2.11 Semantic Web Service Frameworks

The Semantic Web Services Framework (SWSF) [37] is an early development in the direction of a Semantic Web service annotation framework. WSMO, OWL-S [80], SWSF [37], and WSDL-S [34] are some other existing frameworks to define comprehensive specifications for semantically describing Web services. Below, we give an overview of the each conceptual SWS framework. In the next chapter, we give a detailed exposition and evaluation of WSMO, since our research is closely related to it.

## 2.12 OWL-S

OWL-S [80] as the first chronologically approach was defined to describe various aspects of Web services. It is developed based on the DAML Service Ontology (DAMLS), and is the first progressive attempt for semantic annotation of Web services. In other words, It provides an upper ontology for semantically annotating Web services which includes some elements as presented in Figure 2.3 (taken from [80]).

In the top of ontology “Service” concept is located and the three subontologies of OWL-S, namely “Service Profile”, “Service Model”, and “Service Grounding”, are defined. Service Profile prepares information for service advertisement, Service Model states how the Web service works, and Service Grounding indicates the way to access the service in detail.

## 2.13 WSDL-S

The WSDL-S [34] approach has been proposed based on a lightweight mechanism to include Web service descriptions semantically in WSDL. It includes semantic annotations to the XML data types, messages and operations in a WSDL description. The WSDL document is contained with some extra tags to point to an external domain ontology. Three types of annotations can be considered for WSDL-S independent of the use ontology languages, such as WSDL types to refer to concepts of the domain ontology, WSDL operations to be described by using preconditions and effects, and defining a categorization of Web services based on the ontology taxonomy.

## 2.14 SAWSDL

To describe the interface of Web services at a syntactic level and invocation, generally WSDL is used. This description, as already mentioned, does not provide semantic means to describe the actual service functionality, or other relevant aspects. The syntactic description is concerned with the structure of input and output messages of an interface and invocation of the services. Semantic annotations define some mechanisms for WSDL and XML Schema (SAWSDL) in order to add semantic annotations to WSDL components. The annotations can be used to classify, discover, match, compose, and invoke Web services [35].

## 2.15 Discussion

The Semantic Web is widely seen as the new generation of the current Web, and new technologies are constantly being introduced in accordance with the vision of Tim Berners-Lee to make the Web comprehensible to machines.

## 2.16 Conclusion

In this chapter we introduced the reader to current Semantic Web service technologies.

# Chapter 3

# WSMO AND WSML: A DETAILED INVESTIGATION

To complete the investigation of Semantic Web services technology, this chapter introduces WSML language, based on WSMO framework that is a large and highly complex framework designed for the specification of Semantic Web services. In this chapter, we also perform an in-depth study of WSML and experiment with it in order to critically evaluate it by identifying its strong points and areas in which improvement would be beneficial. Our experimentation takes the form of the specification of a Web service in the area of e-health.

## 3.1 Motivation and Overview

The goal of Web services is to allow normally incompatible applications to interoperate over the Web regardless of language, platform, or operating system [68]. Web services are much like remote procedure calls, but they are invoked using Internet and World Wide Web (WWW) [41] standards and protocols such as Simple Object Access Protocol (SOAP) [1] and Hypertext Transfer Protocol (HTTP) [2]. In order to use a Web service, it must first be discovered, and its member functions called in the correct order and with the correct number and type of arguments.

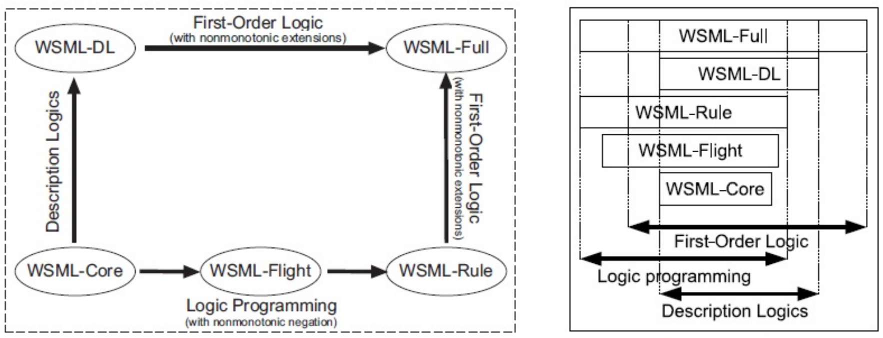
Currently, the interface to a Web service is described in a Web Service Description Language (WSDL) [24] document. WSDL allows the specification of function signatures so that methods of the Web service described by the WSDL document can be called with the correct number and type of parameters. However, there is no formal semantics in WSDL, i.e. *what* the Web service is for is not described in the WSDL document in a formal way.

Discovery of Web services is made possible through Universal Description Discovery and Integration (UDDI) [18], which allows organizations to describe their businesses and services, to discover other businesses that offer desired services, and to integrate with these other businesses [68]. Once a Web service is discovered and the decision taken to make use of it, automated tools can read the corresponding WSDL file and generate local classes that the programmer can use to invoke the Web methods as if they were methods of local objects in the program.

Although some level of automation is available once a Web service is discovered, the full process of discovery and invocation is still largely manual and is on the shoulders of the programmer. This is due to the fact that the standards and technologies mentioned above enable the description of Web services at the syntactic level only and contain no formal semantic specification that is machine-processable. Any description of the semantics of a Web service has to be done using natural language, which currently cannot be reliably processed by machines, and this necessitates the involvement of programmer in the discovery and invocation of Web services. Semantic Web services aim to rectify this drawback by making use of semantic Web standards such as Resource Description Framework (RDF) [6], Web Ontology Language (OWL) [4] as well as logic to provide machine processable semantic descriptions such that automatic discovery and invocation of Web services, with no or minimal programmer involvement, becomes possible.

Web Service Modelling Ontology (WSMO) [25] is a comprehensive framework for describing Web services, goals (high-level queries for finding Web services), mediators (mappings for resolving heterogeneities) and ontologies. Web Services Modeling Language (WSML) [32] is a *family* of concrete languages based on F-logic [74] that implement the WSMO framework. The variants of WSML are WSML-core, WSML-Flight, WSML-Rule, WSML-DL, and WSML-Full. WSML is large, relatively complex, and somewhat confusing, with different variants being based on different formalisms. The complexity and confusion arise mainly from the many variants of the language, and the rules used to define the variants. The variants of WSML form a hierarchy, shown in Figure 3.1 (taken from [65]), with WSML-Full being on top (the most powerful) and WSML-core being at the bottom (weakest).

Our literature search has failed to reveal any significant industrial real-life ap-



(a) Language variants (b) Layering

Figure 3.1: WSML language hierarchy (taken from [65])

plication that uses WSML. We believe this is due to the inherent complexity of the language, the "less-than-complete" state of WSML (e.g. the syntax of WSML-DL does not conform to the usual description logic syntax, choreography specification using Abstract State Machines (ASM) [51] seems unfit for the job due to the execution semantics of ASMs, goals, choreographies and Web services are not integrated in the same logical framework etc.), as well as the lack of proper development tools and execution environments. So WSML looks like it is still in a "work-in-progress" state, rather than a finished product.

It is our conviction however that WSML has a strong basic foundation based on F-logic which we can use to build a much smaller, reasonably simple, self-contained logic based language with well-understood semantics for semantically describing Web services. WSML-rule, which can be processed using (non-monotonic) rule reasoners, and is compatible with the Stable Model Semantics for logic programs [47] is a promising place to start for the new language. Another advantage of WSML-rule is that it has the "flavour" of Prolog [98] which has an established track record of acceptance in the computing profession.

To achieve the goal of streamlining, and at the same time enhancing WSML to the degree that it becomes a practical and usable language for the specification of Semantic Web services, we need to first determine specific areas of weakness in WSML and WSMO. We have done this in two ways: (i) through an in-depth study of the documentation provided for all parts of WSMO and WSML, and (ii) through the specification of a semantic Web service in the e-health domain (making doctor appointments in hospitals). These two activities have permitted us to be able to critically evaluate the strengths and weaknesses of WSMO and WSML, and determine the areas of improvement that will result in a usable Semantic Web service specification language. This is the main contribution of this work, which will be input to the next phase of our research, the actual design and implementation of such a language.

The reminder of this chapter is organized as follows: Section 3.2 is a brief introduction to WSMO and WSML. Section 3.3 semantically describes, using WSML-Rule, a Web service for making appointments in hospitals for patients. Also included are typical goals, and underlying ontologies. This section helps to introduce the reader to the syntax and semantics of WSML-rule in an almost tutorial manner. Section 3.4 contains the main contribution of the chapter: a critical evaluation of WSMO and WSML, including its strengths, weaknesses and deficiencies, discovered through our detailed study of the documentation provided for WSMO and WSML, as well as the specification of the e-health Web service described in section 3.3.

## 3.2 The WSMO and WSML

In this part we give a brief overview of WSMO and WSML.

### 3.2.1 WSMO

WSMO [25] is a framework for semantic description of Semantic Web services which is based on the Web Service Modelling Frame-work (WSMF) [53]. WSMO has four main components: Ontologies, Web Services, Goals and Mediators. WSMO itself is a kind of meta-ontology that defines the four main components mentioned above. Below we describe each of these components of WSMO.

#### 3.2.1.1 Ontologies

Ontology is a core element in Semantic Web [40] area. An ontology provides a common understanding of terms for different applications to use. In practical terms, we can think of an ontology as an intelligent information store where terms are defined through concepts, concept hierarchies and logical statements. A concept - described by a concept definition - provides attributes with names and types [77]. In general, conceptualization defines knowledge about the domain, not about the particular state of affairs in the domain [66].

In WSMO, ontologies that describe the relevant aspects of the domain of discourse provide the terminology used by other WSMO elements [97]. A WSMO ontology is defined using concepts, relations, functions, axioms, and instances of concepts and relations, as well as non-functional properties, imported ontologies, and used mediators.

#### 3.2.1.2 Goals

A goal specifies the desired functionality by a Web service client. It can be likened to a query in the sense of databases. The requester states what inputs it can provide to a Web service, and the results it wants from the Web service. Given a goal, it is the job of a semantic matchmaker to determine Web services that can answer the goal. A goal in WSMO is described by non-functional properties, imported ontologies, used mediators, requested capability and requested interface.

#### 3.2.1.3 Web Services

A Web service description is the mirror image of a goal in that it specifies the provided functionality to requesters. A WSMO Web service description consists of a capability, which describes the functionality in the form of preconditions, assumptions, postconditions and effects, one or more interfaces which describe the possible ways of interacting with the service, and non-functional properties, which describe non-functional aspects of the service [25].

Preconditions specify the conditions on the input data that comes from the requester. The Web service cannot be called if the preconditions are not met.

Assumptions specify the state of the world before the Web service execution can begin. If the assumption does not hold, the successful execution of the service is not guaranteed.

Postconditions specify conditions on the result data that are guaranteed to hold. The result data is provided by the Web service as a consequence of its execution. These conditions may relate inputs to the outputs as well.

Effects denote the state of the world after the successful execution of the Web service i.e., if the preconditions and the assumptions of the service are satisfied, and the Web service has finished it execution.

Capabilities can also have “shared variables,” which are universally quantified logic variables with a scope that is the whole Web service capability. The logical interpretation of a Web service capability is: for any values taken by the shared variables, the conjunction of the precondition and of the assumption implies the conjunction of the postcondition and of the effect [36].

“Choreography” is the name given to the specification of the interaction pattern between the requester and service provider. In WSMO, choreographies are defined using the formalism of Abstract State Machines [51]. State signature and the transition rules are the most important parts of the definition of the choreography. The state signature describes the state ontology used by the service, together with the definition of the types of modes the concepts and relations may have, which describes the service’s and the requester’s rights over the instances [54]. The transition rules express changes of states by changing the set of instances [60]. In practical terms, the abstract state machine operates like a forward-chained expert system shell, where the instances in a snapshot of the ontology plays the role of the contents of the working memory at a given instant.

“Orchestration” describes how the overall function of the service is realized through cooperation with other services. From a practical point of view, the requester may not necessarily be interested in how the Web service providing the functionality the requester desires does its job (i.e. what other services it makes use of in providing the service it provides).

#### 3.2.1.4 Mediators

Mediators provide the facility to solve terminology mismatches between different components of a system. Four kinds of mediators exist in WSMO: ontology-ontology mediators, Web service-Web service mediators, goal-goal mediators and Web service-goal mediators.

### 3.2.2 WSML

The Web Service Modeling Language (WSML) [65] is a *family* of languages for describing semantic Web services in conformance to the WSMO framework. WSML consists of its five variants, namely WSML-Core, WSML-DL, WSML-Flight, WSML-

Rule and WSML-Full. Each language variant provides different levels of logical expressiveness [65].

#### 3.2.2.1 WSML-Core

WSML-core is the language at the intersection point of Description Logic and Horn Logic, based on Description Logic Programs [64]. WSML-core has the least expressive power among all the languages of the WSML family, but it has the most preferable computational characteristics. Support for modelling classes, attributes, binary relations and instances are the most important features of the language, which also supports class hierarchies, relation hierarchies, datatypes and datatype predicates.

#### 3.2.2.2 WSML-Flight

The extension of WSML-Core with features such as meta-modelling, constraints and non-monotonic negation constitute WSML-flight. This language is based on a logic programming variant of F-Logic [9] and is semantically equivalent to Datalog with inequality and (locally) stratified negation.

#### 3.2.2.3 WSML-Rule

WSML-rule is an extension of WSML-Flight in the direction of Logic Programming. The language includes several extensions WSML-Flight such as function symbols and unsafe rules. The semantics for negation is based on the Stable Model Semantics of logic programs [59].

#### 3.2.2.4 WSML-DL

WSML-DL is an extension of WSML-Core which fully captures the Description Logic SHIQ(D) and a major part of the (DL species of the) Web Ontology Language OWL [5]. The syntax of the language however has been left largely unspecified, and for practical purposes, it is not a viable option for specifying semantic Web services.

#### 3.2.2.5 WSML-Full

This language incorporates WSML-DL and WSML-Rule under a First-Order logic with extensions to support the nonmonotonic negation of WSML-Rule. Since there is no widespread consensus about what is the right formalism for combining Description Logic ontologies with nonmonotonic rules, WSML does not specify a semantics for its Full variant [32]. Furthermore, since it incorporates full first order logic, drawing logical conclusions from WSML-Full programs is undecidable.

