A Critical Evaluation of Web Service Modeling Language

Şengül Çobanoğlu

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

Master of Science
in
Computer Engineering

Eastern Mediterranean University
February 2013
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 Master of Science in Computer Engineering.

Assoc. Prof. Dr. Muhammed Salamah
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 Master of Science in Computer Engineering.

Assoc. Prof. Dr. Zeki Bayram
Supervisor

Examining Committee

1. Prof. Dr. Erden Başar
2. Assoc. Prof. Dr. Zeki Bayram
3. Assist. Prof. Dr. Ahmet Ünveren

ABSTRACT

The aim of this thesis is to analyze and evaluate the Web service Modeling Language (WSML) as the formal language of Web service Modeling Ontology (WSMO) and then provide a number of improvement recommendations for obtained deficiencies in WSML. In order to facilitate understanding of our work, this thesis also briefly provides background information about web services, semantic web, semantic web ontology languages, Web Service Modeling Ontology and conceptual syntax of Web Service Modeling Language as well as logical formalism used by WSML.

In this thesis, WSML has been critically analyzed and evaluated in detail by developing semantic web service for "University Course Registration" using the WSML rule variant and first order logic. At the end of the thesis the weak and missing parts of WSML were defined and possible suggestions were made for improvement.

Keywords: Web service modeling language, web service modeling ontology, web services, semantic web, ontology.

ÖZ

Bu tezin amacı, web servisi geliştirmek için, Web Servisleri Modelleme Ontolojisi tarafandan, formal bir dil olarak kullanılan Web Servisi Modelleme dilini detaylı olarak inceleyerek değerlendirmek ve WSML'in eksik yönlerinini ortaya çıkartıp, geliştirici bazı tavsiyerde bulanmaktır. Ayrıca tezde, yaptığımız çalışmanın daha iyi anlaşılması için web servisleri, anlamsal ağ, anlamsal ağ ontoloji dilleri, web servisi modelleme ontolojisi ve web servisi modelleme dili hakkında kısa açıklamalar verilmektedir.

Bu tez de , WSML rule ve first order logic kullanılarak tanımladığımız " üniversite ders kayıt" web servisi üzerinden Web Servisi Modelleme dilini eleştrisel olarak inceleyip ve sonucunda Web Servisi Modelleme dilinin güçsüz ve eksik yönlerini bularak, WSML'in geliştirilmesi için mümkün olan önerilerde bulunduk.

Anahtar Kelimeler: Web servisi modelleme dili, web servisleri modelleme ontolojisi, web servisi, anlamsal ağ, ontoloji.

To my mother Yeter Çobanoğlu; always devoted, loving and caring,

&

To my sister Esma Kerçin ; always helped, encouraged,

&

To my fiance Mounir Debbouza; always supported, encouraged .

ACKNOWLEDGMENT

Sincerely, I would like to thank my advisor Assoc. Prof. Dr. Zeki Bayram for accepting to be my supervisor, for continuous support of my study and research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my master’s study.

In addition, I would like to thank my family, especially my mother, sister and fiancé for their continuous love and support in my decisions.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Nowadays, World Wide Web (WWW) [1] has become very huge, full of text and a lot of unrelated documents which is primarily designed for human interpretation and use [2]. Therefore, finding reliable data on the web is very hard and time consuming [3]. For example, when we search about something, usually we retrieve many unrelated data or documents. In order to find useful document we have to scan all retrieved documents and determine whether the retrieved documents are relevant or not. This method takes too much time and provides an unreliable searching method [3]. Because of that, current web technologies do not thoroughly satisfy our needs [4]. There should be new web technologies which enable computer systems to understand contents of web and find out the exact information that we are looking for. This can be achieved through semantic web. Semantic web provides web contents that machine can understand semantically and provide exact data that we are searching about [5]. Thus, web will become more effective and efficient for both human and machines.

In immediate future, with the semantic web, it can be thought that all web content will become as one huge database and everything will be linked with each other in this database [5] and WWW will enable computer systems to interact with each other to share and use data without any human interaction. Also all of the detailed and specific information that user desires will be provided by WWW [6].

Furthermore, semantic web enabled web services [7] which is fundamental part of the semantic web will transform the web from a collection of information into a distributed computational device. Web services can be developed and described using the model of Web Service Modeling Framework (WSMF) [8] which will provide the appropriate conceptual model for developing web services [8]. In addition, a Web service modeling language such as Web service Modeling Language (WSML) [9] will be used to model web services.

Today, there are many tools, languages and approaches that exist with the aim of building semantic web services. In this thesis, a “university course registration” web service specification is created using Web Service Modeling Language (WSML) [9] with the aim of discovering possible areas of improvement. This exercise has revealed some weakness and deficiencies, which we present as our contribution.

1.1. Structure of the Thesis

The thesis is structured in the following way: Chapter 1 is the introduction, Chapter 2 defines history of web, web services, semantic web, semantic web technologies and semantic web ontology languages, Chapter 3 introduces the Web Service Modeling Ontology (WSMO) [10], its formalism, the Web Service Modeling Language (WSML) [9], its execution environment the Web Service Execution Environment (WSMX) [11] and its modeling toolkit Web service Modeling Toolkit (WSMT) [12]. Chapter 4 provides specification of “university course registration” web service by using web service modeling language, Chapter 5 defines the deficiencies discovered in WSML through the university course registration specification, and it also includes suggestions for improvement. Finally, Chapter 6 is related with further research and conclusion.

Chapter 2

THE WORLD WIDE WEB (WWW)

2.1. Brief History of Web

World Wide Web (WWW) [1] was introduced by the development of HTML (Hyper Text Markup Language) [13] by Tim Berners Lee who was a computer programmer in CERN (European Organization for Nuclear Research) [14] in 1989 [15]. In 1990 web browsers were developed which were used for searching HTML documents which changed the way information published and broadcast [16].

HTML (Hypertext Markup Language) Web Pages (Web 1.0) were in the first decades of World Wide Web [17] which only provided publishing HTML web pages containing static information [13]. Users were able to read the web contents only and could not interact with other web site users. After Web 1.0, Dynamic Web Pages (Web 2.0) has been released [17], which enables two way communications and provides dynamic web pages which means web pages can be updated by users easily [18].

2.2. Web Services

2.2.1. What is a Web Service?

The Web Services Architecture Working Group defines a Web Service as: "A software application identified by an URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artifacts. Web Service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols. Web service is a specification that is available over the World Wide Web" [19].

Web services can be discovered by finder mechanism of any requester (human or machine). Therefore, web service enables any software to communicate with each other without interoperability problem [20]. For example, while Java based applications can communicate with php based applications, Windows applications can also communicate with Unix applications [21].

2.2.2. Web Service Technologies

Web services describe the web based applications using web service technologies which are communicated through XML based messaging system [22]. Therefore, XML is the basis of web services which allows different systems to exchange data over the web regardless of their hardware, programming language, operating system, platforms and frameworks [23].

The Web services framework is composed of the following parts: communication protocols, service invocation, service descriptions, and service discovery [22]. In figure 2.1, web services frameworks can be seen in which service consumers connect service provider via SOAP, service registrar publish web services via UDDI and service provider sends description of web services to service consumers via WSDL, then service consumers find web services via UDDI.

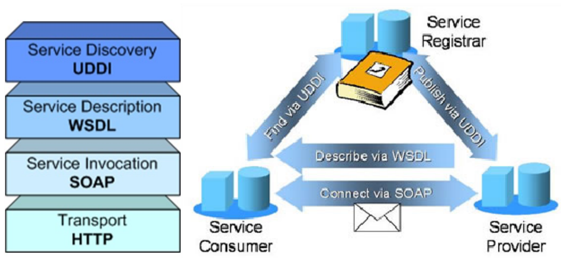


Figure 2.1: Web services frameworks

At the lowest level, Simple Object Access Protocol (SOAP) [24] appears which is a protocol for messaging format to access web services and communicate regardless of platforms. [25] In other words, "SOAP is the envelope syntax for sending and receiving XML messages with web services sent over HTTP" [21]. According to SOAP protocol, messages have to be enveloped as XML document that contains header and body [24]. Header specifies data about body which is optional. The body part contains the name of web service and information request from web service or information responded by web service. SOAP envelops travel over Hypertext Transfer Protocol (HTTP) [26] which is a protocol for exchanging and transferring hypertext documents over the internet [26].

After SOAP messages are defined, WSDL (Web Service Description Language) [27] is needed to describe a set of SOAP messages and it defines the information necessary for a client station in order to interact with the Web Service [28]. For example, where is the web service located, what is the functionality of web service and how is it possible to communicate with it?

After getting the location and functionality of web service with the help of WSDL, there is a need for a discovery mechanism to find web service. Therefore, the function of Universal Description is needed. Discovery and Integration (UDDI) [29] provides a standard discovery mechanism for Web services [23]. Moreover, UDDI is a directory of web service interfaces that is described by WSDL. Web service consumers can be registered on UDDI to access and locate their web services [22].

2.2.3. Discovery of Web Services

Nowadays, web search engine is used in order to retrieve relevant information over the web. However, when searching web services based on their provided functionality, the information retrieval method does not work for retrieving the web services properly [30]. Since web services are described using WSDL web services can be discovered through UDDI.

2.3. OWL-S

The Web Ontology Language for Services (OWL-S)[31] is used to describe functionality of web services, such as providing information about what the properties of service are, how it can be interacted with and how it can be used. OWL-S has three sub concepts, ServiceProfile, ServiceModel and Servicegrounding [32]. ServiceProfile specifies the purpose of the service, what it provides and what kind of information is needed to discover the web service [33]. ServiceModel defines how the service works and how it can be interacted with the web service. Service grounding concept describes how web service works and which kind of information is needed to access web service [32].

2.4. SWSF (Semantic Web Service Framework)

Semantic Web Services Framework (SWSF) [34] is composed of the Semantic Web Services Language (SWSL) [35] and the Semantic Web Services Ontology (SWSO) [36].

Semantic Web Services Language (SWSL) [35] is developed for the purpose of describing semantic web service ontology such as descriptions of individual services and formal characterizations of concepts [35].