## 3.3 E-health Semantic Web Service Specification in WSML-Rule

In this section, we use WSML-Rule to define a semantic specification of a Web service which implements an appointment-making use case scenario. In this scenario, doctors specify their available times, and patients declare their choices with varying degrees of precision (e.g. they may specify only a specialty, or give the name of a specific doctor etc.), and the Web service tries to make an appointment for the patient.

Even though we had no prejudice for or against any variant of WSML when we started, WSML-Rule turned out to have the right balance of expressivity and execution properties for this job. We implemented and published an e-banking scenario based on WSML-Rule language [88]. We believe this is no coincidence, since it is closely related to Horn Clause Logic programming, which itself is a “sweet spot” between formal logic and implementability.

### 3.3.1 E-health Ontology

The e-health ontology works like an intelligent database, keeping track of the needed data in the system, together with the necessary constraints on the data. It contains concepts, relations, instances, relation instances as well as axioms related to the appointment activities.

#### 3.3.1.1 E-health Ontology Concepts Used for Modelling Data

Figure 3.2 depicts graphical view of the e-health ontology concepts that are utilized in setting up an appointment in the requested time. Below, we describe the major concepts used in describing the appointment-making Web service.

• The Doctor Concept: The Doctor concept, depicted in Figure 3.3, inherits from the Person concept, and as such has the “name” attribute by default. A doctor also has attributes “hasFreetime,” which is a list of time/day combinations at which the doctor is free, “hasSpecialty” which is the doctor’s field of specialty, and “worksAt,” the list of hospitals the doctor works at (may be more than one).

• The Hospital Concept: The Hospital concept, shown in Figure 3.4, is a subconcept of MedicalCenter and inherits the “hasCity” and “hasCountry” attributes. Its own attributes are “employsDoctor,” which is the inverse of the “worksAt” attribute of the “Doctor” concept, and “hasDepartment” which is a list of departments in the

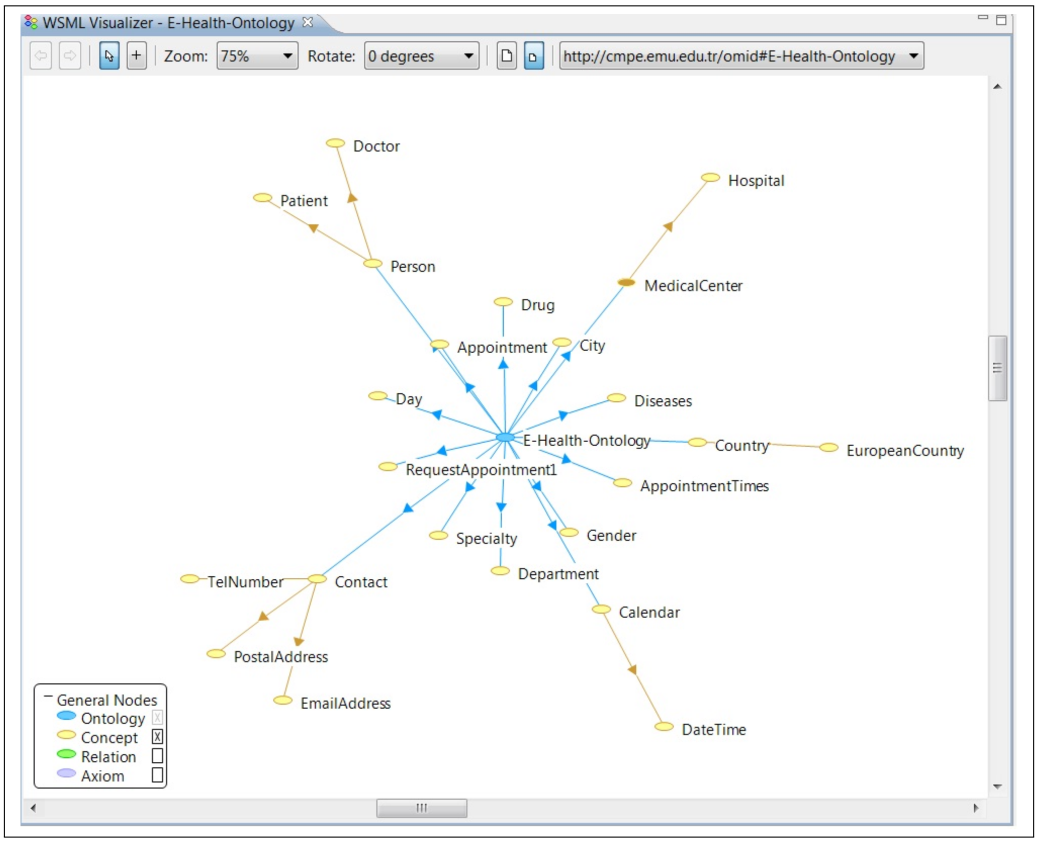


Figure 3.2: WSML visualizer showing the concepts of e-health ontology

```
concept Doctor subConceptOf Person
  worksAt inverseOf(employsDoctor) ofType Hospital
  hasSpecialty ofType Specialty
  hasFreeTime ofType DateTime
```

Figure 3.3: The Doctor concept

hospital.

#### 3.3.1.2 Utility Concepts

Calendar and DateTime Concepts: These concepts, depicted in Figure 3.5, are used for both specifying the available times of doctors, and also request times for appointments.

#### 3.3.1.3 E-Health Ontology Concepts for Passing Parameters to and Returning Values from the Web Service

• RequestAppointment Concept: Currently, this is the only concept, shown in Figure 3.6, used for passing parameters to the Web service for requesting an appointment. It has attributes for doctor, hospital, datetime, speciality as well as patient. Depending

```
concept Hospital subConceptOf MedicalCenter
  hasDepartment ofType Department
  employsDoctor inverseOf(worksAt) ofType Doctor

concept MedicalCenter
  hasCountry ofType (1 1) Country
  hasCity ofType (1 1) City
```

Figure 3.4: The Hospital concept

```
concept Calendar
  year ofType (1 1) _integer
  month ofType (1 1) _integer
  day ofType (1 1) _integer
  nameOfDay ofType (1 1) Day




concept DateTime subConceptOf Calendar
  hour ofType (1 1) _integer
```

Figure 3.5: The Calendar and DateTime concepts

on the specificity of the request, some of these fields may be left empty. For example, a patient may specify a speciality, whereas another patient may insist on a specific doctor.

- Appointment Concept: Currently, this is the only concept used for returning results to the requester. It has attributes for doctor, hospital, datetime, and patient. Figure 3.7 gives its definition.

```
concept RequestAppointment
  specialty ofType Specialty
  patient ofType Patient
  onDateTime ofType DateTime
  hospital ofType Hospital
  doctor ofType Doctor
```

Figure 3.6: The RequestAppointment concept

```
concept Appointment
     dateTime ofType   DateTime
     doctor ofType     Doctor
     patient ofType    Patient
     hospital ofType   Hospital
```

Figure 3.7: The Appointment concept

### 3.3.1.4 E-health Instances

Instances in WSMO are used to describe the state of the world in the form of objects and relations on the objects. Instances can be defined either explicitly by specifying concrete values for attributes or parameters or by a link to an instance store, i.e., an external storage of instances and their values [46, 36]. Inputs to Web services and outputs of Web services are also in the form of instances. Figure 3.8 depicts three different instances, each one of a different concept.

```
instance hurkan memberOf Doctor
  hasSpecialty hasValue otolaryngology
  worksAt hasValue magusaTipMerkezi




instance app1 memberOf Appointment
  dateTime hasValue dt2012_5_4_12_friday
  doctor hasValue hurkan
  patient hasValue alex




instance magusaTipMerkezi memberOf Hospital
  hasCountry hasValue cyprus
  hasCity hasValue famagusta
  hasDepartment hasValue {cochlear_dept , dental_dept }
```

Figure 3.8: Some instances of concepts in the e-health ontology

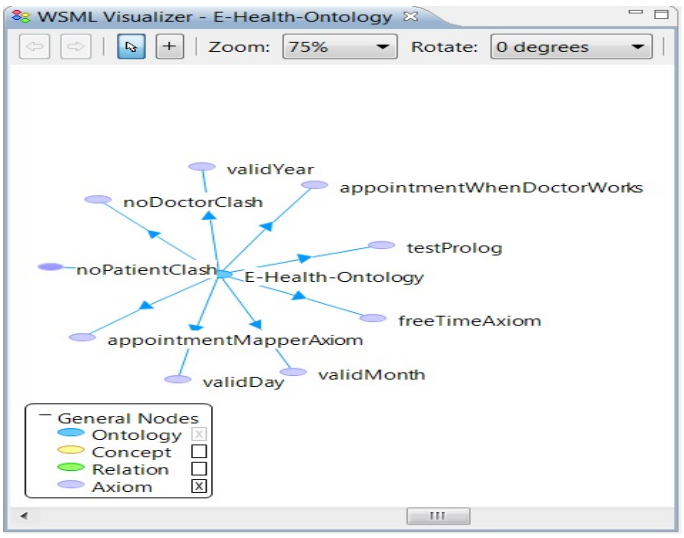


Figure 3.9: WSML visualizer showing the axioms of e-health ontology

```
axiom noDoctorClash
  definedBy
    !- appointment(?dt,?doc,?pat1)
    and appointment(?dt,?doc,?pat2)
    and ?pat1 != ?pat2.
```

Figure 3.10: The "noDoctorClash" axiom

### 3.3.1.5 E-health Ontology Axioms

Axioms form the logical part of WSML. They are used both to specify computation (in the same way Prolog clauses are used), and also to specify constraints on the state of the ontology that should not be violated.

In this section, we describe in some detail the axioms used in the e-health ontology. Figure 3.9 depicts a graphical view of the e-health axioms.

• E-health axiom "noDoctorClash": Figure 3.10 depicts the constraint that a doctor cannot be seeing two patients at exactly the same time slot. The notation "!-" denotes a condition which should *not* ever hold (like a forbidden state). This is in contrast to the database concept of constraints, where constraints are expected to always hold. The equivalent database constraint to "!- someCondition" would thus be "not someCondition."

• E-health axiom "appointmentWhenDoctorWorks": This axiom, depicted in Figure 3.11, makes sure that a doctor is never assigned to a patient at a time and day when

```
axiom appointmentWhenDoctorWorks
  definedBy
    !- appointment(?dt,?doc,?pat)
    and ?dt[
            nameOfDay hasValue ?nod,
            hour hasValue ?h
           ]memberOf DateTime
    and naf worksOn(?doc,?nod,?h).
```

Figure 3.11: The "appointmentWhenDoctorWorks" axiom (constraint)

```
axiom noPatientClash
  definedBy
    !- appointment(?dt,?doc1,?pat)
    and appointment(?dt,?doc2,?pat)
    and ?doc1 != ?doc2.
```

Figure 3.12: The "noPatientClash" axiom (constraint)

s/he is not working. The times and days a doctor works are represented explicitly in the ontology.

• E-health axiom "noPatientClash": This is the mirror image of the "noDoctor-Clash" axiom. Through this axiom, given in Figure 3.12, we are forbidding a patient having simultaneous appointments with doctors.

• E-health axiom "appointmentMapperAxiom": Being able to treat objects as instances of relations would have been very useful, but this functionality is not provided by default in WSML. In Figure 3.13 we define an axiom to do exactly this: treat instances of the "Appointment" concept as instances of the "appointment" relation (excluding the "Hospital" attribute, since this is not needed in places where the "appointment" relation is used). Now the predicate "appointment" can be used in logical expressions in the style of Prolog.

• E-health axiom "freeTimeAxiom": This axiom, depicted in Figure 3.14, defines the predicate "free" which gives the times at which a doctor is free. It is similar to the "appointmentMapperAxiom" in principle.

• Axioms for checking validity of dates: Figure 3.15 depicts axioms for making

```
axiom appointmentMapperAxiom
  definedBy
    ?a[dateTime hasValue ?dt,
       doctor hasValue ?doctor,
       patient hasValue ?patient
      ]memberOf Appointment
    implies appointment(?dt,?doctor,?patient).
```

Figure 3.13: The “appointmentMapperAxiom” axiom (mapping objects to relation instances)

```
axiom freeTimeAxiom
  definedBy
    ?d[hasFreeTime hasValue ?dt] memberOf Doctor
    equivalent
    free(?dt,?d).
```

Figure 3.14: The “freeTimeAxiom” axiom (mapping doctors to their free times)

sure that day, month and year values are all valid.

#### 3.3.1.6 E-health Ontology Relations

Relations allow the specification of explicit relationships in the from of relation instances between objects or simple data (such as integers). Relations can also be defined implicitly through axioms. Figure 3.16 and Figure 3.17 depict the relations used in the e-health ontology. These are

- “appointment,” which relates a “DateTime” value with a doctor and a patient,
- “worksOn,” which relates a doctor with the days and times s/he works, and
- “free” which records the current free times of a doctor.

### 3.3.2 E-health Web services

We have specified Web services of two hospitals for making appointments. They are very similar to one another, differing only in small details. We describe one in detail, and the second one only enough to point out the differences with the first. These examples alert us to the need of having a mechanism/method to specialize generic, published semantic Web service definitions to fit the needs of a specific Web service.

```
axiom validDay
  definedBy
    !- ?c[day hasValue ?v] memberOf Calendar
    and ?v > 31.

axiom validMonth
  definedBy
    !- ?c[month hasValue ?v] memberOf Calendar
    and ?v > 12.

axiom validYear
  definedBy
    !- ?c[year hasValue ?v] memberOf Calendar
    and ?v < 2012.
```

Figure 3.15: Axioms for validity of year, month and day

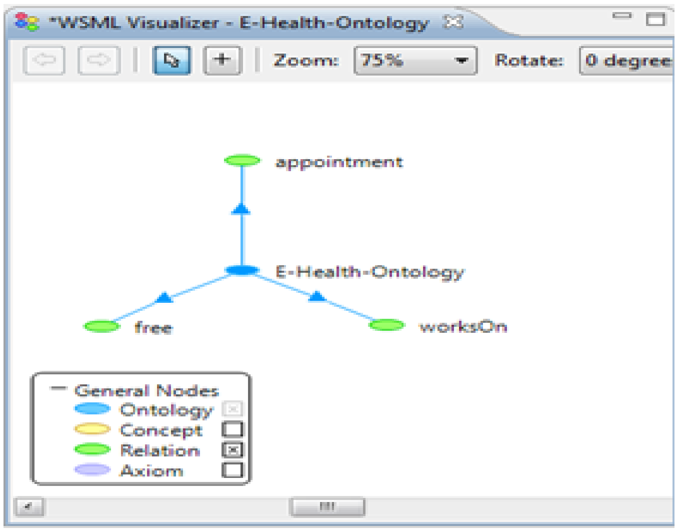


Figure 3.16: WSML visualizer showing the relations of e-health ontology

```
relation appointment(ofType DateTime,ofType Doctor,ofType Patient)

relation worksOn(ofType Doctor, ofType Day,ofType _integer)

relation free( ofType DateTime,  ofType Doctor)
```

Figure 3.17: Relations in the e-health ontology

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace { _"http://cmpe.emu.edu.tr/omid/services#",
   omid _"http://cmpe.emu.edu.tr/omid#",
   dc        _"http://purl.org/dc/elements/1.1#",
   wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
   discovery _"http://wiki.wsmx.org/index.php?
                  title=DiscoveryOntology#"  }

webService ServiceDoctorAppointmentMTM
nfp
  dc#title hasValue "making an appointment
    with a doctor at MTM(magusaTipMerkezi)"
  dc#type hasValue
    _"http://www.wsmo.org/2004/d2/#webservice"
  wsml#version hasValue "$Revision: 1.4 $"
endnfp

importsOntology
            _"http://cmpe.emu.edu.tr/omid#e-health-Ontology"
```

Figure 3.18: Prelude of the Web service for MTM

Such a specialization method, which is currently lacking in WSMO, will also help enormously in Web service discovery and execution.

#### 3.3.2.1 E-health Web Service Specification for Magusa Tip Merkezi (MTM) Hospital

Figure 3.18 depicts the first part of the WSMO specification for the appointment-making Web service of MTM. It includes the variant of WSML used (in this case WSML-Rule), namespace declarations to avoid writing fully qualified names in the specification, the name of the Web service ("ServiceDoctorAppointmentMTM"), its non-functional properties, as well as imported ontologies. Non-functional properties are used to describe information which does not have direct effect on the functionality of the component being described. Imported ontologies form the bridge between goals and Web services, so that both refer to the same "state of the world."

Figure 3.19 contains the beginning of the capability specification. Again it has non-functional properties, followed by the shared variables *?onDateTime, ?patient, ?hospital, ?doctor, and ?specialty*.

Figure 3.20 depicts the *precondition* for making an appointment. The precondition simply requires the existence of an instance of the RequestAppointment concept in the ontology. This instance contains all the necessary information for making an

```
capability DoctorAppointment2Capability
  nonFunctionalProperties
    discovery#discoveryStrategy hasValue
            discovery#HeavyweightDiscovery
    discovery#discoveryStrategy hasValue
            discovery#NoPreFilter
  endNonFunctionalProperties

 sharedVariables {?onDateTime,?patient,?doctor,?specialty}
```

Figure 3.19: Capability Specification of the MTM Web Service: non-functional properties and shared variables

```
precondition
  nonFunctionalProperties
    dc#description hasValue "the input is Specialty;
       The output is the an appointment at the earliest
       time with a doctor which has this Specialty."
    endNonFunctionalProperties
    definedBy
    ?reqApp[
            omid#specialty hasValue ?specialty,
            omid#patient hasValue ?patient,
            omid#onDateTime hasValue ?onDateTime,
            omid#hospital hasValue magusaTipMerkezi,
            omid#doctor hasValue ?doctor
           ]memberOf RequestAppointment.
```

Figure 3.20: Precondition for MTM Web service capability

appointment. Depending on the level of specificity desired by the requester, some attributes of this instance can be left empty though.

Figure 3.21 depicts the *assumption* for the appointment Web service. It states that there must be a doctor at the MTM hospital whose specialty is the same as the one in the request, and who is free at the time and date for which the appointment is requested.

Figure 3.22 depicts the *postcondition* of the appointment making Web service. It simply states the existence of an instance of the Appointment concept in the ontology after the Web service has finished its execution. This instance contains information about the patient, the doctor with whom the appointment was made, the hospital and the date/time of the appointment.

Figure 3.23 depicts the definition of the *effect* element in the Web service speci-

```
assumption
  nonFunctionalProperties
    dc#description hasValue "Make sure the doctor
       is free"
    endNonFunctionalProperties
    definedBy
      ?doctor[
              omid#hasSpecialty hasValue ?specialty ,
                omid#worksAt hasValue magusaTipMerkezi
               ]memberOf omid#Doctor and
            free(?onDateTime ,?doctor).
```

Figure 3.21: Assumption for MTM Web service capability

```
postcondition
        nonFunctionalProperties
                dc#description hasValue "make an appointment"
        endNonFunctionalProperties
    definedBy
         ?appointment[ omid#dateTime hasValue ?onDateTime ,
                       omid#doctor hasValue ?doctor ,
                       omid#patient hasValue ?patient ,
                       omid#hospital hasValue magusaTipMerkezi
                     ] memberOf omid#Appointment and
         ?doctor[  omid#hasSpecialty hasValue ?specialty ,
                   omid#worksAt hasValue magusaTipMerkezi
                  ] memberOf omid#Doctor .
```

Figure 3.22: Postcondition for MTM Web service capability

```
effect
  nonFunctionalProperties
    dc#description hasValue "make an appointment"
  endNonFunctionalProperties
  definedBy
    not (free(?onDateTime,?doctor)).
```

Figure 3.23: Effect for MTM Web service capability

fication where, as a consequence of the execution of the Web service, the date of the appointment will be considered as a busy hour for the doctor involved in the appointment. A *choreography* specification would have been needed if there was a sequence of activities that were needed between the requester and service provider. In our case, this was not the case, so we did not attempt to give a choreography specification.

We have however studied in detail the proposed choreography specification mechanism in WSMO, and realized that practically it is useless for specifying the interaction of the requester and the Web service. We have more to say on this in section 4.5.

Since our Web service made use of no other Web services to implement its functionality, there was no need to specify *orchestration* for it. But even if we wanted to do it, WSMO orchestration does not yet exist (an omission in the definition of WSMO) and we would not have the means to do it.

#### 3.3.2.2 E-health Web Service for Yasam Hastanesi(YH) Hospital

In this section we give the semantic specification of an appointment-making Web service for Yasam Hastanesi (YH) Hospital. The specification, depicted in Figure 3.24 is very similar to the one for MTM, and quite self explanatory. We wrote this specification in order to experiment with different discovery strategies. It has however demonstrated the need for a specialization mechanism for Web services, i.e. "deriving" a specific Web service specification from a more general one, much like the "generic" or "template" mechanism of programming languages. We shall talk about this more in section 4.5.

```
webService ServiceDoctorAppointmentYasamHastanesi

importsOntology
  _"http://cmpe.emu.edu.tr/omid#e-health-Ontology"

capability DoctorAppointment2Capability
nonFunctionalProperties
 discovery#discoveryStrategy hasValue discovery#HeavyweightDiscovery
 discovery#discoveryStrategy hasValue discovery#NoPreFilter
endNonFunctionalProperties

sharedVariables {?onDateTime,?patient,?doctor, ?specialty}

precondition
nonFunctionalProperties
  dc#description hasValue "Data from requester."
  endNonFunctionalProperties
  definedBy
    ?reqApp[
      omid#specialty hasValue ?specialty,
      omid#patient hasValue ?patient,
      omid#onDateTime hasValue ?onDateTime,
      omid#hospital hasValue yasamHastanesi,
      omid#doctor hasValue ?doctor
    ] memberOf RequestAppointment.

assumption
nonFunctionalProperties
  dc#description hasValue "Make sure doctor available and free"
endNonFunctionalProperties
definedBy
   ?doctor[omid#hasSpecialty hasValue ?specialty,
           omid#worksAt hasValue yasamHastanesi]
               memberOf omid#Doctor and
   free(?onDateTime,?doctor).

postcondition
nonFunctionalProperties
  dc#description hasValue "Appointment object"
endNonFunctionalProperties
definedBy
  ?appointment[ omid#dateTime hasValue ?onDateTime,
                omid#doctor hasValue ?doctor,
                omid#patient hasValue ?patient,
                omid#hospital hasValue yasamHastanesi
              ] memberOf omid#Appointment and
   ?doctor[  omid#hasSpecialty hasValue ?specialty,
                 omid#worksAt hasValue yasamHastanesi
               ] memberOf omid#Doctor.

effect
        nonFunctionalProperties
        dc#description hasValue "make an appointment"
    endNonFunctionalProperties
    definedBy
        not (free(?onDateTime,?doctor)).
```

Figure 3.24: WSMO specification of the appointment-making Web service for Yasam Hastanesi

### 3.3.3 E-health Goals: Making an Appointment

In this section we give two goals for making appointments, each for a different person. Figure 3.25 depicts the goal of making an otolaryngology appointment for "alex." In Figure 3.26 we have "sarah" requesting an appointment for gynecology at magusaTip-Merkezi hospital. Note the similarity between the two goals. Just like in the case with Web services, it is obvious that a mechanism for specifying goals at a more abstract level and specializing this abstract specification to get different concrete goals would be very desirable. More on this in section 4.5.

### 3.3.4 E-health Mediators

Given the simplicity of our application, no need has arisen for defining mediators. Although it is a big challenge to define useful mediators in real life, the possibility of being able to define them in WSMO is promising.

## 3.4 Evaluation of WSMO and WSML

In this section we discuss the strong and weak points of WSMO and WSML as discovered through our studies of their specification and the practical experience gained through the e-health use case scenario. We also suggest possible improvements wherever possible. All suggested improvements can be the basis of future development of WSMO and WSML.

### 3.4.1 General Observations

WSMO boasts a comprehensive approach that tries to leave no aspect of semantic Web services out. These include ontologies, goals, Web services and mediators. In the same spirit of thoroughness, designers of WSML have adopted the paradigm of trying to provide everything everybody could ever want and let each potential user chose the "most suitable" variant of the language for the job at hand. This approach has resulted in a complex syntax (please refer to Figure 3.27,taken from[32], for the syntax of WSML-Full logic rules), as well as a complex set of rules that differentiate one version of the language form another.

### 3.4.2 Deficiencies in Syntax

WSML-DL and WSML-Full have no explicit syntax for the description logic component [32], relying on a first-order encoding of description logic statements. Without

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace { _"http://cmpe.emu.edu.tr/omid/goals#",
           omid _"http://cmpe.emu.edu.tr/omid#",
           dc        _"http://purl.org/dc/elements/1.1#",
           wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
     discovery
       _"http://wiki.wsmx.org/index.php?title=DiscoveryOntology#"}



goal Goal_MakeAppointment2-alex
importsOntology _"http://cmpe.emu.edu.tr/omid#e-health-Ontology"


capability Goal_MakeAppointment2-alexCapability


nonFunctionalProperties
    discovery#discoveryStrategy
          hasValue discovery#HeavyweightDiscovery
    discovery#discoveryStrategy
          hasValue discovery#NoPreFilter
endNonFunctionalProperties



sharedVariables {?dateTime,?hospital,?doctor}




precondition
  definedBy
    ?reqApp[
            omid#specialty hasValue otolaryngology,
            omid#patient hasValue alex,
           ]memberOf RequestAppointment.





postcondition
  definedBy
    ?appointment[
                 omid#dateTime hasValue ?dateTime,
                 omid#doctor hasValue ?doctor,
                 omid#patient hasValue alex,
                 omid#hospital hasValue ?hospital
                ]memberOf omid#Appointment   and
         ?doctor[
                   omid#hasSpecialty hasValue
                           otolaryngology,
                   omid#worksAt hasValue ?hospital
                  ]memberOf omid#Doctor and
         ?dateTime memberOf DateTime and
         ?hospital memberOf Hospital.
```

Figure 3.25: Requesting an otolaryngology appointment for “alex”

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace { _"http://cmpe.emu.edu.tr/omid/goals#",
           omid _"http://cmpe.emu.edu.tr/omid#",
           dc        _"http://purl.org/dc/elements/1.1#",
           wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
     discovery
       _"http://wiki.wsmx.org/index.php?title=DiscoveryOntology#" }




goal Goal_MakeAppointment2-sarah
importsOntology
  _"http://cmpe.emu.edu.tr/omid#e-health-Ontology"



capability Goal_MakeAppointment2-sarahCapability
nonFunctionalProperties
discovery#discoveryStrategy hasValue discovery#HeavyweightDiscovery
discovery#discoveryStrategy hasValue discovery#NoPreFilter
endNonFunctionalProperties


sharedVariables {?dateTime,?doctor  }




precondition
  definedBy
    ?reqApp[
            omid#specialty hasValue gynecology,
            omid#patient hasValue sarah,
            omid#hospital hasValue magusaTipMerkezi
           ]memberOf RequestAppointment.