The Semantic Web Services Ontology (SWSO) [36] is a theoretical model, which Web Services can be illustrated, and an axiomatization or formal characterization for that model [34].

2.5. Semantic Web

Nowadays web is huge and growing. There are plenty of web pages that are published every day. It is very hard to find real data and that is due to the volume of information the web contains. In addition to that, in today's web everything is not machine readable, all useful contents of web pages are hidden in the media (text, pictures, videos) which are readable by humans only. New mechanism is needed that allows machine to understand content of web and save our time.

With the aim of the retrieving, extracting and integrating of web services automatically through World Wide Web, Tim Berners Lee initialized the Semantic web. According to Berners Lee, Hendler and Lissila [6],

> "*The majority of the content of the web is designed for humans to read and not for computer programs to manipulate in meaningful way. The Semantic Web will bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users. The Semantic Web is an extension of the current web in which information is given well defined meaning, better enabling computers and people to work in cooperation*".

2.5.1. Architecture of the Semantic Web

The semantic web architecture proposed by Berne's-Lee in 2001 is as a layered pyramid [37]. It includes several layers which provide core functionalities and play a main role for semantic web [38]. In this part, the features of the Semantic Web of technologies will be briefly explained and highlighted. Furthermore, XML, RDF and OWL will be defined in detail under semantic web ontology language topic. Figure 2.2 depicts the "Semantic Web Layer cake" which includes core elements of overall semantic web architecture [38].

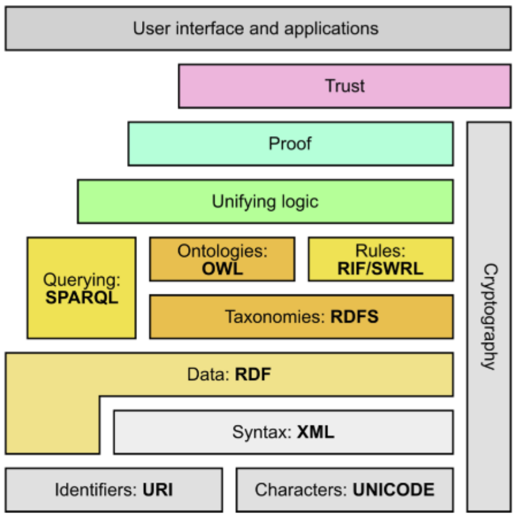


Figure 2.2: The semantic web layer cake [38]

At the lowest layer of the semantic web architecture URI, UNICODE and XML are found. Uniform Resource Identifier (URI) [39] is a structured string that is used to identify source of web services on the internet, each web service source has unique URI which is linking them to each other [39].

Unicode [40] is a standard for encoding characters which provides unique numbers for one million characters, regardless of the platform, program, language [40]. It is primarily designed to facilitate the job of developers who would like to create a Multilanguage software applications.

Extensible Markup Language (XML) [41] separates web contents from its presentation and enables web content to be structured and meaningful by proving us to define data between our own tags [6]. Moreover, XML is designed to store web content with its meaning. Thus, machines can read xml files and process them.

Resource Description Framework is a standard model and language used to describe web sources and relations between them to exchange data on the web [42]. It uses XML and triples, such as subject, object and predicate for representing web sources, in which triples can be URI or string literal [6]. In addition, RDF is designed to be read and understood by machines only rather than by humans.

SPARQL Query Language [43] is a standard language used to query RDF data. SPARQL is like SQL but it is designed for matching graph patterns providing functionality like optional parts, nesting, union of patterns, filtering values of possible matching, and the option of choosing the data source to be matched by a pattern [44].

SPARQL Update is a language that is used to update RDF graphs [16]. SPARQL Update serves the following facilities: Inserting new triples to an RDF graph, deleting triples from an RDF graph, performing a group of update operations as a single action, creating a new RDF Graph to a Graph Store and deleting an RDF graph from a Graph Store [45].

The Rule Interchange Format (RIF) is a standard for exchanging rules among rule systems, mainly among Web rule engines [46]. RIF specifies three rule based languages in order to cover aspects of Rule layer such as first-order, logic-programming and action rules [46].

Finally, the top layers, **Logic**, **Proof** and **Trust**, are under research and some simple application demos are created. The Logic layer provides the writing of rules while the Proof layer work to execute and evaluate the rules with the help of the Trust layer mechanism for applications in order to check whether to trust the given proof or not [47].

2.6. Ontology

Today's web has vast amount of text content and interlinks between them. Furthermore, web contains repetition of web contents of which Machine must understand the meaning of web and discover the common meaning of web contents in order to provide exact information to user. This problem can be solved providing semantic web in the form of collections of information called ontologies [48]. Therefore, web content will be kept on ontologies in a meaningful way by using set of taxonomies and rules. The taxonomy defines concept of attributes and their relationships whereas rules define logical constraints [6].

Ontologies can be reused. For example, ontology about something can be pointed by many web pages which reduce the heterogeneity [2].

Ontologies include concepts in a hierarchal way; hierarchy means that if a class A is a subclass of class B, then every attribute in A is also included in B. For instance, Student and Teacher concepts are sub concepts of Person concept; they inherit all attributes of Person concept. Apart from including classes and subclass, ontologies include instances (real world objects). In addition, ontologies may permit relationships between instances.

2.7. Semantic Web Ontology Languages

There are many semantic web languages used to represent ontologies such as Extensible markup language (XML) [41], Resource Description Framework (RDF) [49], Defense Advanced Research Projects Agency Markup Language (DAML) + Ontology Interface Layer (OIL), usually abbreviated as DAML+OIL and the Web Ontology Language (OWL) [4]. The important point that each web ontology language takes advantage of other languages beneath them this so called layer cake [21] can be seen in figure 2.3.

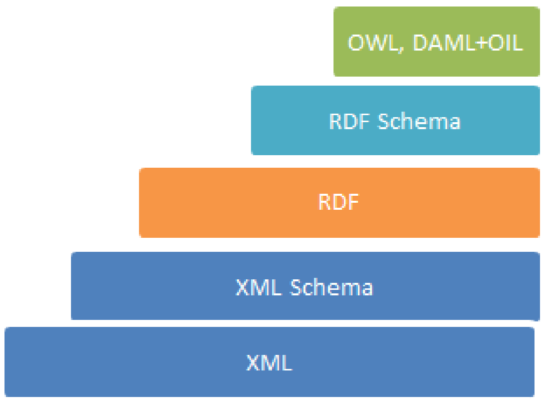


Figure 2.3: Web ontology language layer cake [21]

2.7.1. Extensible Markup Language (XML)

Extensible Markup Language (XML) [41] is a standard for describing data and relationships by using hierarchy or tree structure [41]. It defines data and relationships using tag (metadata) which enables separate web content from its presentation and provides any XML enabled system to read web page and understand its content easily [21]. Figure 2.4 shows XML tags which define information about the book.

```xml
<?xml version="1.0" encoding="ISO-8859-2"?>
<book>
<title> Programming the Semantic Web </title>
<author>  Toby Segaran, Colin Evans, Jamie Taylor </author>
<publisher> O'Reilly Media  </publisher>
<year> 2009 </year>
<ISBN> 9789512289844 </ISBN>
</book>
```

Figure 2.4: XML syntax

Additionally, XML uses Unicode encoder system which makes xml documents platform independent. [50]. Also, XML has parser for converting xml documents into xml DOM object which is a standard way for accessing and manipulating documents [51].

XML describes the purpose or meaning of raw data values via a text format which enables exchange, interoperability and application independence [21]. Therefore, XML is very powerful and essential for web services. However, XML is not enough to describe web content using metadata [41]. Since XML is based on metadata approach, there is a question of how sentences and paragraphs can be described. For instance, how following information can be encoded, “The book programming the semantic web is published by O’Reilly Media”. As can be seen in figure 2.5, there can be various ways to define this information with xml tags. Also, there can be multiple tags with the same name including different values. These problems cause obscurity and even make it harder to set up relation between different web service.

Because of this reason directed graph is used as data model which is called resource description framework (RDF).

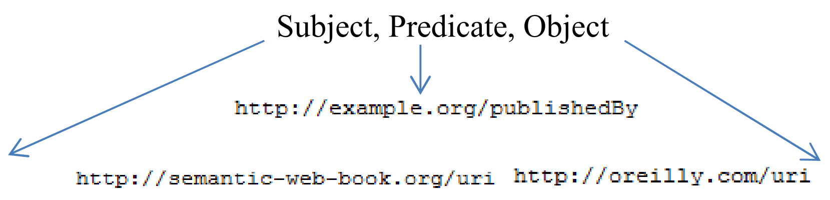
```
<?xml version="1.0" encoding="ISO-8859-2"?>
<book>
<title> Programming the Semantic Web </title>
<publisher> O'Reilly Media  </publisher>
</book>

<publisher>
<name> O'Reilly Media  </name>
<book><title> Programming the Semantic Web </title></book>
</publisher>
```

Figure 2.5: XML problems

2.7.2. Resource Description Framework (RDF)

Resource Description Framework RDF is a standard model for data interchange on the Web [42]. It defines metadata about web pages and provides a model to represent web sources and relationship between them by using XML syntax. Thus, RDF documents can be easily read and understood by machines [52].

RDF model uses triples in order to indicate information in a more clear way. For instance, it uses subject, predicate and object. *Subject* is resource, whatever thing that may contain URI [53] such as http://semantic-web-book.org/uri, *predicate* is property that describes the resource [53] such as http://example.org/publishedBy. *Object* is property value [53] such as http://oreilly.com. Property value can be literal or refers to another source. RDF triple language can be seen in figure 2.6.


Subject, Predicate, Object
http://example.org/publishedBy
http://semantic-web-book.org/uri
http://oreilly.com/uri

```
<http://semantic-web-book.org/uri>
<http://example.org/publishedBy> <http://oreilly.com>
<http://semantic-web-book.org/uri>         <http:/example.org/title>
"Programming the semantic web"
<http://semantic-web-book.org/uri>          <http:/example.org/name>
"O'Reilly Media"
```

Figure 2.6: RDF triples

A statement is formed by combining a subject, a predicate and an object. [53] For example, the book is about programming the semantic web published by O'Reilly Media. Xml definition for RDF can be seen in figure 2.7.

```
<? xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:ex=http://example.org/>

<rdf:Description rdf:about="http://semantic-web-book.org/uri>"
      <ex:publishedBy>
            <rdf:Description rdf:about="http://oreilly.com/uri>"
            </rdf:Description>
      </ex:publishedBy>
</rdf:Description>
</rdf:RDF>
```

Figure 2.7: XML definition for RDF

Since RDF defines the web sources by using subject, predicate and object in the form of sentence, **RDF schema** defines them in application specific classes and properties similar to classes in object oriented programming languages. This permits resources to be defined as instances of classes, and subclasses of classes [11]. For example, teaches is a relationship between instructor and course. It also allows you to describe in human readable text the meaning of a relationship or a class which is called a schema that provides information about legal uses of various classes and relationships. It is also used to specify that a class or property is a subtype of a more general type. For example, ”Person” is a superclass of ”Student” and ”Instructor” and ”CourseOpening” is a subclass of ”Course”. However, RDF has a number of limitations. For instance, classes cannot be disjoint, intersection or union cannot be used to create another class, cardinality restriction cannot be defined to say student can take 6 courses at most, property cannot be defined to say year must be greater than 2012 and it cannot be said that property is inverse of another property. Due to these problems, RDF schema is extended and OWL language has been built up.

2.7.3. DAML+ OIL

DAML+OIL is a semantic markup language for Web resources [54] which is designed using RDF and XML web standards in order to describe structure of a domain such as classes and properties [54] .

2.7.4. Web Ontology Language (OWL)

Web Ontology Language (OWL) is the most powerful ontology language currently defined for semantic web which provides semantic markup language in order to publish and share on semantic web [12]. In addition, OWL builds upon XML, RDF and DAML+OIL and it has greater machine interpretability of Web content than the other ontology languages by providing supplementary vocabulary along with a formal semantics [13] such as classes, subclasses, relations between classes, properties, sub properties, characteristics and restriction of properties [13]. For example, OWL can indicate that “Ali teaches cmpe318” which implies “cmpe318” is taught by “Ali”. Or if “cmpe211” is prerequisite of “cmpe318” and ”cit318” is prerequisite of “cit418” then “cit211” is “prerequisite of “cit418”. Another useful thing OWL adds is the ability to say two things are the same which is very helpful for joining up data expressed in different schemas. It can be pointed out that relationship “teaches” in one schema is owl which is the same as ”taught” in some other schema. It can also be used to say two things are the same, such as the book entitled “Programming the semantic web” published in “oreilly.com” is the same book published in “elibrary.com” which is very exciting as it means joining up data can be started from multiple sites called ”Linked Data”.

There are three sub languages of OWL:

OWL Lite provides primary needs for classification hierarchy and simple constraints [55] such as properties, concepts, instances and cardinality constraints, maxCardinality and minCardinality which can take value 0 or 1. OWL lite is the simplest OWL language and corresponds to description logic [56].

OWL DL uses some constructs of OWL and RDFs under restriction and conceptually is based on description logics. OWL DL provides more cardinality restriction, not limited with 1 and 0 like OWL lite. In addition, OWL DL enables union, intersection and complement of classes. Therefore, OWL DL supports maximum expressiveness while maintaining computational completeness and decidability [56].

OWL Full contains full OWL vocabulary and full syntactic of RDF [56].

Chapter 3

WSMO, WSML WSMX, WSMT

This chapter presents Web Service Modeling Ontology (WSMO) [57] that defines conceptualization model for web services. Web Service Modeling Language (WSML) [58] that is a formal language for the specification of WSMO elements such as ontologies, mediators, goals and web services. Web Service Execution Environment(WSMX) [11] which enables running and discovering of web services, and briefly outlines the Web Service Modeling Toolkit (WSMT) [12] which provides tools to create and edit WSMO elements visually.

3.1. Web Service Modeling Ontology (WSMO)

The Web Service Modeling Ontology (WSMO) [57] is a conceptual model for creating semantic descriptions for web services that can be used to resolve interoperability issues among web services [12]. In addition, WSMO is based on the Web Service Modeling Framework (WSMF) [12] which defines conceptual model with the aim of developing and describing web services [8] which provides conceptualization of ontologies, goals, mediators and web services. WSMO uses Web Service Modeling Language (WSML) which provides formal syntax and semantics for web services.

3.1.1. WSMO Core Elements

WSMO defines four core elements as the main concepts which have to be described in order to define Semantic Web Services[59]. They are defined briefly as follows;

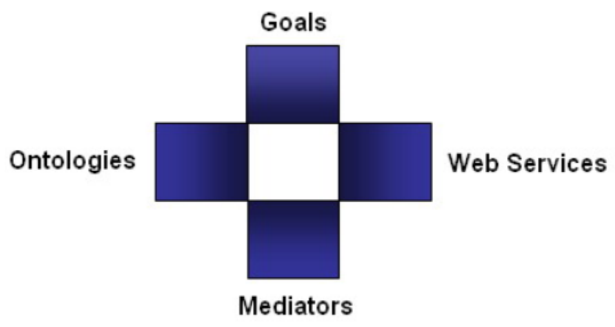


Figure 3.1: WSMO core elements [57]

Ontologies: an ontology is a formal explicit specification of a shared conceptualization [60]. It provides data model that represents the knowledge as a set of concept and allows relationship between these concepts within a domain. Moreover, ontologies allow machines to understand the semantic of information and relationship that is published on the website. Ontologies can include concepts, sub concepts, relationships, instances, relation instances, functions and axioms to define the web content semantically.

Goals can be described as web services that would potentially satisfy the user's desires [57, 59]. In other words, a goal is specification of needs that have to satisfy in order to communicate with web service.

Mediators handle possible semantic mismatches between different WSMO elements [59]. WSMO defines different kinds of mediators in order to provide connection and communication between different WSMO elements. For instance, OOMediators solve interoperability problems between two or more ontologies, GG Mediators enable to connect Goals with each other, WG Mediators link Web Services to Goals, expressing whether web service fulfills the goal or not, and WW

Mediators connect web services together and express interrelations between them such as two web services which have the same functionality [61].

Web Services describe the functionality of service through their capabilities. A web service tries to meet with user goal. If web service provides required functionality then desire of user is returned.

3.2. Web Service Modeling Language (WSML)

Web Service Modeling Language (WSML) [58] is a language specifically designed to express semantic descriptions according to the WSMO Meta model which is specified in terms of a normative human readable syntax [57]. Furthermore, WSML separates between conceptual syntax and logical expression syntax [62]. Conceptual syntax is used to model ontologies [63] by using concepts, instances, relations and relation instances. Logical expressions are used to refine ontology definitions by arbitrary rules [63] such as axioms.

This part discusses motivation about introduction of WSML. In Chapter 4, the WSML is analyzed in detail by providing examples on each of its constructs.

3.2.1. WSML Logical Expressions

WSML defines five language variants through composition of several rule based languages such as Description Logic, Logic Programming and First Oder Logic [12]. Figure 3.2 and Figure 3.3 depicts WSML variants and their interrelation.

In WSML layer, every valid specification of an extended variant is also a valid specification of the new variant [62]. WSML-Core is defined by the intersection of Description Logics and Logic Programming [9]. This variant has the least expressive power within all languages of the WSML [9]. WSML DL is the extension of WSML core and belongs to Description Logic. WSML Flight is the extension of WSML core and is based on logic programming [9]. WSML Rule extends WSML-Flight and uses Logic Programming [9]. Finally, WSML Full unifies WSML-DL and WSML-Rule under First Order logic [9].

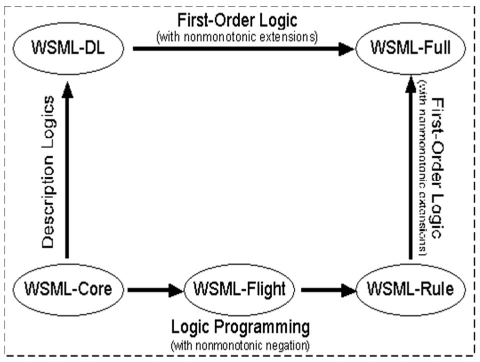


Figure 3.2: WSML variants [62]

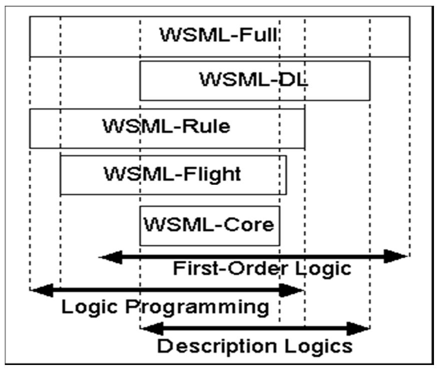
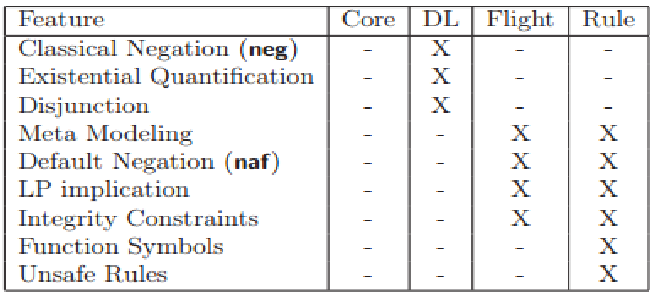


Figure 3.3: WSML variant layers

Definition of different WSML variants through the different rule based language causes complexity. Figure 3.4 compares WSML variants with their most distinct features, as can be seen in the figure, WSML Full is not mentioned because its semantic is not completed, it is still in process. However, when the definition of WSML full is completed, there are expectations about providing all features listed in the figure.