postcondition
  definedBy
    ?appointment[
                 omid#dateTime hasValue ?dateTime,
                 omid#doctor hasValue ?doctor,
                 omid#patient hasValue sarah,
                 omid#hospital hasValue magusaTipMerkezi,
                ]memberOf omid#Appointment   and
         ?doctor[
                   omid#hasSpecialty hasValue gynecology,
                   omid#worksAt hasValue magusaTipMerkezi
                 ]memberOf omid#Doctor and
         ?dateTime memberOf DateTime.
```

Figure 3.26: Requesting a gynecology appointment for “sarah”

proper syntax, it is not possible to use them in the specification of semantic Web services in a convenient way.

### 3.4.3 Logical Basis of WSMO

The ontology component of WSMO is based on F-logic, which gives this component a solid theoretical foundation. However, its precise relationship to F-logic has not been given formally, and what features of F-logic have been left out are not specified explicitly.

### 3.4.4 Lack of a Semantics Specification for Web Service Methods/Messages

In spite of all the effort at comprehensiveness, there are significant omissions in WSMO, such as specification of the semantics of actual methods (messages) that the Web service provides, which makes it impossible to *prove* that after a "match" occurs between a goal and a Web service, the postcondition of the goal will indeed be satisfied. Even worse, once matching succeeds and the Web service is called according to the specified choreography, the *actual results* of the invocation may not satisfy the postcondition of the goal! Below, we explain why.

In WSMO, matching between a goal and Web service occurs by considering the pre-post conditions of the goal and Web service, and this is fine. The problem occurs because of the *lack* of a semantic specification (for example, in the form of pre-post conditions) for Web service *methods/messages*, and how these methods are actually called through the execution of the choreography engine. Method calls are generated according to availability of "data" in the form of instances, and the mapping of instances to parameters of methods [15]. There is no consideration of logical conditions which must be true before the method is called, and no guarantee of the state of the system after the method is called, since these are not specified for the Web methods. Instances of a concept can be parameters to more than one Web method. Assuming two methods *A* and *B* have the same signature, it may be the case that an unintended method call can be made to *B*, when in fact the call should have been made to *A*, which results in wrong computation. Consequently, not only is it impossible to *prove* that after a "match" occurs between a goal and a Web service, the post-condition of the goal will be satisfied, but also once the Web service execution is initiated, the

Any atomic formula $\alpha$ which does not contain the
inequality symbol (!=) or the unification
operator (=) is in Head(V).

Let $\alpha,\beta \in$ Head(V), then $\alpha$ and $\beta$ is in Head(V).

Given two formulae $\alpha$, $\beta$ such that $\alpha$, $\beta$ do not
contain implies, impliedBy, equivalent
, the following formulae are in Head(V):

$\alpha$ implies $\beta$, if $\beta \in$ Head(V) and $\alpha \in$ Head(V) or $\alpha \in$ Body(V)
$\alpha$ impliedBy $\beta$, if $\alpha \in$ Head(V) and $\beta \in$ Head(V) or $\beta \in$ Body(V)
$\alpha$ equivalent $\beta$ if $\alpha \in$ Head(V) or $\alpha \in$ Body(V) and $\beta \in$ Head(V) or $\beta \in$ Body(V)

Any admissible head formula in Head(V) is a formula in L(V).
Any atomic formula $\alpha$ is in Body(V).
For $\alpha \in$ Body(V), naf $\alpha$ is in Body(V).
For $\alpha,\beta \in$ Body(V), $\alpha$ and $\beta$ is in Body(V).
For $\alpha,\beta \in$ Body(V), $\alpha$ or $\beta$ is in Body(V).
For $\alpha,\beta \in$ Body(V), $\alpha$ implies $\beta$ is in Body(V).
For $\alpha,\beta \in$ Body(V), $\alpha$ impliedBy $\beta$ is in Body(V).
For $\alpha,\beta \in$ Body(V), $\alpha$ equivalent $\beta$ is in Body(V).
For variables ?x1,...,?xn and $\alpha \in$ Body(V), forall ?x1,...,?xn ($\alpha$) is in Body(V).

For variables ?x1,...,?xn and $\alpha \in$ Body(V), exists ?x1,...,?xn ($\alpha$) is in Body(V).
Given a head-formula $\beta \in$ Head(V) and a body-formula $\alpha \in$ Body(V), $\beta$ :- $\alpha$ is a formula.

Figure 3.27: Syntax of rules in WSML-Full (taken from [65])

computation itself can produce wrong results, invalidating the logical specification of the Web service.

Unfortunately, the interplay between choreography, grounding and logical specification of what the Web service does (including the lack of the specification of semantics for Web service methods) has been overlooked in WSMO. All these components need development and integration in order to make them part of a *coherent* whole.

### 3.4.5 Implementation and Tool Support

Some developmental tools exist which make writing WSML specifications relatively easy (e.g. WSMO ontologies, axioms), such as “Web Services Modelling Toolkit” [30]. At the same time, implementations of WSMO and WSML rely on external reasoner support, rather than having intrinsic reasoning capabilities. As such, development and testing of semantic Web service specifications cannot be made in a reliable manner. For example, no explanations are given when discovery fails for a given goal.

### 3.4.6 Choreography in WSMO

We have already talked about how the interplay of choreography and grounding can result in incorrect execution, invalidating the logical specification of a Web service. In this section, we delve more deeply into the problems of WSMO choreography.

- WSMO choreography is purportedly based on the formalism of abstract state machines [51], but in fact it is only a crude approximation. Very significantly, evolving algebras are magically replaced with the state of the ontologies as defined by instances of relations an concepts. This transformation seems to have no logical basis, so the applicability of any theory developed for abstract state machines to WSMO choreography specifications is questionable. The choreography attempt of WSMO looks more like a forward-chained expert system shell, where the role of the "working memory" is played by the current set of instances in the ontologies. It probably would be more reasonable to consider WSMO choreography in this way, rather than being based on abstract state machines.

- Both goals and Web services have choreography specifications, but there is no notion of how the choreographies of goals and Web services are supposed to match during the discovery phase. It is also not clear how the two are supposed to interact during the execution phase. In the documentation of WSMO, only the choreography of the service is made use of.

- Choreography grounding in WSMO tries to map instances to method parameters of the Web service methods by relating concepts to the methods directly. Methods are then called when their parameters are available in the current working memory. The firing of the rules are intermixed with the invocation of methods (with appropriate lowering/lifting of parameters), and changes to working memory by actions on the right hand side are forbidden (presuming that any changes will be made by the actual method call) [15]. This is a strange state of affairs, since the client may itself need to add something to the working memory, and there is no provision for this.

- The choreography rule language allows nested rules [32]. Although this nesting permits very expressive rules to be written, using the "if", "forall" and "choose" constructs in any combination in a nested manner, the resulting rules are prohibitively complex, both to understand, and to execute.

- As mentioned before, in the grounding process, only the availability of instances that can be passed as parameters to methods, and the predetermined mapping between concepts and parameters, are considered, with no preconditions for method calls. This is a major flaw, since it may be that two methods have exactly the same parameter set, but they perform very different functions, and the wrong one gets called.

- The choreography specification is disparate from the capability specification (preconditions, postconditions), whereas they are in fact intimately related and intertwined. The actions specified in the choreography should actually take the initial state of the ontologies to their final state, through the interaction of the requester and Web service. This fact is completely overlooked in WSMO choreography.

- Choreography engine execution stops in WSMO when no more rules apply. A natural time for it to stop would be when the conditions specified in the goal are satisfied by the current state of the ontology stores. Again this is a design flaw, which is due to the fact that the intimate relationship between the capability specification and choreography has been overlooked.

### 3.4.7 Orchestration in WSMO

The orchestration component of WSMO is yet to be defined. The creators of WSMO say it will be similar to choreography, and be part of the interface specification of a Web service. At a conceptual level, however, we find the specification of orchestration for a Web service somewhat unnecessary. Why would a requester care about *how* a service provider provides its service? Composition of Web services to achieve a goal *would* be much more meaningful, however. So the idea of placing orchestration within a Web service specification seems misguided. Its proper place would be inside the specification of a *complex* goal, which would help and guide the service discovery component to not only find a service that meets the requirements of the goal, but also mix-and-match and compose different Web services to achieve the requirements of the goal.

### 3.4.8 Goal Specification

The goal specification includes the components "preconditions," "assumptions," "postconditions" and "effects," just like the Web service specification. The logical correspondence between the "preconditions," "assumptions," "postconditions" and "effects," of goals and Web services is not specified at all. The usage of the same terminology for both goals and Web services is also misleading. In reality, the Web service *requires* that its preconditions and assumptions hold before it can be called, and *guarantees* that if it is called, the postconditions and effects will be true. On the other hand, the goal declares that it *guarantees* a certain state, perhaps by adding instances to the instance store, of the world before it makes a request to a Web service, and *requires* certain conditions to be true as a result of the execution of the Web service. The syntax of the goals should be consistent with this state of affairs.

### 3.4.9 Reusing Goals through Specialization

As we saw in our appointment-making semantic Web service, being able to reuse an existing goal after specializing it in some way would be very beneficial. The template mechanism of programming languages, or "prepared queries with parameters" in the world of databases are concepts which can be adapted to goals in WSMO to achieve the required specialization. Such a functionality is currently missing.

### 3.4.10 Specialization Mechanism for Web Service Specifications

Developing a Web service specification from scratch is a very formidable task. Just like in the case of specializing goals, a mechanism for taking a "generic" Web service specification in a domain, and specializing it to describe a specific Web service functionality would be a very useful proposition. To take this idea even further, a hierarchy of Web service specifications can be published in a central repository, and actual Web services can just declare that they implement a pre-published specification in the hierarchy or, they can grow the hierarchy by specializing an existing specification, and "plugging" their specification into the existing hierarchy. Such an approach will help in service discovery as well. A specialization mechanism for Web services does not exist in WSMO, and would be a welcome addition to it.

### 3.4.11 Missing Aggregate Function Capability

The logic used in WSML (even in WSML full) does not permit aggregate functions in the sense of database query languages (sum, average etc.). For example, it was not possible to impose the restriction (in a convenient way) that a doctor should see at least 15 patients per day, or a specific doctor A should work less hours per week than another doctor B (for some reason) etc. Such an addition however would require moving away from first order logic into higher order logic, with corresponding loss of computational tractability. Still, it may be worthwhile to investigate restricted classes of aggregate functionality which lend themselves to practical implementation. For example, a built-in *set of* predicate could be used to implement aggregate functions.

### 3.4.12 Extra-logical Predicates

The ability to check whether a logic variable is bound to an object, or whether it is in an unbound state (the *var* predicate of Prolog [98])is missing. At the same time, a "molecule" such as `?x[att1 hasValue ?y, att2 hasValue ?z] memberOf SomeConcept` unifies with an instance of `SomeConcept` only if it has values for attributes `att1` and `att2`. Although this is sound from a logic point of view, it is very restrictive. We may want to have it also unify with objects which, for example, have no value for the `att1` attribute. In the WSMO specification for the appointment application, such a need arises, since the appointment may be requested with differing level of detail specified in the goals. The ability to specify "optional variables" (i.e. variables that are allowed to *not* be bound to some object during unification) and then checking whether a variable is bound or not is an important feature, currently lacking in WSML.

An alternative to being able to check the "bounded" property of a variable would be to provide default “null” values in attributes for which no value has been specified, and then checking whether a variable has “null” in it or not.

### 3.4.13 Automatic Mapping between Attributes and Relations

Although we can define an axiom for mapping each attribute to a binary relation, as we have done in our e-health SWS specification, this is cumbersome when done manually. Having it done automatically would be nice, a feature currently not available in WSML.

### 3.4.14 Error Processing

There is currently no mechanism specifying how to handle errors when they arise. For example, what should be done when a constraint is violated in some ontology? There should be a way of communicating error conditions to the requester when they arise. This could be the counterpart of the exception mechanism in programming languages.

### 3.4.15 No Agreed-upon Semantics for WSML-Full

WSML-Full, which is a combination of WSML-DL and WSML-rule, has no agreed-upon semantics yet [47] yet. With no semantics available, it is hard to imagine how WSML-Full specifications could be processed at all.

### 3.4.16 Discussion

Our investigation has revealed several deficiencies and flaws with WSMO and WSML. Most notably, the choreography component needs to be re-thought over and made to fit better into the remaining framework. Other important weaknesses include the overall complexity and size of WSML (in all its variants), the lack of a specialization mechanism for goals and Web services, semantics for Web service methods, aggregate function capability, extra-logical predicates, mapping between attributes and relations, error processing, and an agreed-upon semantics for WSML-Full.

# Chapter 4

# LOGICAL INFERENCE BASED DISCOVERY AGENT

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.

## 4.1 The Intelligent Semantic Web Service Matchmaker Agent

### 4.1.1 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.

### 4.1.2 FLOG4SWS: A Sub-language of F-logic for Semantic Web Services Specification

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 [57] or some of its derivatives, is not an easy proposition. What is needed is a compromise: a "clean" and concise subset of first order formulas that are easily understood, and can efficiently be used in determining the validity of certain implications that will guarantee a satisfactory match, but is at the same time expressive enough in the context of Web service discovery.

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

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

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

### 4.1.3 Syntax of the Sub-language of F-logic used for Specification of Goals, Web Services and Ontologies

In Figure 4.1 we have the Extended Backus Normal Form (EBNF) grammar for the sublanguage of F-logic we used for specifying goals, Web services and ontologies. This grammar should be considered in conjunction with the full grammar of Flora-2 given in [83]. 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 the constraint as its parameter. Constraints are verified at various stages in the matchmaking process. A “Fact” is either a predicate or an object with an id that is an anonymous variable. The nonterminal “Term” is used to define both predicates and terms. Finally, an “atom” is any symbol without any internal structure.

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


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


Ontology := (Rule | ConstraintRule)*


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


FactList := Fact (',' Fact)*


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


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


Query := Fact | QueryNegative | QueryConjunct | QueryDisjunct


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


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


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


ConstraintID := atom


GoalID := atom


WebserviceID := atom
```

Figure 4.1: EBNF grammar for FLOG4SWS

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

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

1. $co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre \models ws.pre$: The precondition of the Web service should be logically entailed by the common ontology, local ontologies of the goal and Web service, and what is provided/guaranteed by the goal (i.e. $goal.pre$).

2. $co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre \bigwedge (ws.pre \Rightarrow ws.post) \models goal.post$: If the conditions for the Web service call are satisfied, then the requirements of the goal should be satisfied. Note how we assume that the execution of the Web service guarantees the validity of the implication in $ws.pre \Rightarrow ws.post$.

3. $worldBefore \models ws.assumption$: The assumptions the Web service makes about the world must be true before it can be called. These assumptions are independent of what the client supplies to the Web service (e.g. flights canceled due to weather conditions etc.)

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

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

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

Of special interest is the way the computation of a Web service is treated above: as an implication from its precondition to its postcondition ($ws.pre \Rightarrow ws.post$), and an implication from its assumption to its effect ($ws.assumption \Rightarrow ws.effect$). In the presence of state change caused by the execution of the Web service, however, this can be problematic. What was true in the precondition of the Web service may suddenly become false after the 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 [73]) *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 [42] or temporal logic [58], are required to reason with non-monotonic changes caused by the execution of the Web service, we can get around this problem by writing our specifications that use only monotonic changes to the state, but *simulate* non-monotonicity. For state changes inside an object, we can return information about *scheduled* activity regarding changes to the object. For example, the postcondition of the Web service can contain a predicate $scheduledUpdate(?x,4)$ instead of $?x = 4$ for some variable that was supposed to have some other value in the precondition. The goal postcondition then can query the $scheduledUpdate(?x,4)$ predicate. Similarly, an object can be marked for deletion, e.g. $scheduledDelete(someObject)$ may be part of the post condition instead of the object actually being deleted. After all, in the Web service discovery phase, we are not concerned about real changes actually happening, but rather the knowledge that such changes will take place if the Web service is called.