Feature	Core	DL	Flight	Rule
Classical Negation (**neg**)	-	X	-	-
Existential Quantification	-	X	-	-
Disjunction	-	X	-	-
Meta Modeling	-	-	X	X
Default Negation (**naf**)	-	-	X	X
LP implication	-	-	X	X
Integrity Constraints	-	-	X	X
Function Symbols	-	-	-	X
Unsafe Rules	-	-	-	X

Figure 3.4: Language Framework for Semantic Web Services [58]

In the thesis, university course registration specification have been built up based on WSML Rule which seems the best variants to evaluate WSML. Since it is powerful enough to model our application and has reasoner.

3.2.2. WSML Conceptual Syntax

3.2.2.1. Ontologies

Ontology in WSML consists of the elements such as concepts, relations, instance, relation instances and axioms [58].

• **Concepts:** The notion of concepts (sometimes also called 'classes') plays a central role in ontologies [58]. Concept definition in WSML starts with **concept** keyword and is followed by concept name. A concept may inherit all attributes from its super concept. Definition of inherited concept starts with keyword **SubConceptOf** then is followed by the name of the inherited concepts in curly brackets. The definition of concept and subconcept can be seen in the following figure 3.5.

Additionally, Concept can include attribute and attribute types by using **ofType** keyword. Also, concept can include nonfunctional property defined by **nonFunctionalProperties… endNonFunctionalProperties** which is optionally used to describe the concept or attribute. Besides, nonfunctional property **nfp…endnfp** keyword is used to define logical constraint for particular attribute. Furthermore, in the concept, attributes can be defined with cardinality constraints like maxcardinality and mincardinality within the open brackets.

```
concept Student subConceptOf Person
nonFunctionalProperties
          Creator  hasValue"Sengul"
endNonFunctionalProperties
     yearEnrolled  ofType  (0 1) _integer
     overallCGPA  ofType  (0 1) _float
     enrolledIn  ofType  (0 2) AcademicProgram
     semesterEnrolled  ofType  (0 1) Semester
     tookCourse_  ofType (0 *) Course
nfp
                 dc#relation  hasValue {TookRelation}
endnfp
```

Figure 3.5: Concept definition

• **Relations**: Relations are used to model interdependencies between several concepts [57]. A relation is defined with the **relation** keyword and followed by the identifier of the relation. It should be noted here that relation of parameters must be strictly ordered.

```
relation teaches(  ofType Instructor,   ofType CourseOpening)
```

Figure 3.6: Relation definition

• **Instances:** A concept represents a set of objects in a real or abstract world with a specific shared property. The objects themselves are called instances [59]. Instances are defined with the keyword **instance** and followed by instance identifier and specification of concept name by **memberOf** keyword. Furthermore, Instance values must be the same type with the corresponding attribute type declaration in the concept definition [59].

```
instance eastern_mediterranean_university memberOf University
uname hasValue"Eastern Mediterranean University"
locatedAt hasValue EMUAddress
```

Figure 3.7: Instance Definition

In addition, WSML defines relation instance with starting keyword **relationInstance** followed by identifier. Relation instances can have unlimited parameters within open brackets. Parameters must be in same order with the definition of corresponding relation.

```
relationInstance prerequisite(cmpe211,cmpe354)
```

Figure 3.8: Relation instance definition

• **Axioms:** axiom defines logical expressions. WSML defines axiom with the starting keyword **axiom** followed by axiom name. The "**definedBy**" keyword enables to define logical constraints.

```
axiom registrationRules
definedBy
clashes(?co1,?co2):-
?co1memberOf CourseOpening
and?co2memberOf CourseOpening
and?co1 != ?co2
and?co1[teaching_times  hasValue?tt1, year  hasValue?y1, semester
hasValue?s1]
and?co2[teaching_times  hasValue?tt1, year  hasValue?y1, semester
hasValue?s1].
```

Figure 3.9: Axiom definition

3.2.2.2. Goals

Goals are the desired functionality of web service [58]. Goal in WSML is defined with the keyword **goal** followed by goal identifier.

```
goal goalCourseRegistration
```

Figure 3.10: Goal definition

Goals must include one capability which defines the functionality of web service through the five core elements which are **preconditions** describing conditions to invoke web services, **postconditions** describing what the web service outputs are after invoking, assumptions specifying what must hold of the state of the world for the Web service to be able to execute successfully, and the effects describing real world effects of the execution of the Web service which are not reflected in the output [64]**.** Goals have also let to define shared variables which specify the variables that are going to be shared between capability elements.

3.2.2.3. Mediators

Mediators enable different WSMO elements to communicate with each other without any interoperability problem. There are four types of mediators that mediate mismatches between ontology- ontology, web service-web service, web service-goal, and goal-goal. In this thesis, there is no need to use mediator since within only one domain is being worked on**.**

3.2.2.4. Web Services

Web services are symmetric to Goals defining the actual functionality provided through the definition of capabilities just like goals. Web services are defined in WSML with keyword **webservice** followed by identifier as shown below.

```
webService web_service_courseRegistration
```

Figure 3.11: Web service definition

3.3. Web Service Execution Environment (WSMTX)

Web Service Execution Environment (WSMX) [25] is a middleware platform used for discovery, composition, execution and mediation of Semantic Web services [65]. WSMX environment also enables requester goals for matching with capabilities of most appropriate Web service that exists in WSMX. Moreover, WSMX enables web service requester to interact with web service provider and launches the selected Web service.

The completed WSMX system will allow service providers to describe their web services in WSML and publish these descriptions on the WSMX system. When end users send goals, described in WSML, to WSMX, these goals are matched against the capabilities of the web services registered with the WSMX system. These services can then be invoked to realize the user’s goals [12].

3.4. Web Service Modeling Toolkit (WSMT)

The Web Services Modeling Toolkit is an Integrated Development Environment (IDE) that helps developing Ontologies, Goals, Web Services and Mediators through the Web Service Modeling Ontology (WSMO) [27] formalism. The Web Service Modeling Toolkit provides a number of tools and visualizer graph to create and edit ontologies visually by using WSML syntax. Visualizer graph enables us to edit ontology directly using a graphical interface and see the changes instantly. WSMT also provides the text editor to manually create or edit semantic descriptions [12].

Chapter 4

MODELING UNIVERSITY COURSE REGISTRATION

This chapter provides a critical analysis of Web Service Modeling Language (WSML) [9] by implementing a specification of “university course registration” web service. Therefore, in this chapter, detailed explanations of WSMO framework can be seen through examples, such as ontology, goal and web service.

4.1. University Course Registration Scenario

In the scenario, a student wants to make a course registration request to the web service of the university. However, in order to make a successful registration, the student has to satisfy some steps and logical rules that the web service requires.

The steps can be given as follows:

Step 1: Students have to submit student name, course, year, and semester information to the web service through the goal. Goal is a specification of user desires, it specifies the student’s expectations from the web service, then goal interacts with the web service which describes the functionality of “course registration” web service. At this point the student’s goal and functionality of web service should meet with each other to go through the registration steps. Therefore in the scenario goal and web service satisfy each other. Web service takes *name*, *course*, *year* and *semester* coming from goal and checks if the information satisfies the information needed to register for the course. If the student has submitted all information required by “course registration” web service, web service takes action to complete the registration.

Step 2: In order to complete course registration successfully, web service takes the following processes; web service makes validation of student *name*, *course*, *semester* and *year* into ontology. Ontology like database has all attributes, attribute values and relations as well as logical constraints needed for “course registration” web service. For example, the following validations have to be done.

- If the requested course is opened in the submitted year and semester.
- If the requested course has prerequisite and student has taken the prerequisite course.
- Student should not have clashed courses after registration, so it tries to validate there is not a clash.

During all the processes, if any problem does not raised, web service creates instance and relation that shows the student who is taking the requested course. Finally, web service responds to students with student *name*, *course*, *year*, *semester*, *groupno* and increases the current size of course by one.

4.2. Course Registration Ontology

Course registration ontology is the main ontology. It consists of concepts, instances, relations, relation instances and axioms which are needed for the specification of “course registration” web service. Before starting to define concepts, at the beginning of the WSML document, definition of WSML language variant and namespaces with their prefix are required.

Figure 4.1 shows the Prologue of a WSML File. WSML variant is defined with the keyword wsmlVariant, while namespace is defined with the keyword namespace, also it can be seen that the WSML document variant has adopted WSML-rule and name space is named by courseRegistartion.