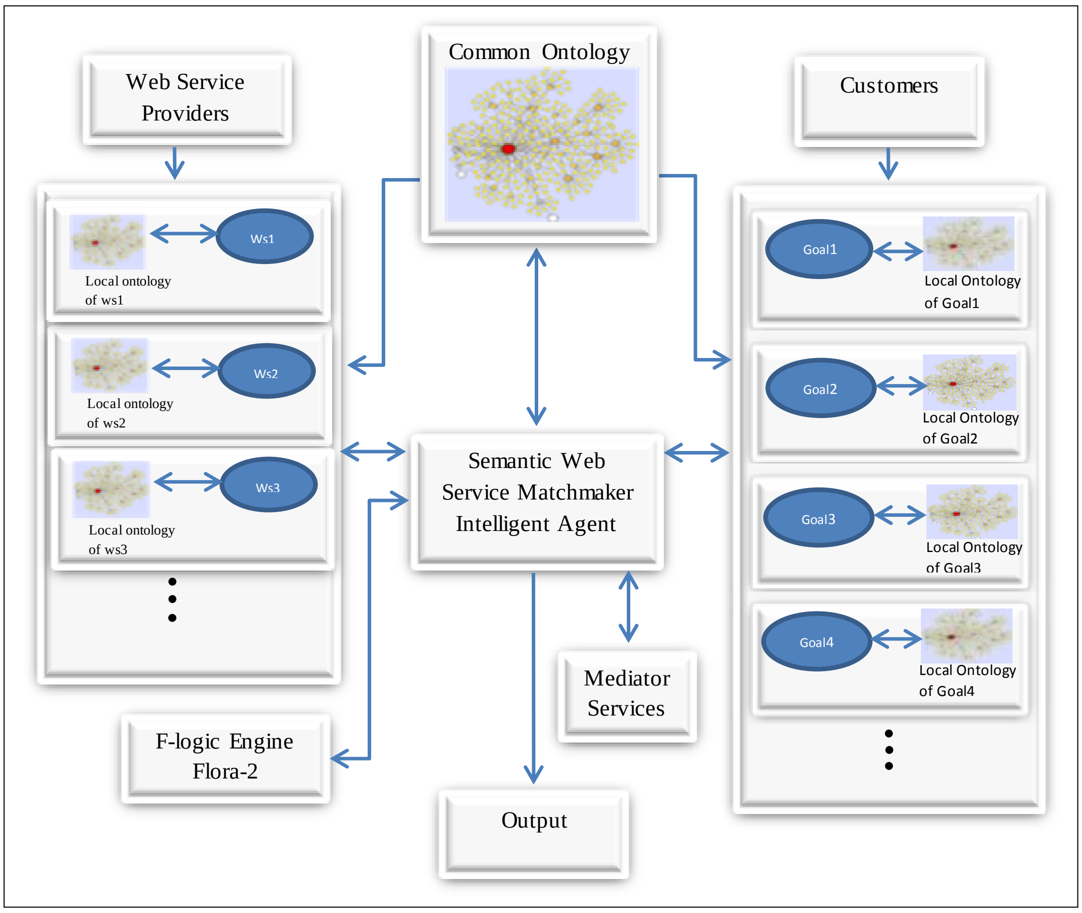


Figure 4.2: The proposed Semantic Web service matchmaker intelligent agent architecture

### 4.1.6 The Intelligent Matchmaker Agent Architecture

Figure 4.2 depicts the architecture of our intelligent matchmaker agent. At the center of the figure sits the agent itself. It has access to Web service specifications, goal specifications, the common ontology, and mediation services. Since it is implemented in FLORA-2, it has access to all the underlying functionality of the FLORA-2 engine as well. The agent verifies proof commitments one by one for each goal-Web service pair, keeping track of both the successful matches and the unsuccessful ones, and reports the result at the end. There is no "degree of match" computed, since we are interested only in Web services that satisfy the requirements of a goal completely.

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

## 4.2 Implementation of the Matchmaker Agent in FLORA-2

### 4.2.1 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 [43, 42] into tabled Prolog code [83]. Basically, FLORA-2 supports a programing language that is a dialect of F-logic including numerous extensions, that involves a natural way to do meta-programming in the style of HiLog, logical updates in the style of Transaction Logic, and a form of defeasible reasoning described in [96]. 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 [81].

### 4.2.2 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 this chapter. 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

```

WebService(ws1, 'C:\Users/OMID/Desktop/test/ws1', W1):- true ,!.
WebService(ws2, 'C:\Users/OMID/Desktop/test/ws2', W2):- true ,!.
WebService(ws3, 'C:\Users/OMID/Desktop/test/ws3', W3):- true ,!.

Goal(goal1, 'C:\Users/OMID/Desktop/test/goal1'):- true ,!.
Goal(goal2, 'C:\Users/OMID/Desktop/test/goal2'):- true ,!.
Goal(goal3, 'C:\Users/OMID/Desktop/test/goal3'):- true ,!.
Goal(goal4, 'C:\Users/OMID/Desktop/test/goal4'):- true ,!.

CommonOntology('C:\Users/OMID/Desktop/test/CommonOntology'):- true ,!.

WebServices([ws1, ws2, ws3]).
Goals([goal1, goal2, goal3, goal4]).
```

Figure 4.3: Facts about goals, Web services and common ontology

postconditions, and extending our program to deal with them is straightforward.

### 4.2.3 The Top-level Matcher Loop

Figure 4.3 has predicates that contain information about the files containing goals, Web services and the common ontology. Due to a syntactic glitch in the implemetnation of FLORA-2, the *cut (!)* operator is not allowed directly after the implication (: −) operator, so we had to use the *true* predicate before the *cut* operator. Each *Webservice* predicate contains information about a specific Web service, a *Goal* predicate contains information about a specific goal, and *CommonOntolgy* contains information about the common ontology. The predicates *WebServices* and *Goals* describe which goals and Web services should be included in the match operation.

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

Figure 4.4 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. *run*2 pairs goals and Web services, and checks by calling *main* whether a goal matches a Web service. *main* takes a goal name and a Web service name, loads the goal and Web service data, local ontologies of the Web service and goal (they are in the same file as the Web service or goal respectively), as well as the common ontology data into a module that is unique to the current Web service, and calls *matchh* which does the proof commitment verification for the goal and Web service.

In lines 24 and 37 of Figure 4.4, the current module is forced to empty its contents by loading a file called “empty” (which naturally contains nothing inside) into it, so that the same module can be used to test whether some other goal matches the Web service that “owns” the module.

### 4.2.4 Proving the Commitments for a Successful Match

In Figure 4.5 we have the code that tries to prove the commitments specified in section 4.1. The predicate *matchh* retrieves the precondition of goal and inserts it temporarily into the module that was given to it. Then *prove* is called, which verifies the proof commitments. Specifically, in line 6, the proof commitment $[co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre \models ws.pre]$ is verified, and in line 8, the proof commitment $[co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre \bigwedge (ws.pre \Rightarrow ws.post) \models goal.post]$ is verified.

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

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

We could not use the exact same approach for the second proof commitment $[co \bigwedge goal.ont \bigwedge ws.ont \bigwedge goal.pre \bigwedge (ws.pre \Rightarrow ws.post) \models goal.post]$, however, since FLORA-2 does not allow temporary insertion of rules (of the form $lhs : -rhs$) in a module. What we needed was the ability to insert into the module $ws.post : -ws.pre$ ($ws.post$ is a set-valued attribute, and logical variables get instantiated to each member of the set one at a time according to the semantics of FLORA-2). To get around this restriction, we devised the $prove/3$ predicate which takes three parameters (postcondition of a goal, postcondition of a Web service, and precondition of the same Web service). In lines 16-17 of Figure 4.5, the objective of using the implication $(ws.pre \Rightarrow ws.post)$ is achieved operationally by unifying the postcondition of the

```

run(?Result):-
    WebServices(?WSS),
    Goals(?GS),
    run2(?GS, ?WSS, ?Result).

run2([],?_,[]):-
    true,!. // goals finished

run2([?Goal|?RestGoals],[], ?Result):-
     true,!,
     WebServices(?WSS),
     run2(?RestGoals,?WSS,?Result).

run2([?Goal|?RestGoals],[?WS|?RestWebServices],
       [[?Goal,‘ MATCHES ’, ?WS]|?RestResult]):-
           main(?Goal,?WS), !,
           run2([?Goal|?RestGoals],?RestWebServices, ?RestResult).

run2([?Goal|?RestGoals],[?WS|?RestWebServices],
       [[?Goal,‘ DOES NOT MATCH ’, ?WS]|?RestResult]):-
           true,!,
           WebService(?WS, ?, ?WsModule),
           [‘C:\Users/OMID/Desktop/test/empty’>>?WsModule],
           run2([?Goal|?RestGoals],?RestWebServices, ?RestResult).

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,
   [‘C:\Users/OMID/Desktop/test/empty’>>?WsModule],!.
```

Figure 4.4: Matching agent top-level predicates

```
matchh(?Goal, ?WebService, ?WsModule):-
    check_constraints,
    ?WebService[inputs->${startW[pre->?WsPre, post->?WsPost]}],
    ?Goal[inputs->${startG[pre->?GoalPre, post->?GoalPost]}],
    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( ?X):- ?X@?WsModule.

prove(?GoalPost, ?WsPost, ?WsPre):-
    ?GoalPost = ?WsPost,!, prove(?WsPre).

prove(and(?X,?Y), ?WsPost, ?WsPre):- !,
   prove(?X, ?WsPost, ?WsPre), prove(?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)).

prove( ?X, ?WsPost, ?WsPre):- ?X@?WsModule.
```

Figure 4.5: Proving the commitments for a match

```
1   check_constraints:-
2       constraints(?C),
3       check_constraints(?C,?R),
4       verify_results(?R).
5
6   check_constraints([],[]).
7
8   check_constraints([?H|?T],[?F|?R]):-
9       check_constraint(?H,?F),
10      check_constraints(?T,?R).
11
12  check_constraint(?Cons, successful(?Cons)):-  constraint(?Cons).
13
14  check_constraint(?Cons, failure(?Cons)):-
15      naf  constraint(?Cons).
16
17  verify_results([]).
18  verify_results([successful(?C)|?T]):-
19      writeln(successful(?C))@_plg,
20      verify_results(?T).
21
22  verify_results([failure(?C)|?T]):-
23        writeln(failure(?C))@_plg,
24        verify_results(?T).
```

Figure 4.6: Checking constraints for violations

goal with the postcondition of the Web service, and then proving the precondition of the Web service, thereby implementing backward reasoning manually, and achieving the same result as if $ws.post : -ws.pre$ was inserted into the module.

### 4.2.5 Checking Constraints

Figure 4.6 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.

## 4.3 Specifying Web Services, Goals and Ontologies in FLORA-2

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

### 4.3.1 Sample Web Service Specification

Figure 4.7 depicts the specification of a Web service for making doctor appointments. The precondition requires an object of concept *RequestAppointment* to be provided by the goal, containing the request information. *?DS,?PN,?Date,?HN* and *?X* are logic variables that will be bound to the corresponding values that should be provided by the requester. Before the match is successful, it must be verified that the age of the patient is more than 18 (for some reason!), there is a hospital in the same country that the patient lives in, and there is a doctor with the correct specialization area in that hospital.

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

The local ontology used by the Web service specification, also given in Figure 4.7, represents an abstraction of an actual local database that might be used by the Web service that is being semantically described by the specification.

### 4.3.2 Sample Goal for Consuming an Appointment Service

We have in Figure 4.8 a goal for consuming an appointment making service. The goal gives specific information about the appointment, such as the required specialty, hospital name, and age of the patient. The appointment date field is left empty, which might mean that any date is fine. However, in the postcondition of the goal, the actual date returned by the Web service is checked to be in the range 20 to 22, so any other date will cause a mismatch.

As part of the precondition, the *fact* that the patient (philip) lives in Paris is given. This fact will be used to deduce that philip lives in France and can only make an appointment at a hospital in France through using the Web service whose specification is given in Figure 4.7.

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.

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

            post->${
                   ?_[
                      appointmentDate->?Date,
                      doctorName->?DN,
                      patientName->?PN,
                      hospitalName->?HN
                     ]:Appointment
                   }
           ]
     }
    ].
doctor1[doctorName->robert,
        specialty->opthalmology,
        hospitalName->MontpellierHospital,
        availableDate->{20, 21, 22, 23, 24, 25}
       ]:Doctor.
doctor2[doctorName->omid,
        specialty->otolaryngology ,
        hospitalName->MontpellierHospital,
        availableDate->{13, 14, 15, 16}
       ]:Doctor.
doctor3[doctorName->martin,
        specialty->gynecology ,
        hospitalName->MontpellierHospital,
        availableDate->{3, 4, 5, 6}
       ]:Doctor.
hospital(MontpellierHospital, France).
```

Figure 4.7: Web service specification for making appointments

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

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

Figure 4.8: Goal for consuming the appointment making Web service

### 4.3.3 Common Ontology

The common ontology, given in Figure 4.9, 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.

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.

```
                                //CommonOntology.flr

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

constraints([noPatientClash, noDoctorClash]).

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

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

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

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

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

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

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

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

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

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

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

Figure 4.9: Common ontology used by goals and Web services

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

Figure 4.10: Result returned by the matchmaker on some goals and Web services

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

Figure 4.10 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 this part, but are available for downloading at [89].

### 4.3.5 Performance Statistics for FLOG4SWS

Table 4.1 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 4.11 and 4.12 depict the same information graphically. As can be seen, there is a linear relationship between the number of Web 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.