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

Figure 4.1: Prologue of a WSML file

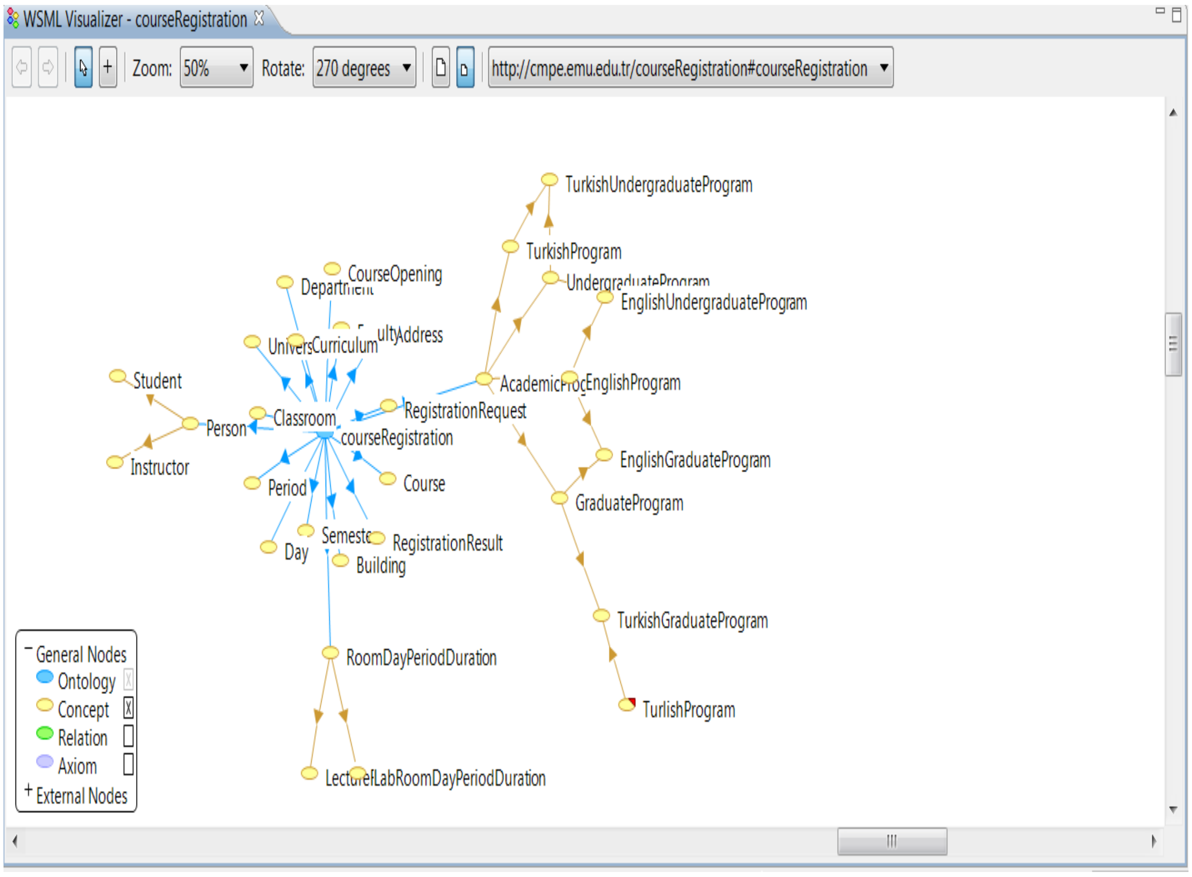
After the definition of WSML variant and namespace, we can import other ontologies as necessary. In the domain, there are five ontologies, ontology for “concepts”, ontology for “axioms”, ontology for “relations”, ontology for “instances” and ontology for “relation instances”. Although, we might define concepts, axioms, instances, relations and relations instances in one ontology, we preferred to separate them into different ontologies to reduce complexity of specification. Therefore, there are four ontologies imported into course registration ontology. Figure 4.2 below shows how to import, *courseRegistrationAxioms*, *courseRegistrationInstances*, *courseRegistrationRelations* and *courseRegistrationRelationInstances* ontologies into *courseRegistration* ontology.

```
ontology courseRegistration

importsOntology {courseRegistrationAxioms,
courseRegistrationRelations,courseRegistrationInstances,
courseRegistrationRelationInstances}
```

Figure 4.2: Importing ontologies

4.2.1. Concept Definitions of CourseRegistration Ontology

CourseRegistration ontology includes several super concepts and sub concepts together with necessary restrictions on attributes. In addition, *CourseRegistration* ontology contains some utility concepts, for example; *period*, *day* and *building*. Figure 4.3 depicts all super concepts, sub concepts and utility concepts of course registration ontology through the WSML visualizer. However, reading concept names might be very difficult because of the graph resolution. But from figure 4.4 you can see the whole super concepts of course registration ontology clearly. Also, later in the thesis, the definition of each concept of course registration ontology will be seen with its functionality.


WSML Visualizer - courseRegistration

Zoom: 50%

Rotate: 270 degrees

http://cmpe.emu.edu.tr/courseRegistration#courseRegistration

Figure 4.3: Course registration ontology concepts

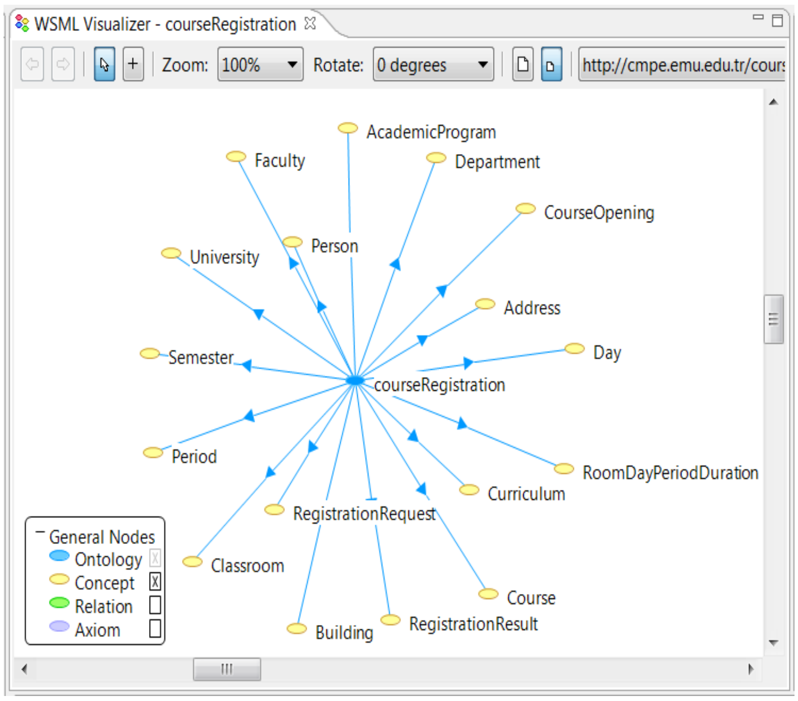


Figure 4.4: Super concepts of course registration ontology

```
concept University
     uname  ofType (1 1) _string
     locatedAt  ofType  (1 *) Address

concept Address
     street  ofType  (0 1) _string
     city  ofType  (1 1) _string
```

Figure 4.5: Definition of University and Address concept

Figure 4.5 shows the definition of *University* and *Address* concepts. *University* concept has two attributes *uname* and *locatedAt* which defines university name and address of university respectively. In addition, *Address* concept defines *street* and *city* attributes. As we can see, under the *University* concept *LocatedAt* attribute has type *Address*. This means that *LocatedAt* attribute is multivalued attribute which keeps all attribute values defined in *Address* concept such as *street* and *city*. In figure 4.22 you can see example instances for *University* and *Address* concept respectively.

```
concept Faculty
facultyName ofType  (1 1) _string
atUniversity ofType  (0 1) University

concept Department
deptID ofType  (1 1) _string
deptName ofType  (0 1) _string
inFaculty ofType  (0 1) Faculty
```

Figure 4.6: Definition of Faculty and Department Concept

Figure 4.6 depicts the definition of *Faculty* and *Department* concepts. *Faculty concept* has two attributes; *facultyName* (specifies the name of faculty) and *atUniversity* (specifies the university information). *Department* concept has *deptID*, *deptName* (specifies the name of department) and *inFaculty* (specifies the name of faculty) attributes. In addition, under *Department* concept, there is the *inFaculty* attribute. The value of *inFaculty* must be instance of *Faculty* concept. In figure 4.23 and 4.24 you can see example instances created for *faculty* and *department* concept respectively.

```
concept AcademicProgram
programName ofType  (0 1) _string
programID ofType  (0 1) _string
belongsTo ofType  (0 1) Department
concept UndergraduateProgram subConceptOf AcademicProgram
concept GraduateProgram subConceptOf AcademicProgram
concept TurkishProgram subConceptOf AcademicProgram
concept EnglishProgram subConceptOf AcademicProgram
concept EnglishGraduateProgram subConceptOf {EnglishProgram,
GraduateProgram}
concept EnglishUndergraduateProgram subConceptOf { EnglishProgram,
UndergraduateProgram}
concept TurkishGraduateProgram subConceptOf {TurkishProgram,
GraduateProgram}
concept TurkishUndergraduateProgram subConceptOf { TurkishProgram,
UndergraduateProgram}
```

Figure 4.7: Definition of AcademicProgram concept

Figure 4.7 shows the *AcademicProgram* concept which defines the academic program of departments. It includes the *programName*, *ProgramID* and *belongsTo* attributes. Its values are instance of the *Department* concept defined in figure 4.6. As shown in figure, *AcademicProgram* concept has four sub concepts; *UndergraduateProgram*, *GraduateProgram*, *TurkishProgram* and *EnglishProgram* which also have sub concepts named *EnglishGraduateProgram*, *EnglishUndergraduateProgram*, *TurkishGraduateProgram* and *TurkishUndergraduateProgram*. These sub concepts are defined as utility concepts to specify the type of academic program

```
concept Course
courseCode  ofType  (0 1) _string
courseName  ofType _string
hasPrerequisite  ofType (0 *) Course
lecture_hour  ofType  (0 1) _integer
tutorial_hour ofType  (0 1) _integer
credits  ofType  (0 1) _integer
ects  ofType  (0 1) _integer
belongsToProgram ofType  (0 1) UndergraduateProgram
```

Figure 4.8: Definition of Course concept

Figure 4.8 shows the definition of *Course* concept which has eight attributes; *courseCode* (specifies course code of course), *courseName* (specifies course name), *hasPrerequisite* (specifies the pre perquisite of course if exist, course can have zero or more perquisite course), *lecture_hour* (specifies lecture hours), *tutorial_hour* (specifies tutorial hour if exists), *credits* (specifies the credit of course), *ects*, *belongsToProgram* (specifies the program that course belongs). In addition *hasPrerequisite* and *belongsToProgram* attributes are multivalued attributes. *hasPrerequisite* attribute takes all attributes of *course* concept. *belongsToProgram* attribute takes all attributes value of *UndergraduateProgram* concept defined in figure 4.7.

```
concept Building
concept Classroom
classID  ofType  (0 1) _string
location  ofType  (0 1) Building
capacity  ofType (0 1) _integer
inDepartment  ofType  (0 1) Department
roomNumber  ofType  (0 1) _string
```

Figure 4.9: Definition of Building and Classroom concept

Figure 4.9 describes the *Classroom* concept which includes the following attributes; *classID*, *location* (specifies the building information of classroom, it has type as *Building* therefore *location* value must be instance of *building* concept), *capacity* (specifies capacity of classroom), *inDepartment* (specifies the department that classroom belongs, it takes all attributes value of *Department* instance), *roomNumber* (specifies room number of classroom). In figure 4.28, we can see the instances of *Classroom* concept.

```
concept Semester
      syear  ofType  (0 1) _integer
concept Day
concept Period
```

Figure 4.10: Defining Semester, Day, Period concept

```
concept RoomDayPeriodDuration
room  ofType  (1 1) Classroom
day  ofType  (1 1) Day
period  ofType  (1 1) Period
duration  ofType  (1 1) _integer
```

Figure 4.11: RoomDayPeriodDuration concept

In figure 4.10 the definition of utility concepts can be seen which are *Semester*, *Day*, *Period*. Also from figure 4.11 we can see the definition of *RoomDayPeriodDuration* concept which includes attributes, *room* (inherits all attributes value of *Classroom* instance), *day* (specifies the teaching day of course), *period* (specifies time period of course), *duration* (specify the teaching time duration of course) can be seen. These concepts will be used to define information of opening course in the following figures.

```
concept LectureRoomDayPeriodDuration subConceptOf
RoomDayPeriodDuration
```

Figure 4.12: Definition of LectureRoomDayPeriodDuration concept

```
concept LabRoomDayPeriodDuration subConceptOf RoomDayPeriodDuration
```

Figure 4.13: Definition of LabRoom DayPeriodDuration concept

From figure 4.12 and figure 4.13 you can see the *LectureRoomDayPeriodDuration* and *LabRoomDayPeriodDuration* concepts which are utility concepts that show us which courses can have lecture and lab session.

```
concept CourseOpening
groupNo  ofType  (0 1) _integer
ofCourse  ofType  (0 1) Course
semester  ofType  (0 1) Semester
id_courseOpening  ofType  (0 1) _string
teaching_times  ofType  (1 4) RoomDayPeriodDuration
current_size  ofType (0 1) _integer
year  ofType  (0 1) _integer
```

Figure 4.14: Definition of CourseOpening concept

The figure 4.14 shows the definition of *CourseOpening* concept which is one of the most important concepts in the domain. It includes the following attributes; *groupNo* (defines group number of course), *ofCourse* (takes attributes value of *Course* instance), *semester* (specifies the semester of opening course), *id_courseOpening*, *teaching_times* (specifies the information about teaching times, takes all attributes value of *RoomDayPeriodDuration* instance), *current_size* (specifies the number of students that enrolled in the opening course currently), *year* (specifies year of opening course).

```
concept Person
ID   ofType  (0 1) _string
gender  ofType  (0 1) _string
Date_of_Birth   ofType  (0 1) _date
name  ofType  (0 1) _string
lastName  ofType  (0 1) _string
address  ofType Address
```

Figure 4.15: Definition of Person concept

Figure 4.15 shows the definition of *Person* concept. *Person* concept has six attributes; *ID*, *gender*, *Date_of_Birth*, *name*, *lastName* and *address*. *Person* concept is a super concept of *Student* and *Instructor* concept as defined in figure 4.16.

```
concept Student subConceptOf Person
yearEnrolled  ofType  (0 1) _integer
overallCGPA  ofType  (0 1) _float
enrolledIn  ofType  (0 2) AcademicProgram
semesterEnrolled ofType  (0 1) Semester
tookCourse_  ofType (0 *) Course
nfp
dc#relation  hasValue {TookRelation}
endnfp
concept Instructor subConceptOf Person
works_in ofType  (1 *) Department
```

Figure 4.16: Definition of Student and Instructor concept

Figure 4.16 shows the definition of *Student* and *Instructor* concepts. *Student* concept is a subconcept of *Person* concept. It inherits all attributes of *Person* concept. Additionally it includes *yearEnrolled* (the year that student enrolled in university), *overallCGPA* (student cgpa), *enrolledIn* (specifies academic program in which student is enrolled, takes all attributes value of *AcademicProgram* instance), *semesterEnrolled* (specifies the current semester that student registered), *tookCourse* (specifies the course that student has taken, it accepts instances of *Course* concept as value and keeps all attributes value of *Course* instance). In addition, *tookCourse_* attribute has nonfunctional property which includes relation, named as *TookRelation* *TookRelation* is a logical expression that checks if course has perquisite or not.

Instructor concept is also sub concept of *Person* concept. It includes all attributes value of *Person* concept as well as including its own attributes such as *works_in* attribute which specifies the department that the instructor works in.

```
concept Curriculum
academicProgram ofType  (0 1) AcademicProgram
refCode ofType  (0 1) _string
courseName ofType  (0 1) Course
```

Figure 4.17: Definition of Curriculum concept

Figure 4.17 depicts the definition of *Curriculum* concept which includes the following attributes; *academicProgram* (data value must be instance of *AcademicProgram*), *refCode*, *courseName* (data value must be instance of *Course*).

```
concept RegistrationRequest
student ofType Student
course ofType Course
year ofType _integer
semester ofType Semester

concept RegistrationResult
student ofType Student
courseOpening ofType CourseOpening
```

Figure 4.18 Definition of RegistrationRequest and RegistrationResult concept

Figure 4.18 describes *RegistrationRequest* and *RegistrationResult* concepts. *RegistrationRequest* concept defines attributes like *student*, *course*, *year* and *semester*. Remember that when students make course registration request to "course registration" web service, the name of student, course, year and semester information must be submitted to the web service. The web service takes the information and makes validation in ontology. This validation is done through the *reqistartionRequest* concept. Therefore, *RegistrationRequest* concept is required to define which attributes have to be submitted by the student in order to make course registration request. In addition we have to define *RegistrationResult* concept which defines the information that will be sent to the student after registration is completed.

4.2.2. Relations of Course Registration Ontology

In WSML relations are machine readable and understandable and used to model dependencies between several concepts [58] therefore we can define many relations between many concepts.

In the "course registration" web service domain there are four relations defined as following; *teaches* specifies the relationship between teacher and opening course concepts. *takes* describes the relationships among the student and course concepts, *tookCourse* defines the relationships between the student and course concepts, and finally *prerequisite* specifies the interrelation of courses. In figure 4.19, all relations defined through WSML visualizer graph can be seen.

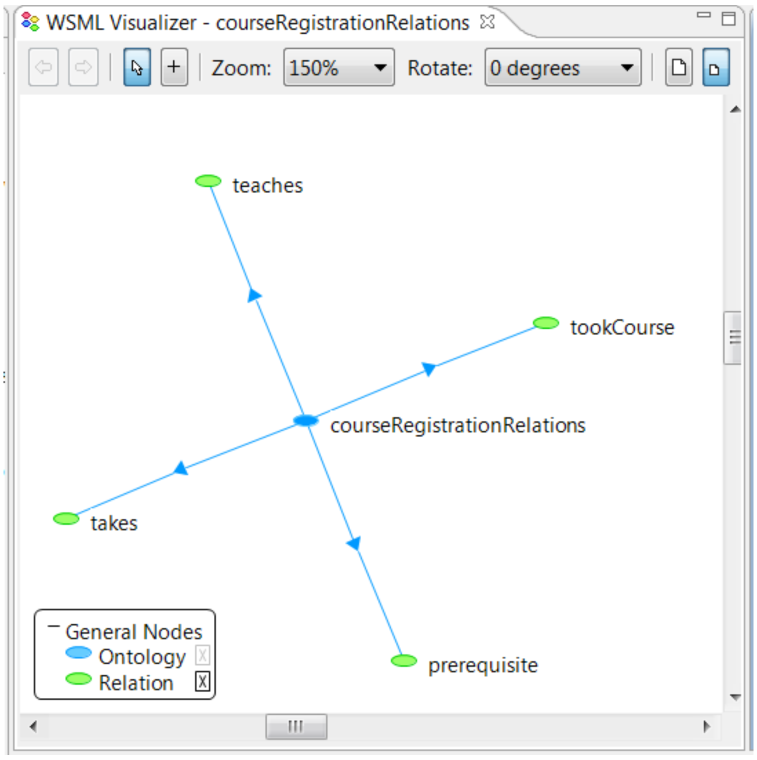


Figure 4.19: Specification of relations through WSML visualizer graph

```
relation teaches(  ofType Instructor,   ofType CourseOpening)
relation takes(  ofType Student,   ofType CourseOpening)
relation tookCourse( ofType Student,  ofType Course)
relation prerequisite( ofType Course, ofType Course)
```

Figure 4.20: Specification of relation for course registration

Figure 4.20 depicts the relations through WSML text editor. As can be seen *teaches* relation includes two parameters; *Instructor* and *CourseOpening*. It defines who is teaching which course. *takes* relation includes two parameters as well, *Student* and *CourseOpening*. It defines the relation between student and taken course by student currently. *tookCourse* relation allows two parameters, *Student* and *Course*. It defines which course has been taken by the student. *Prerequisite* relation has two parameters; *Course* and *Course*. It defines which course is prerequisite of the other course.

4.2.3. Instances of Course Registration Ontology

A concept represents real world objects with a specific shared property. The objects themselves are called instances [9]. In the domain, in order to exemplify WSML in detail, many instances and relation instances have been defined for each concept and relation. However , for some concepts we defined only one instance which is enough to demonstrate the specification of "course registration" web service. Figure 4.21 shows the definition of instances for each concept through the WSMO visualizer. However, difficulties may rise in order to read the name of instances because "course registartion" ontology has a very large number of instances and the resolution is very low. Therefore, you can see all instance definitions in detail through the WSML text editor in further reading of the thesis.