Table 4.1: Matching time of one goal with different number of Web services in FLOG4SWS

| Number of Web services | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Goal (matched) | 0.407 | 0.860 | 1.437 | 1.828 | 2.203 | 2.734 | 3.265 | 3.703 | 4.891 |
| Goal (not matched) | 0.312 | 0.500 | 0.766 | 0.890 | 1.312 | 1.453 | 1.813 | 2.000 | 2.750 |

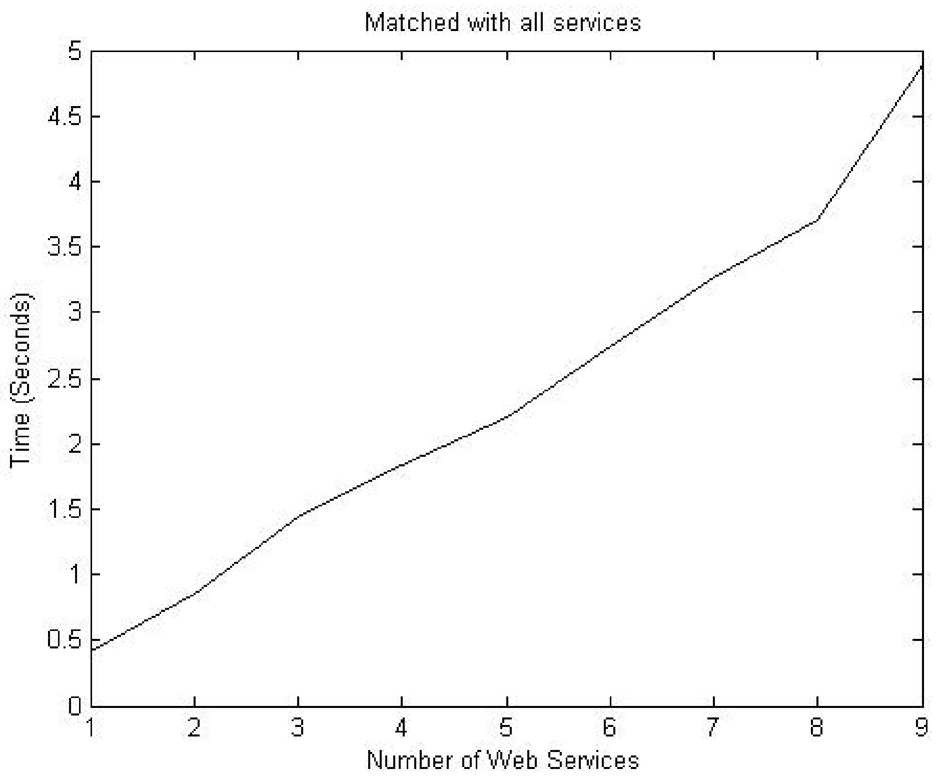


Figure 4.11: Timing when goal matches all Web services

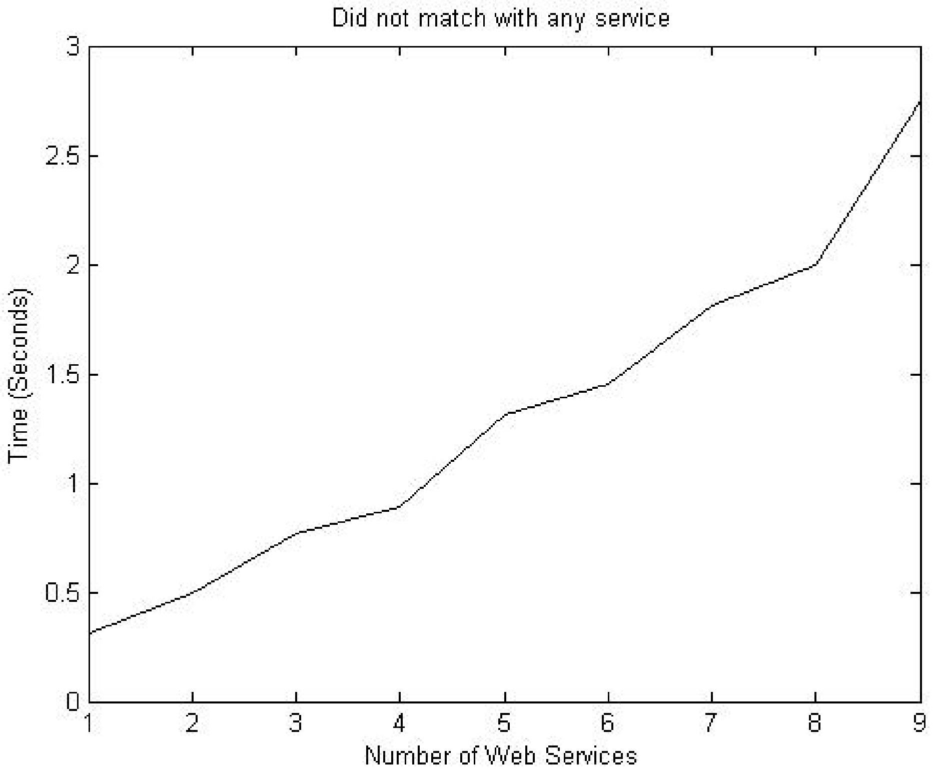


Figure 4.12: Timing when goal does not match any Web service

## 4.4 Comparison with 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 [71] 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 [71]. Incidentally, we do use the transactional logic capabilities of FLORA-2 in our implementation, but only as a tool.

In [85, 94, 90, 91, 69], 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 [84]. 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 [39] 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 [95] 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.

[86] 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 [91] the authors propose OWL-S/UDDI matchmaker. They embed OWL-S profile information inside UDDI advertisements before performing a match. Their algorithm matches the outputs/inputs of the request against the inputs/outputs of the published advertisements. The match between the inputs or outputs depends on the relationship between the OWL concepts to which the objects belong. So the proposal is a set-based approach using OWL-S. Any reasoning done is for determining subsumption realtionships.

A matchmaking algorithm based on bipartite graphs for Semantic Web services specifies in OWL-S is presented in [67]. 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 [50], 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. Similary, in [63] 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 [49] 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 [75], uses different matching filters to retrieve Semantic Web services written in a dialect of WSML-Rule. WSMO-MX recursively computes "logic-based and syntactic similarity-based matching degrees and returns a ranked set of services that are semantically relevant to a given query. The matching filters perform ontology-based type matching, logical constraint matching, and syntactic matching." The proposed system is "aproximative" and does not guarantee with 100% certainty the suitability of the discovered services to satisfy the needs of the requester. This is in line with the authors' belief that Semantic Web research has started to shift towards "more scalable and approximative rather than computationally expensive logic-based reasoning with impractical assumptions." Our work disproves this belief: it *is* possible to have logic-based reasoning that is not prohibitively expensive, provided that an appropriate subset of logic which is expressive enough to practically specify goal requirements and Web service capabilities is used.

In other related work, [44] 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 [52]. In [76], a graded relevance scale for Semantic Web services matchmaking is proposed as measurements to evaluate SWS matchmakers based on such graded relevance scales.

Table 4.2: FLOG4SWS versus other SWS languages

| | OWL-S | WSML (All variants) | FLOG4SWS |
|---|---|---|---|
| Dedicated matching engine | - | - | + |
| Matching based on logical implication | - | - | + |
| Logic language included | - | + | + |
| Efficient goal-directed reasoning for matching | - | - | + |

Table 4.2 depicts a high-level comparison of FLOG4SWS with two of the most well-known semantic Web service frameworks, namely OWL-S and WSMO (as implemented through WSML).

## 4.5 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 Figure 4.13 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).

We should also note that the matching agent matches a goal to each Web service specification sequentially, so that issues of atomicity of access to the ontologies do not come up in the implementation.

```
pred1( f[b->?x,c->?x] ).
// query:  pred1(f[b->1,c->?y]).
// Ans:     ?y unbound

a[b->${c[d->?x]},  k->${e[f->?x]}].
// query: a[b->${c[d->1]},k->${e[f->?x]}].
// Ans:    ?x unbound.

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

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

project2(${a2[b2->?z]},?z).
// query: project2(?x,?y).
// Ans: ?x = ${a2[b2 -> ?_h2209]}
//      ?y = ?_h2209
// query: project2(a2[b2->1],?z)@ml.
// Ans: No
// query: project2(${a2[b2->1]},?z).
// Ans: ?z = 1
```

Figure 4.13: Behavior of FLORA-2 with respect to unification, reification and logic variables

# Chapter 5

# UNIFYING F-LOGIC MOLECULES

In this chapter we present a correct unification algorithm for unifying F-Logic molecules. The original unification algorithm for F-Logic molecules presented by Kifer et al. contains a mistake. We identify the mistake and demonstrate it through two examples, suggest a solution to the problem in the original algorithm and prove that the resulting algorithm correctly unifies F-Logic molecules. The results presented here have been accepted for publication by Oxford journals [38].

## 5.1 Motivation and Overview

F-Logic [74] attempts to combine first order logic with object-oriented concepts. It has been used as the formal basis of deductive object oriented databases [79] and also forms the underlying logic of WSML [45], a language for semantically describing Web services, as well as the foundation of the Rule Interchange Format (RIF) [9] from W3C. F-Logic introduces the concept of an *id-term*, which syntactically is the same as a normal first order term, and is used as an abstract object identifier. Two different id-terms can denote the same object.

In F-Logic, there are classes, class hierarchies, objects, as well as scalar and multivalued methods. Objects can play the role of classes, so the distinction between classes and instances of classes is blurred.

The proof theory of F-Logic makes use of a unifier for F-Logic molecules, which is presented in [74] (Appendix C, Fig. C1, p. 837). This "unifier" is incorrect. It is meant to return the "complete set of unifiers from a molecule $m_1$ *into* some other molecule $m_2$," however the so-called "complete set of unifiers" may not be unifiers at all. In cases of non-unifiability of the two molecules, the presented algorithm can give a non-empty set as well, implying that the molecules are unifiable, when they are not. Our literature search has failed to discover any correction to the proposed "unification" algorithm in [74], and since unification is such an important aspect of automated deduction, we decided to rectify the algorithm and present a correct version of this algorithm.

We show the incorrect working of the proposed unification algorithm in [74] by giving two simple unification problems and tracing the given algorithm on them. We then suggest a modification to the proposed algorithm and prove that the modified version indeed is correct.

Below, we give some definitions (taken directly from [74]) that are made use of in the algorithm, either directly or indirectly.

The *constituent atoms* of a molecule $a[b_1@Q_1,\ldots,Q_{k1} \rightarrow c_1;\ b_2@Q_1,\ldots,Q_{k2} \rightarrow c_2;\ \ldots\ ;\ b_n@Q_1,\ldots,Q_{kn} \rightarrow c_n]$ are $a[b_1@Q_1,\ldots,Q_{k1} \rightarrow c_1]$, $a[b_2@Q_1,\ldots,Q_{k2} \rightarrow c_2]$, $\ldots$, $a[b_n@Q_1,\ldots,Q_{kn} \rightarrow c_n]$. We can replace $\rightarrow$ with any other arrow type allowed in F-Logic (namely $\bullet\!\!\rightarrow$, $\twoheadrightarrow$, $\Rightarrow$, $\Rightarrow\!\!\!\!>$ or $\bullet\!\!\twoheadrightarrow$), and the definition still is valid. The specific type and meaning of the arrows in a molecule has no bearing on the correctness or functioning of the unification algorithm, however, and a discussion on arrows is left out of the presentation.

In F-Logic, the notion of unifying two terms is replaced by "asymmetric unification of object molecules." The idea is that an instance of one object may be contained in (an instance of) another object, and the two objects should be considered unifiable. Formally, let $L_1 = S[...]$ and $L_2 = S[...]$ be a pair of object molecules with the same object id, $S$. $L_1$ is said to be a *sub-molecule* of $L_2$, denoted $L_1 \sqsubseteq L_2$, if and only if every constituent atom of $L_1$ is also a constituent atom of $L_2$. A substitution $\sigma$ is a *unifier* of $L_1$ *into* $L_2$ if and only if $\sigma(L_1) \sqsubseteq \sigma(L_2)$.

Let $\varphi$ be an atom of either of the following forms:

1. $P[Method@Q_1,\ldots,Q_k \rightsquigarrow R]$, where $\rightsquigarrow$ denotes any one of the six types of arrows allowed in method expressions (i.e. $\rightarrow$, $\bullet\!\!\rightarrow$, $\twoheadrightarrow$, $\Rightarrow$, $\Rightarrow\!\!\!\!>$, $\bullet\!\!\twoheadrightarrow$);

2. $P[Method@Q_1,\ldots,Q_k \rightsquigarrow \{\}]$, where $\rightsquigarrow$ denotes $\twoheadrightarrow$ or $\bullet\!\!\twoheadrightarrow$;

3. $P[Method@Q_1,\ldots,Q_k \rightsquigarrow ()]$, where $\rightsquigarrow$ denotes $\Rightarrow$ or $\Rightarrow\!\!\!\!>$.

Then,

$$\begin{array}{lcl}
id(\varphi) & = & P \\
method(\varphi) & = & Method \\
arg_i(\varphi) & = & Q_i, for\ i = 1,\dots,k \\
val(\varphi) & = & \begin{cases} R & if\ \varphi\ is\ of\ the\ form\ (1)\ above \\ \emptyset & if\varphi\ has\ the\ form\ (2)\ or\ (3)\ above \end{cases}
\end{array}$$

If T is a molecule, then $atoms(T)$ denotes the constituent atoms of T. If $T_1$ and $T_2$ are molecules, then $Map(T_1,T_2)$ denotes the collection of mappings $\{\lambda : atoms(T_1) \longrightarrow atoms(T_2)\}$ that preserve method arities and the type of the method expression (i.e. the type of the arrow used in those expressions).

## 5.2 Faulty "Unification" Algorithm

The "unification" algorithm of [74] is replicated in ALGORITHM 1. The function "UNIFY" is assumed to be any standard unification algorithm for terms, which also works on tuples of terms, and returns the most general unifier of its two parameters. Here, it is used to unify term-ids or tuples of term-ids.

### 5.2.1 Tracing the "Unification" Algorithm on Two Unifiable Molecules

We now trace the execution of the faulty ALGORITHM 1 on a simple case of unification. We use it to unify $a[b \rightarrow c, d \rightarrow e]$ *into* $a[b \rightarrow X, d \rightarrow Y]$. Obviously, the only unifier is $\{X\backslash c, Y\backslash e\}$, and the unification algorithm is supposed to return $\{\{X\backslash c, Y\backslash e\}\}$. Due to the fact that atoms of a molecule can be selected in any order, the given algorithm can return more than one answer. One possible answer is $\{\{X\backslash c\}, \{Y\backslash e\}, \{X\backslash c, Y\backslash e\}\}$. $\{X\backslash c\}$ and $\{Y\backslash e\}$ obviously are not unifiers, but they are returned in the result as well.

In Figure 5.1 we have a (partial) trace of the algorithm. We use indices to distinguish the values loop variables take in each iteration. For example, $\alpha[i]$ is supposed to mean the value of $\alpha$ in iteration $i$. The "line numbers" column refers to the line numbers of the algorithm.