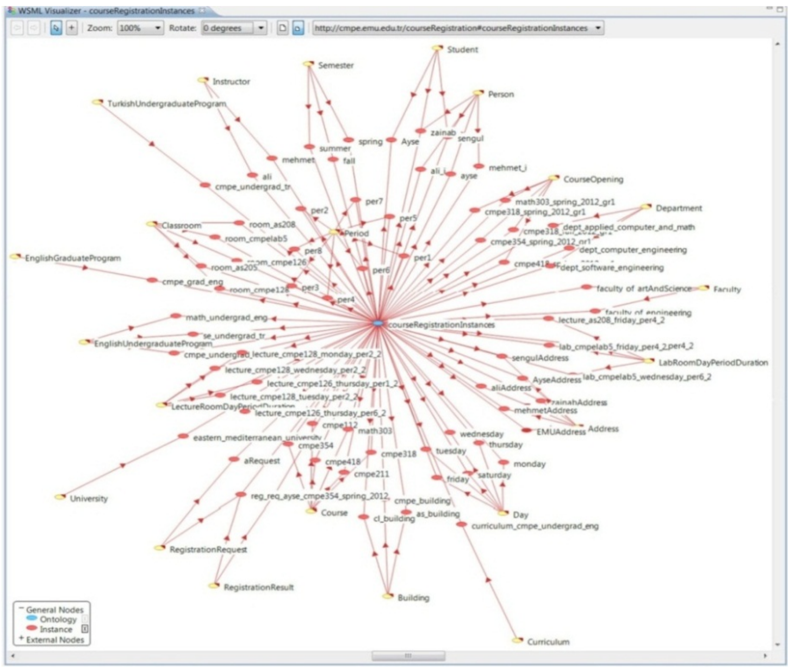


Figure 4.21: Instances of CourseRegistration Ontology Concepts

```
instance eastern_mediterranean_university memberOf University
uname hasValue "Eastern Mediterranean University"
locatedAt hasValue EMUAddress

instance EMUAddress memberOf Address
City  hasValue "Famagusta"
Street  hasValue "Salamis Road"

instance AyseAddress memberOf Address
street hasValue "Duplupınar"
city hasValue "Famagusta"

instance zainabAddress memberOf Address
street hasValue "Karakol"
city hasValue "Famagusta"

instance sengulAddress memberOf Address
street hasValue "Sakarya"
city hasValue "Famagusta"

instance aliAddress memberOf Address
street hasValue "Canakkale"
city hasValue "Famagusta"

instance mehmetAddress memberOf Address
street hasValue "Ortakoy"
city hasValue "Nicosia"
```

Figure 4.22: Definition of University and Address instances

Figure 4.22 shows *University* and *Address* instances. As can be seen, there is only one instance for *University* concept has been defined which is named as *eastern_mediterranean_university*. It contains *locatedAt* attribute with *EMUAddress* as value. *EMUAddress* is instance defined for *Address* concept which contains *city* and *street* attributes. When we are defining instance, we have to be sure that attribute types of instances must be compatible with the corresponding attribute type in the concept definition. For example, type of *locatedAt* attribute defined as *Address* in the *University* concept. So when we are creating instance for *University* concept, *locatedAt* attribute value must be instance of *Address* concept.

```
instance faculty_of_engineering memberOf Faculty
facultyName hasValue "Engineering"
atUniversity hasValue eastern_mediterranean_university

instance faculty_of_artAndScience memberOf Faculty
facultyName hasValue "Art and Science"
atUniversity hasValue eastern_mediterranean_university
```

Figure 4.23: Definition of Faculty instances

The figure 4.23 specifies the instances for *Faculty* concept. As can be seen, there are two *Faculty* instances defined which are faculty of "engineering" and faculty of "art and science". Later, *Faculty* instances will be used in order to describe *inFaculty* attribute of *Department* concept.

```
instance dept_applied_computer_and_math memberOf Department
deptName hasValue "Applied Computer and Mathemetics"
inFaculty hasValue faculty_of_artAndScience
deptID hasValue "applied computer and mathematics"

instance dept_computer_engineering memberOf Department
deptName hasValue "Computer Engineering"
inFaculty hasValue faculty_of_engineering
deptID hasValue "computer_engineering"

instance dept_software_engineering memberOf Department
deptName hasValue "Software Engineering"
inFaculty hasValue faculty_of_engineering
deptID hasValue "sotware_engineering"
```

Figure 4.24: Definition of Department instances

In figure 4.24, the example instances for *Department* concept can be seen. There are three different departments in the domain which are *dept_applied_computer_and_math*, *dept_computer_engineering* and *dept_software engineering*. In figure 4.6, the definition of *Department* concept can be seen. As seen, *inFaculty* attribute is a multivalued attribute, it takes *Faculty* of instance as value.

```
instance cmpe_undergrad_eng memberOf EnglishUndergraduateProgram
     programID  hasValue"cmpe_undergrad_eng"
     programName  hasValue"Computer Engineering Undergraduate 
English"
     belongsTo  hasValue dept_computer_engineering

instance math_undergrad_eng memberOf EnglishUndergraduateProgram
     programID  hasValue"math_undergrad_eng"
     programName  hasValue"Applied Computer and Mathemetic"
     belongsTo  hasValue dept_applied_computer_and_math

instance se_undergrad_tr memberOf EnglishUndergraduateProgram
     programID  hasValue"se_undergrad_eng"
     programName  hasValue"Software Engineering Undergraduate 
Turkish"
     belongsTo  hasValue dept_software_engineering

instance cmpe_undergrad_tr memberOf TurkishUndergraduateProgram
     programID  hasValue"cmpe_undergrad_tr"
     programName  hasValue"Computer Engineering Undergraduate 
Turkish"
     belongsTo  hasValue dept_computer_engineering

instance cmpe_grad_eng memberOf EnglishGraduateProgram
     programID  hasValue"cmpe_grad_eng"
     programName  hasValue"Computer Engineering Graduate English"
     belongsTo  hasValue dept_computer_engineering

instance turkish_teaching memberOf TurkishGraduateProgram
     programID  hasValue"turkish_grad_tr"
     programName  hasValue"Instructional Turkish Teaching Graduate 
English"
     belongsTo  hasValue dept_computer_engineering

instance computer_teaching memberOf TurkishGraduateProgram
     programID  hasValue"computer_grad_tr"
     programName  hasValue"Instructional Computer Teaching Graduate 
English"
     belongsTo  hasValue dept_computer_engineering
```

Figure 4.25: Definition of Academic Programs instances

Figure 4.25 defines the instances of *AcademicPrograms* concept. In the figure, four different academic program can be seen; *English Under graduate Program*, *Turkish Under graduate Program*, *English Graduate Program* and *Turkish Graduate*

Program. In addition, *belongsTo* attribute accept instance of *Department* as value and it includes all attributes value of *Department* concept.

```
instance cmpe354 memberOf Course
credits hasValue 3
belongsToProgram hasValue cmpe_undergrad_eng
courseName hasValue"Introduction to Databases"
courseCode hasValue"cmpe354"
hasPrerequisite hasValue cmpe211
lecture_hour hasValue  6
ects hasValue 4

instance cmpe318 memberOf Course
credits hasValue 3
belongsToProgram hasValue cmpe_undergrad_eng
courseName hasValue"Introduction to Programming Languages"
courseCode hasValue"cmpe318"
hasPrerequisite hasValue cmpe112
lecture_hour hasValue  6
ects hasValue 4

instance cmpe112 memberOf Course
credits hasValue 3
belongsToProgram hasValue cmpe_undergrad_eng
courseName hasValue"Programming Fundamentals"
courseCode hasValue"cmpe112"
lecture_hour hasValue  6
ects hasValue 4

instance cmpe211 memberOf Course
credits hasValue 3
belongsToProgram hasValue cmpe_undergrad_eng
courseName hasValue"Object Oriented Programming"
courseCode hasValue"cmpe211"
lecture_hour hasValue  6
ects hasValue 4

instance cmpe418 memberOf Course
courseName hasValue"Internet Programming"
belongsToProgram hasValue cmpe_undergrad_eng
tutorial_hour hasValue 1
credits hasValue 4
lecture_hour hasValue 4
courseCode hasValue"cmpe418"
ects hasValue 4
```

Figure 4.26: Definition of Course instances

Figure 4.26 depicts example instances defined for *Course* concept. As we can see, there are many course instances defined for *Course* concept which are quite enough to demonstrate “course registration” web service. For example, in figure, there is a *cmpe354* instance which has a course name “introduction to database”, that belongs to cmpe_undergrad_eng_program. course code is cmpe354, credits is 3 and has a pre requisite which is cmpe211. *hasPrerequisite* attribute is a multivalued attribute, it accepts instance of *Course* as value, in the example, cmpe112 is prerequisite of cmpe354 which is named with “Programming Fundamentals”, belongs to the cmpe_undergrand_eng_program, course code is cmpe112, credits is 3 and it does not have prerequisite course. Additionaly *belongsToProgram* is multivalued attribute, it accepts instance of *UndergraduateProgram* concepts as value.

```
instance cmpe_building memberOf Building
instance cl_building memberOf Building
instance as_building memberOf Building
```

Figure 4.27: Definition of Building instances

Figure 4.27 shows the instances about *Building* concept. In the following example, these instances will be used to define the location of classes.

Figure 4.28 shows the instances for *Classroom* concept. There are five classrooms defined such as; *cmpe128*, *cmpe126*, *room cmpelab5*, *room as208*, *room as205*. Let’s take *cmpe128* classroom as an example, it has *deptId* as ComputerEngineering, *roomNumber* as 128, *inDepartment* dept_computer_engineering, *ClassID* as “room_cmpe128”, *Capacity* as 60, and *location* as cmpe_building. *inDepartment* is a multivalued attribute refers to *Department* concept and *Department* instance. Likewise, *location* attribute refers to instance of *Building* department.

```
instance room_cmpe128 memberOf Classroom
deptID hasValue ComputerEngineering
roomNumber hasValue"128"
inDepartment hasValue dept_computer_engineering
classID hasValue"room_cmpe128"
capacity hasValue 60
location hasValue cmpe_building

instance room_cmpe126 memberOf Classroom
roomNumber hasValue"126"
inDepartment hasValue dept_computer_engineering
classID hasValue"cmpe126"
location hasValue cmpe_building
capacity hasValue 50

instance room_cmpelab5 memberOf Classroom
roomNumber hasValue"lab5"
inDepartment hasValue dept_computer_engineering
classID hasValue"cmpelab5"
location hasValue cmpe_building
capacity hasValue 50

instance room_as205 memberOf Classroom
deptID hasValue"ArtandScience"
roomNumber hasValue"205"
inDepartment hasValue dept_applied_computer_and_math
classID hasValue"room_as205"
capacity hasValue 70
location hasValue as_building

instance room_as208 memberOf Classroom
deptID hasValue"ArtandScience"
roomNumber hasValue"208"
inDepartment hasValue dept_applied_computer_and_math
classID hasValue"room_as208"
capacity hasValue 70
location hasValue as_building
```