**Algorithm 1:** Faulty unification algorithm for F-Molecules

```
   Input: A Pair of molecules, $T_1$ and $T_2$
   Output: $\Omega$- a complete set of mgu's of $T_1$ and $T_2$
1  if $id(T_1)$ and $id(T_2)$ are unifiable then
2      $\Theta := UNIFY(< id(T_1) >, < id(T_2) >)$
3  else
4      Stop: $T_1$ and $T_2$ are not unifiable
5  end
6  if $T_1$ is of the form $S[]$ (a degenerated molecule) then
7      Stop: $\Theta$ is the only mgu
8  end
9  $\Omega := \{\}$.
10 foreach mapping $\lambda \in Map(T_1, T_2)$ do
11     $\sigma_\lambda := \Theta$.
12     foreach atom $\varphi$ in $atoms(T_1)$ do
13         Let $\psi$ be $\lambda(\varphi)$.
14         Unify tuples $\sigma_\lambda(\overrightarrow{S_1})$ and $\sigma_\lambda(\overrightarrow{S_2})$, where
15                 $\overrightarrow{S_1} =< method(\varphi), arg_1(\varphi), \ldots, arg_n(\varphi), val(\varphi) >$ and
16                 $\overrightarrow{S_2} =< method(\psi), arg_1(\psi), \ldots, arg_n(\psi), val(\psi) >$.
           /* if $val(\varphi)$ or $val(\psi)$ is $\emptyset$, they are treated as
              constant $\emptyset$ for the unification purposes     */
17         if $\sigma_\lambda(\overrightarrow{S_1})$ and $\sigma_\lambda(\overrightarrow{S_2})$ unify then
18             $\sigma_\lambda := UNIFY(\sigma_\lambda(\overrightarrow{S_1}), \sigma_\lambda(\overrightarrow{S_2})) \circ \sigma_\lambda$
19         else
20             Discard this $\sigma_\lambda$ and jump out of the inner foreach to select another
               $\lambda$.
21         end
22         $\Omega := \Omega \cup \{\sigma_\lambda\}$.
23     end
24 end
25 return $\Omega$, a complete set of mgu's of $T_1$ into $T_2$
```

It is obvious that $\Omega$ that is supposed to be the complete set of *unifiers* of $T_1$ into $T_2$ possibly contains the substitutions $\{X \backslash c\}$ and $\{Y \backslash e\}$, which are not unifiers at all. Hence the presented algorithm is incorrect.

### 5.2.2 Tracing the "Unification" Algorithm on Molecules that Have No Unifier

In Figure 5.2, although $a[b \rightarrow c, d \rightarrow e]$ and $a[b \rightarrow X, d \rightarrow f]$ are obviously not unifiable, the algorithm returns $\{\{X \backslash c\}\}$ as the complete set of unifiers. Again, this is one of the possibilities, since the order in which atoms of a molecule are selected in line 12 is not specified. With a different selection order, it is possible that $\emptyset$ is returned as the result as well.

| Line no | variables | explanation |
|---|---|---|
| None | $T_1 = a[b \rightarrow c, d \rightarrow e]$ | Input molecule 1 |
| None | $T_2 = a[b \rightarrow X, d \rightarrow Y]$ | Input molecule 2 |
| 2 | $\Theta = \emptyset$ | Heads unifiable |
| 9 | $\Omega = \{\}$ | Initialize $\Omega$ |
| 10 | $\lambda[1](a[b \rightarrow c]) = a[b \rightarrow X]$, $\lambda[1](a[d \rightarrow e]) = a[d \rightarrow Y]$ | 4 possible mappings: $\lambda[1], \lambda[2], \lambda[3], \lambda[4]$. |
| 11 | $\sigma_{\lambda[1]} = \emptyset$ | Current solution |
| 12 | $\varphi[1] = a[b \rightarrow c]$ | Get atom |
| 13 | $\psi = a[b \rightarrow X]$ | Get corresponding atom |
| 15 | $\overrightarrow{S_1} =< b, c >$ | |
| 16 | $\overrightarrow{S_2} =< b, X >$ | |
| 18 | $\sigma_{\lambda[1]} = \{X \backslash c\}$ | Two atoms unify |
| 22 | $\Omega = \{\{X \backslash c\}\}$ | Problem! Partial result saved. |
| 12 | $\varphi[2] = a[d \rightarrow e]$ | Get atom |
| 13 | $\psi = a[d \rightarrow Y]$ | Get corresponding atom |
| 15 | $\overrightarrow{S_1} =< d, e >$ | |
| 16 | $\overrightarrow{S_2} =< d, Y >$ | |
| 18 | $\sigma_{\lambda[1]} = \{X \backslash c, Y \backslash e\}$ | Two atoms unify |
| 22 | $\Omega = \{\{X \backslash c\}, \{X \backslash c, Y \backslash e\}\}$ | Contains non-unifier |
| 10 | $\lambda[2](a[b \rightarrow c]) = a[b \rightarrow X]$, $\lambda[2](a[d \rightarrow e]) = a[b \rightarrow X]$ | Choose another mapping. |
| ... | ... | Continue ... |
| 25 | $\Omega = \{\{X \backslash c\}, \{Y \backslash e\}, \{X \backslash c, Y \backslash e\}\}$ | Return result |

Figure 5.1: Tracing the faulty algorithm on two unifiable molecules

### 5.2.3 Problem with the Faulty "Unification" Algorithm

The problem is the position of $\Omega$ in the algorithm. It accumulates partial results as the possible unifier is incrementally constructed, rather than just the unifiers themselves, due to its wrong position in the algorithm. Specifically, $\Omega$ should be updated in the *outer* **foreach** loop, not the *inner* one.

## 5.3 Correct Unification Algorithm for F-Logic Molecules

ALGORITHM 2 is the corrected unification algorithm for F-Logic molecules. The reader can verify that this algorithm does not suffer from the "saving" of substitutions which are not unifiers in $\Omega$.

**Theorem 1 (Correctness)** *ALGORITHM 2 correctly computes the complete set of*

| Line no | variables | explanation |
|---|---|---|
| None | $T_1 = a[b \rightarrow c, d \rightarrow e]$ | Input molecule 1 |
| None | $T_2 = a[b \rightarrow X, d \rightarrow f]$ | Input molecule 2 |
| 2 | $\Theta = \emptyset$ | Heads unifiable |
| 9 | $\Omega = \{\}$ | Initialize $\Omega$ |
| 10 | $\lambda[1](a[b \rightarrow c]) = a[b \rightarrow X]$, $\lambda[1](a[d \rightarrow e]) = a[d \rightarrow f]$ | 4 possible mappings: $\lambda[1], \lambda[2], \lambda[3], \lambda[4]$. |
| 11 | $\sigma_{\lambda[1]} = \emptyset$ | Current solution |
| 12 | $\varphi[1] = a[b \rightarrow c]$ | Get atom |
| 13 | $\psi = a[b \rightarrow X]$ | Get corresponding atom |
| 15 | $\overrightarrow{S_1} =< b, c >$ | |
| 16 | $\overrightarrow{S_2} =< b, X >$ | |
| 18 | $\sigma_{\lambda[1]} = \{X \backslash c\}$ | Two atoms unify |
| 22 | $\Omega = \{\{X \backslash c\}\}$ | Problem! Partial result saved. |
| 12 | $\varphi[2] = a[d \rightarrow e]$ | Get atom |
| 13 | $\psi = a[d \rightarrow f]$ | Get corresponding atom |
| 15 | $\overrightarrow{S_1} =< d, e >$ | |
| 16 | $\overrightarrow{S_2} =< d, Y >$ | |
| 20 | | Two atoms do not unify |
| 10 | $\lambda[2](a[b \rightarrow c]) = a[b \rightarrow X]$, $\lambda[2](a[d \rightarrow e]) = a[b \rightarrow X]$ | Choose another mapping. |
| ... | ... | Continue ... |
| 25 | $\Omega = \{\{X \backslash c\}\}$ | Return result |

Figure 5.2: Tracing the faulty algorithm on two non-unifiable molecules

*unifiers from $T_1$ into $T_2$.*

PROOF (Outline). ALGORITHM 2 utilizes any standard term unification algorithm of first order logic (implemented by the assumed function "UNIFY") to unify id-terms, as in the original algorithm. The object ids, method names, method parameters and values of methods are all id-terms, so any standard term unification algorithm can be used to unify them. Nested objects (e.g. $a[b \rightarrow c[d \rightarrow e]]$) are not handled by the algorithm, but as explained in [74] such expressions are just syntactic sugar for "flat" expressions (e.g. $a[b \rightarrow c] \wedge c[d \rightarrow e]$). The algorithm effectively unifies one list of id-terms, i.e. object ids, method names, method parameters and values of the first molecule with a corresponding list of id-terms of the second molecule, composing the partial unifiers to get the final result (lines 11 and 18). The only complication is that atoms of the first molecule need to be mapped to atoms of the second molecule

**Algorithm 2:** Correct unification algorithm for F-Molecules

```
Input: A Pair of molecules, $T_1$ and $T_2$
Output: $\Omega$- a complete set of mgu's of $T_1$ and $T_2$
1  if $id(T_1)$ and $id(T_2)$ are unifiable then
2      $\Theta := UNIFY(< id(T_1) >, < id(T_2) >)$
3  else
4      Stop: $T_1$ and $T_2$ are not unifiable
5  end
6  if $T_1$ is of the form $S[]$ (a degenerated molecule) then
7      Stop: $\Theta$ is the only mgu
8  end
9  $\Omega := \{\}$.
10 foreach mapping $\lambda \in Map(T_1, T_2)$ do
11     $\sigma_\lambda := \Theta$.
12     foreach atom $\varphi$ in $atoms(T_1)$ do
13         Let $\psi$ be $\lambda(\varphi)$.
14         Unify tuples $\sigma_\lambda(\overrightarrow{S_1})$ and $\sigma_\lambda(\overrightarrow{S_2})$, where
15                 $\overrightarrow{S_1} =< method(\varphi), arg_1(\varphi), \ldots, arg_n(\varphi), val(\varphi) >$ and
16                 $\overrightarrow{S_2} =< method(\psi), arg_1(\psi), \ldots, arg_n(\psi), val(\psi) >$.
           /* if $val(\varphi)$ or $val(\psi)$ is $\emptyset$, they are treated as
              constant $\emptyset$ for the unification purposes    */
17         if $\sigma_\lambda(\overrightarrow{S_1})$ and $\sigma_\lambda(\overrightarrow{S_2})$ unify then
18             $\sigma_\lambda := UNIFY(\sigma_\lambda(\overrightarrow{S_1}), \sigma_\lambda(\overrightarrow{S_2})) \circ \sigma_\lambda$
19         else
20             Discard this $\sigma_\lambda$ and jump out of the inner foreach to select another
               $\lambda$.
21         end
22     end
23     $\Omega := \Omega \cup \{\sigma_\lambda\}$.
24 end
25 return $\Omega$, a complete set of mgu's of $T_1$ into $T_2$
```

and unifiers for each different mapping need to be collected. This is achieved through the outer loop, starting at line 10. Finally, line 23 collects all unifiers for different mappings inside $\Omega$. Critically, only real unifiers from $T_1$ into $T_2$ are collected in $\Omega$.

## 5.4 Discussion

Implementations of F-Logic include the Florid system [56] and the Flora-2 system [72]. Flora-2 does not use the unification algorithm presented in [74], but Florid probably does. However, we have not detected any problems with the unification of molecules in Florid, and if it does actually use the algorithm, its implementors seem to have fixed it.

# Chapter 6

# CONCLUSION AND FUTURE RESEARCH DIRECTIONS

Using a sublanguage of F-logic (which we called FLOG4SWS) 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, 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 illustrated the use FLOG4SWS through the specification of Web services, goals and ontologies in 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.

In our research, in order to obtain the necessary background to understand the issues related to Semantic Web service specification, matching and discovery, we also evaluated the WSMO Semantic Web service framework, and the WSML language through an in-depth study of the WSMO and WSML documentation and the specification of an e-health appointment-making Web service in WSML-Rule. Our investigation has revealed several deficiencies and flaws with WSMO and WSML, and guided us in the design of FLOG4SWS and the implementation of the intelligent matchmaker agent for Semantic Web services. Our studies into F-Logic led to the discovery of a mistake in the original unification algorithm for F-Logic molecules, and we presented a corrected version of the algorithm in this thesis as well.

Future research to continue the work presented in this thesis includes the specification of a variant of WSML based on FLOG4SWS as well as the implementation of a matcher using this new variant. Such a development would, in our opinion, help make WSML a practical language and gain wider acceptance in industry.

# REFERENCES

[1] "D14v1.0. ontology-based choreography, WSMO final draft 15 february 2007," http://www.w3.org/TR/soap/. Last visited: 13 August 2012.

[2] "HTTP - hypertext transfer protocol overview," http://www.w3.org/Protocols/. Last visited: 13 August 2012.

[3] "Ncoic: Network centric operations industry consortium," https://www.ncoic.org/home. Last visited: 14 April 2014.

[4] "OWL web ontology language overview," http:www.w3.org/TR/owl-features/. Last visited: 13 August 2012.

[5] "Owl web ontology language reference, w3c recommendation," http://www.w3.org/TR/2004/REC-owl-ref-20040210/. Last visited: 10 September 2012.

[6] "RDF - semantic web standards," http://www.w3.org/RDF/. Last visited: 13 August 2012.

[7] "Registering and discovering web services," http://www.cs.colorado.edu/k̃ena/classes/7818/f08/lectures/lecture_4_uddi.pdf. Last visited: 17 April 2014.

[8] "Restful web services: The basics," http://www.ibm.com/developerworks/library/ws-restful/. Last visited: 1 August 2014.

[9] “RIF rif basic logic dialect, w3c working draft,” http://www.w3.org/TR/rif-bld/. Last visited: 10 September 2012.

[10] “Semantic web,” http://semanticweb.org/wiki/Main_Page. Last visited: 21 April 2014.

[11] “Semantic web services,” http://en.wikipedia.org/wiki/Semantic_Web_Services. Last visited: 17 April 2014.

[12] “Semantic web services,” http://en.wikipedia.org/wiki/Semantic_Web_Services. Last visited: 21 April 2014.

[13] “Soap introduction,” http://www.w3schools.com/webservices/ws_soap_intro.asp. Last visited: 14 April 2014.

[14] “Soap: Simple object access protocol,” http://en.wikipedia.org/wiki/SOAP. Last visited: 14 April 2014.

[15] “SOAP specifications,” http://wsmo.org/TR/d24/d24.2/v0.1/d24-2v01_20070427.pdf. Last visited: 13 August 2012.

[16] “Tim berners-lee,” http://en.wikipedia.org/wiki/Tim_Berners-Lee. Last visited: 10 April 2014.

[17] “Uddi (universal description, discovery, and integration),” http://searchsoa.techtarget.com/definition/UDDI. Last visited: 14 April 2014.

[18] "UDDI version 3.0.2," http://uddi.org/pubs/uddi_v3.htm. Last visited: 13 August 2012.

[19] "Universal description discovery and integration," http://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration. Last visited: 14 April 2014.

[20] "A very short history of web services," http://my.safaribooksonline.com/book/web-development/web-services/9780596157708/beyond-the-flame-wars/i_sect17_d1e16314. Last visited: 14 April 2014.

[21] "Web 1.0," http://en.wikipedia.org/wiki/Web_1.0. Last visited: 8 April 2014.

[22] "Web 3.0," http://en.wikipedia.org/wiki/Web_2.0#Web_3.0. Last visited: 8 April 2014.

[23] "Web service," http://en.wikipedia.org/wiki/Web_service#cite_note-1. Last visited: 14 April 2014.

[24] "Web service definition language (WSDL)," http://www.w3.org/TR/wsdl. Last visited: 13 August 2012.

[25] "Web service modeling ontology," http://www.wsmo.org/. Last visited: 13 August 2012.

[26] "Web services description language," http://en.wikipedia.org/wiki/Web_Services_Description_L
Last visited: 14 April 2014.

[27] "Web services description language (wsdl) 1.1," http://www.w3.org/TR/wsdl. Last visited: 14 April 2014.

[28] "Web services discovery," http://en.wikipedia.org/wiki/Web_Services_Discovery. Last visited: 17 April 2014.

[29] "Web services glossary," http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice. Last visited: 14 April 2014.

[30] "Web services modelling toolkit," http://wsmt.sourceforge.net. Last visited: 13 August 2012.

[31] "Workshop on web services," http://www.w3.org/2001/03/WSWS-popa/paper08. Last visited: 14 April 2014.

[32] "WSML language reference," http://www.wsmo.org/TR/d16/d16.1/v1.0/. Last visited: 13 August 2012.

[33] A. Agarwal, "Web 3.0 concepts explained in plain english," *Retrieved Sept*, vol. 7, p. 2009, 2009.

[34] R. Akkiraju, J. Farrell, J. A. Miller, M. Nagarajan, A. Sheth, and K. Verma, "Web service semantics-wsdl-s," 2005.

[35] R. Akkiraju and B. Sapkota, "Semantic annotations for wsdl and xml schema usage guide," *Working group note, W3C*, 2007.

[36] S. Arroyo, E. Cimpian, J. Domingue, C. Feier, D. Fensel, B. Knig-Ries, H. Lausen, A. Polleres, and M. Stollberg, "Web service modeling ontology primer," *W3C Member Submission*, 2005.

[37] S. Battle, A. Bernstein, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, D. McGuinness *et al.*, "Semantic web services framework (swsf) overview," *World Wide Web Consortium, Member Submission SUBM-SWSF-20050909*, 2005.

[38] Z. Bayram and O. Sharifi, "Unifying f-logic molecules: a rectification to the original unification algorithm," in *Journal of Logic and Computation (Accepted for publication)*, 2014.

[39] A. B. Bener, V. Ozadali, and E. S. Ilhan, "Semantic matchmaker with precondition and effect matching using swrl," *Expert Systems with Applications*, vol. 36, no. 5, pp. 9371–9377, 2009.

[40] T. Berners-Lee and J. Hendler, "Scientific publishing on the semantic web," *Nature*, vol. 410, pp. 1023–1024, 2001.

[41] T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann, "World-wide web: the information universe," *Internet Research*, vol. 2, no. 1, pp. 52–58, 1992.

[42] A. J. Bonner and M. Kifer, "An overview of transaction logic," *Theoretical Computer Science*, vol. 133, no. 2, pp. 205–265, 1994.

[43] A. J. Bonner and M. Kifer, "A logic for programming database transactions," in *Logics for databases and information systems*. Springer, 1998, pp. 117–166.

[44] Y. Chabeb, S. Tata, and A. Ozanne, “Yasa-m: A semantic web service matchmaker,” in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. IEEE, 2010, pp. 966–973.

[45] J. de Bruijn, “Wsml language reference, deliverable d16. 1v1. 0,” *WSML Final Draft*, pp. 08–08, 2008.

[46] J. De Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, M. Kifer, B. König-Ries, J. Kopecky, R. Lara, E. Oren *et al.*, “Web service modeling ontology (wsmo),” *Interface*, vol. 5, p. 1, 2005.

[47] J. de Bruijn, “Wsml deliverable d16.3 v1.0, wsml abstract syntax and semantics wsml final draft,” Tech. Rep., 2008.

[48] J. De Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, M. Kifer, B. König-Ries, J. Kopecky, R. Lara, E. Oren *et al.*, “Web service modeling ontology (wsmo),” *Interface*, vol. 5, p. 1, 2006.

[49] E. Della Valle and D. Cerizza, “Cocoon glue: a prototype of wsmo discovery engine for the healthcare field,” in *Proceedings of 2nd WSMO Implementation Workshop WIW*, vol. 2005, 2005.

[50] T. Di Noia, E. Di Sciascio, and F. M. Donini, “Semantic matchmaking as non-monotonic reasoning: A description logic approach,” *Journal of Artificial Intelligence Research*, vol. 29, no. 1, pp. 269–307, 2007.

[51] R. S. Egon Börger, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 1984.

[52] J. Fan, B. Ren, and L.-R. Xiong, “An approach to web service discovery based on the semantics,” in *Fuzzy Systems and Knowledge Discovery*. Springer, 2005, pp. 1103–1106.

[53] D. Fensel and C. Bussler, “The web service modeling framework wsmf,” *Electronic Commerce Research and Applications*, vol. 1, no. 2, pp. 113–137, 2002.

[54] D. Fensel, F. Facca, E. Simperl, and I. Toma, *Semantic web services*. Springer-Verlag New York Inc, 2011.

[55] D. Fensel, H. Lausen, A. Polleres, J. d. Bruijn, M. Stollberg, D. Roman, and J. Domingue, *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[56] J. Frohn, R. Himmeröder, P.-T. Kandzia, G. Lausen, and C. Schlepphorst, “Florid: A prototype for f-logic,” in *Proceedings of the Thirteenth International Conference on Data Engineering*, ser. ICDE ’97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 583–. [Online]. Available: http://dl.acm.org/citation.cfm?id=645482.653427

[57] J. H. Gallier, *Logic for computer science: foundations of automatic theorem proving*. Harper & Row Publishers, Inc., 1985.

[58] A. Galton, “Temporal logic,” http://plato.stanford.edu/archives/win2002/entries/logic-temporal/.

[59] M. Gelfond and V. Lifschitz, “The stable model semantics for logic programming,” in *Proceedings of the 5th International Conference on Logic programming*, vol. 161, 1988.

[60] J. Gillies and R. Cailliau, *How the Web was born: The story of the World Wide Web*. Oxford University Press, USA, 2000.

[61] O. GmbH, "How to write f-logic-programs," Electronically available from http://www.semafora-systems.com/documents/tutorial _flogic.pdf.

[62] N. Griffiths and K.-M. Chao, *Agent-based service-oriented computing*. Springer, 2010, vol. 1.

[63] S. Grimm, B. Motik, and C. Preist, "Matching semantic service descriptions with local closed-world reasoning," in *The Semantic Web: Research and Applications*. Springer, 2006, pp. 575–589.

[64] B. Grosof, I. Horrocks, R. Volz, and S. Decker, "Description logic programs: Combining logic programs with description logic," in *Proceedings of the 12th international conference on World Wide Web*. ACM, 2003, pp. 48–57.

[65] W. W. Group *et al.*, "D16. 1v1. 0. wsml language reference," *WSML Working Draft*, 2008.

[66] T. Gruber *et al.*, "Toward principles for the design of ontologies used for knowledge sharing," *International journal of human computer studies*, vol. 43, no. 5, pp. 907–928, 1995.

[67] E. Ilhan and A. Bener, "Improved service ranking and scoring: Semantic advanced matchmaker (sam) architecture," *Evaluation of Novel Approaches to Software Engineering (ENASE 2007), Barcelona*, 2007.

[68] M. S. S. M. James McGovern, Sameer Tyagi, *Java Web Services Architecture*. Morgan Kaufmann, 2003.

[69] T. Kawamura, J.-A. De Blasio, T. Hasegawa, M. Paolucci, and K. Sycara, "Preliminary report of public experiment of semantic service matchmaker with uddi business registry," in *Service-Oriented Computing-ICSOC 2003*. Springer, 2003, pp. 208–224.

[70] U. Keller, R. Lara, H. Lausen, and D. Fensel, "Semantic web service discovery in the wsmo framework," *Semantic Web: Theory, Tools and Applications. Idea Publishing Group*, 2006.

[71] U. Keller, R. Lara, A. Polleres, I. Toma, M. Kifer, and D. Fensel, "Wsmo web service discovery," *WSML Working Draft D*, vol. 5, 2004.

[72] M. Kifer, "Nonmonotonic reasoning in flora-2," in *Proceedings of the 8th international conference on Logic Programming and Nonmonotonic Reasoning*, ser. LPNMR'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1007/11546207_1

[73] M. Kifer and G. Lausen, "F-logic: a higher-order language for reasoning about objects, inheritance, and scheme," in *ACM SIGMOD Record*, vol. 18, no. 2. ACM, 1989, pp. 134–146.

[74] M. Kifer, G. Lausen, and J. Wu, "Logical foundations of object-oriented and frame-based languages," *Journal of the ACM (JACM)*, vol. 42, no. 4, pp. 741–843, 1995.

[75] M. Klusch and F. Kaufer, “Wsmo-mx: A hybrid semantic web service matchmaker,” *Web Intelligence and Agent Systems*, vol. 7, no. 1, pp. 23–42, 2009.

[76] U. Küster and B. König-Ries, “Evaluating semantic web service matchmaking effectiveness based on graded relevance,” in *The 7th International Semantic Web Conference*, 2008, p. 35.

[77] R. Lara, D. Roman, A. Polleres, and D. Fensel, “A conceptual comparison of wsmo and owl-s,” *Web Services*, pp. 254–269, 2004.

[78] R. Lara and D. Olmedilla, “Discovery and contracting of semantic web services,” in *W3C Workshop on Frameworks for Semantic in Web Services, Innsbruck, Austria*, 2005.

[79] S.-L. Lee, “How to derive a new deductive and object-oriented database system from a well-defined f-logic database,” Ph.D. dissertation, Evanston, IL, USA, 1992, uMI Order No. GAX92-29947.

[80] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne *et al.*, “Owl-s: Semantic markup for web services,” *W3C member submission*, vol. 22, pp. 2007–04, 2004.

[81] W. May, “How to write f-logic programs in florid,” Technical report, Institut für Informatik, Universität Freiburg, Tech. Rep., 2000.

[82] S. A. McIlraith, T. C. Son, and H. Zeng, “Semantic web services,” *IEEE intelligent systems*, vol. 16, no. 2, pp. 46–53, 2001.

[83] H. W. C. Z. Michael Kiefer, Guizhen Yang, "Flora-2: Users' manual (version 0.99.3)," *Electronically available from flora.sourceforge.net/docs/floraManual.pdf*, 2013.

[84] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, "Importing the semantic web in UDDI," in *Web Services, E-Business, and the Semantic Web*. Springer, 2002, pp. 225–236.

[85] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, "Semantic matching of web services capabilities," in *The Semantic WebSWC 2002*. Springer, 2002, pp. 333–347.

[86] M. Rambold, H. Kasinger, F. Lautenbacher, and B. Bauer, "Towards autonomic service discovery a survey and comparison," in *Services Computing, 2009. SCC'09. IEEE International Conference on*. IEEE, 2009, pp. 192–201.

[87] J. Scicluna, R. Lara, A. Polleres, and H. Lausen, "D4. 2v0. 1 formal mapping and tool to owl-s," 2004.

[88] O. Sharifi and Z. Bayram, "Specifying banking transactions using web services modeling language (WSML)," in *Accepted to The fourth International Conference on Information and Communication Systems (ICICS 2013)*, Irbid,Jordan, 23-25 April 2013.

[89] O. Sharifi and Z. Bayram, "Sharifi-bayram SWS goals matchmaker," https://sourceforge.net/projects/sws-goal-matchmaker/files/.

[90] N. Srinivasan, M. Paolucci, and K. Sycara, "Adding OWL-S to UDDI, implementation and throughput," *Proceeding of Semantic Web Service and Web Pro-*

*cess Composition 2004*, 2004.

[91] N. Srinivasan, M. Paolucci, and K. Sycara, "An efficient algorithm for OWL-S based semantic search in UDDI," in *Semantic Web Services and Web Process Composition*. Springer, 2005, pp. 96–110.

[92] N. Steinmetz, "Navigation (discovery wg): Discovery," http://wiki.wsmx.org/index.php?title=Discovery.

[93] M. Stollberg, "Scalable semantic web service discovery for goal-driven service-oriented architectures," Ph.D. dissertation, PhD thesis, Faculty of Mathematics, Computer Science and Physics Leopold-Franzens University Innsbruck, Austria (March 2008), 2008.

[94] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan, "Automated discovery, interaction and composition of semantic web services," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 1, pp. 27–46, 2003.

[95] K. P. Sycara, M. Klusch, S. Widoff, and J. Lu, "Dynamic service matchmaking among agents in open information environments," *Sigmod Record*, vol. 28, no. 1, pp. 47–53, 1999.

[96] H. Wan, B. Grosof, M. Kifer, P. Fodor, and S. Liang, "Logic programming with defaults and argumentation theories," in *Logic Programming*. Springer, 2009, pp. 432–448.

[97] H. Wang, N. Gibbins, T. Payne, and D. Redavid, "A formal model of the semantic web service ontology (wsmo)," *Information Systems*, 2011.

[98] C. W.F.Clocksin, *Programming in Prolog*. Springer-Verlag, 1984.