Figure 4.28: Definition of Classroom instances

```
instance spring memberOf Semester
syear hasValue 2012
instance fall memberOf Semester
syear hasValue 2011
instance summer memberOf Semester
syear hasValue 2012
```

Figure 4.29: Definition of Semester instances

```
instance monday memberOf Day
instance tuesday memberOf Day
instance wednesday memberOf Day
instance thursday memberOf Day
instance friday memberOf Day
instance saturday memberOf Day
instance per1 memberOf Period
instance per2 memberOf Period
instance per3 memberOf Period
instance per4 memberOf Period
instance per5 memberOf Period
instance per6 memberOf Period
instance per7 memberOf Period
instance per8 memberOf Period
```

Figure 4.30: Definition of Day and Period instances

Figure 4.29 shows *Semester* instances and figure 4.30 shows the instances for *Day*, *Period* which will enable us to define lecture or lab section of the course in the following figure 4.31.

```
instance lecture_cmpe128_monday_per2_2 memberOf
LectureRoomDayPeriodDuration
room hasValue room_cmpe128
day hasValue monday
period hasValue per2
duration hasValue 2

instance lecture_cmpe128_tuesday_per2_2 memberOf
LectureRoomDayPeriodDuration
room hasValue room_cmpe128
day hasValue tuesday
```

Figure 4.31: Definition of RoomDayPeriodDuration instances

```
period hasValue per2
duration hasValue 2

instance lecture_cmpe128_wednesday_per2_2 memberOf
LectureRoomDayPeriodDuration
room hasValue room_cmpe128
day hasValue wednesday
period hasValue per2
duration hasValue 2

instance lecture_cmpe126_thursday_per1_2 memberOf
LectureRoomDayPeriodDuration
room hasValue room_cmpe126
day hasValue thursday
period hasValue per6
duration hasValue 2

instance lecture_cmpe126_thursday_per6_2 memberOf
LectureRoomDayPeriodDuration
room hasValue room_cmpe126
day hasValue thursday
period hasValue per6
duration hasValue 2

instance lab_cmpelab5_wednesday_per6_2 memberOf
LabRoomDayPeriodDuration
room hasValue room_cmpelab5
day hasValue friday
period hasValue per4
duration hasValue 2

instance lab_cmpelab5_friday_per4_2 memberOf
LabRoomDayPeriodDuration
room hasValue room_cmpelab5
day hasValue friday
period hasValue per4
duration hasValue 2

instance lecture_as205_friday_per4_2 memberOf
LabRoomDayPeriodDuration
room hasValue room_as205
day hasValue friday
period hasValue per4
duration hasValue 2

instance lecture_as208_friday_per4_2 memberOf
LabRoomDayPeriodDuration
room hasValue room_as208
day hasValue friday
period hasValue per4
duration hasValue 2
```

Figure 4.31: Definition of RoomDayPeriodDuration instances (Continued)

Figure 4.31 specifies the example instances for *RoomDayPeriodDuration* concept. There are four instances defined for *LectureRoomDayPeriodDuration* such as; *lecture_cmpe128_monday_per2_2*, *lecture_cmpe128_Tuesday_per2_2*, *lecture_cmpe128_Wednesday_per2_2*, *lecture_cmpe126_Thursday_per1_2* and four instances defined for *LectureRoomDayPeriodDuration* such as; *lab_cmpelab5_wednesday_per6_2*, *lab_cmpelab5_friday_per4_2*, *lecture_as205_friday_per4_2*, *lecture_as205_friday_per4_2*. As can be seen in the figure, *room*, *day*, *period* and *duration* attributes accepts instance of *room*, *day*, *period* and *duration* concept as values. In this figure we have created instances for lecture and lab teaching times. These instances will be used in the following figure to define teaching times of *CourseOpening* instances.

```
instance cmpe354_spring_2012_gr1 memberOf CourseOpening
id_courseOpening hasValue"cmpe354_spring_2012_gr1"
year hasValue 2012
semester hasValue spring
ofCourse hasValue cmpe354
groupNo hasValue 1
current_size hasValue 4
teaching_times hasValue {
                        lecture_cmpe128_monday_per2_2,
                         lecture_cmpe128_wednesday_per2_2,
                         lab_cmpelab5_friday_per4_2 }

instance cmpe318_spring_2012_gr1 memberOf CourseOpening
id_courseOpening hasValue"cmpe318_spring_2012_gr1"
year hasValue 2012
semester hasValue spring
ofCourse hasValue cmpe318
groupNo hasValue 1
current_size hasValue 4
teaching_times hasValue {
                         lecture_cmpe128_tuesday_per2_2,
                          lab_cmpelab5_wednesday_per6_2,
                          lecture_cmpe126_thursday_per6_2}
```

Figure 4.32: Definition of CourseOpening instances

```
instance cmpe318_fall_2012_gr1 memberOf CourseOpening
id_courseOpening hasValue"cmpe318_fall_2012_gr1"
year hasValue 2011
semester hasValue fall
ofCourse hasValue cmpe318
groupNo hasValue 1
current_size hasValue 4
teaching_times hasValue lecture_cmpe126_thursday_per1_2

instance math303_spring_2012_gr1 memberOf CourseOpening
id_courseOpening hasValue"math303_spring_2012_gr1"
year hasValue 2012
semester hasValue spring
ofCourse hasValue math303
groupNo hasValue 1
current_size hasValue 6
teaching_times hasValue lecture_as205_friday_per4_2

instance cmpe418_spring_2012_gr1 memberOf CourseOpening
id_courseOpening hasValue "cmpe418_spring_2012_gr1"
year hasValue 2012
semester hasValue spring
ofCourse hasValue cmpe418
groupNo hasValue 1
current_size hasValue 4
teaching_times hasValue lecture_as208_friday_per4_2
```

Figure 4.32: Definition of CourseOpening instances (Continued)

Figure 4.32 describes instances of *CourseOpening* concept. There are five Course instances defined in the domain, for example, course *cmpe354*, id is cmpe354_spring_2012, year is 2012 of spring semester, group no is 1, *teaching_times* are “monday at period 1”, “wednesday at period 2”, and “Friday at period 3”, at classroom cmpe128 and has 2 sessions. As we can see, *ofCourse* takes a course instance as value and includes all corresponding course information. *teaching_times* attribute is a multivalued attribute which refers to instance of *RoomDayPeriodDuration* concept.

.

```
instance ayse memberOf Person
ID   hasValue"044059"
gender   hasValue"female"
name   hasValue  "Ayse"
lastName   hasValue  "Akçam"
address   hasValue AyseAddress

instance zainab memberOf Person
ID   hasValue"080045"
gender   hasValue"female"
name   hasValue  "Zainab"
lastName   hasValue  "Murtadha"
address   hasValue  zainabAddress

instance sengul memberOf Person
ID   hasValue"105066"
gender   hasValue"female"
name   hasValue  "sengul"
lastName   hasValue  "Cobanoglu"
address   hasValue  sengulAddress

instance ali memberOf Person
ID   hasValue"255"
gender   hasValue"male"
name   hasValue  "Ali"
lastName   hasValue  "Deniz"
address   hasValue aliAddress

instance mehmet memberOf Person
ID   hasValue"105"
gender   hasValue"male"
name   hasValue  "Mehmet"
lastName   hasValue  "Can"
address   hasValue mehmetAddress
```

Figure 4.33: Definition of Person instances

Figure 4.33 shows instances of *Person*. Additionally *address* attribute refers to instance of *Address* concept

In figure 4.34, we can see instances of *Student* and *Instructor* concept. We have defined three student instances named *ayse*, *zainab* and *sengul*. For example Ayse has ID 104059, enrolled in spring semester, cgpa is 3.20, enrolled in cmpe undregraduate program (enrolledIn attribute takes the instance of *AcademicProgram* as value) and year is 2010.

Also we have defined two instructors, ali and mehmet, both of them working in computer engineering department (works_in attribute has type *Department*, which takes *Department* instances as value).

```
instance Ayse memberOf Student
ID  hasValue"044059"
semesterEnrolled hasValue spring
overallCGPA hasValue _float("3.20")
enrolledIn hasValue cmpe_undergrad_eng
yearEnrolled hasValue 2010

instance zainab memberOf Student
ID  hasValue"080045"
semesterEnrolled hasValue spring
overallCGPA hasValue _float("3.50")
enrolledIn hasValue cmpe_undergrad_eng
yearEnrolled hasValue 2008

instance sengul  memberOf Student
ID  hasValue"105066"
semesterEnrolled hasValue spring
overallCGPA hasValue _float("3.56")
enrolledIn hasValue cmpe_grad_eng
yearEnrolled hasValue 2009

instance ali memberOf Instructor
works_in hasValue dept_computer_engineering

instance mehmet memberOf Instructor
works_in hasValue dept_computer_engineering
```

Figure 4.34: Definition of Student and Instructor instances

At the figure 4.35, we can see *curriculum_cmpe_undergrad_eng* instance for *Curriculum* concept. In addition, *academicProgram* attribute has value *cmpe_undergrad_eng* which is instance of *EnglishUndergraduateProgram*.

```
instance curriculum_cmpe_undergrad_eng memberOf Curriculum
academicProgram hasValue cmpe_undergrad_eng
refCode hasValue "cmpecurriculum"
courseName hasValue cmpe318
```

Figure 4.35: Definition of Curriculumn instances

Figure 4.36 shows instances of *RegistrationRequest* and *RegistrationResult* concept. *student*, *course*, *semester* and *CourseOpening* attributes accept instance of *Student*, *Course*, *Semester* and *couseOpening* concept as value. *ReqistrationRequest* instance is created when student has made request for course registration. For example, when, Jane made request to "course registration" web service, *reg_req_jane_cmpe354_spring_2012* is created. After validation of attributes submitted by *Jane, reg_res_jane_cmpe354_spring_2012* instance becomes created.

```
instance reg_req_jane_cmpe354_spring_2012 memberOf
RegistrationRequest
student hasValue Jane
course hasValue cmpe354
year hasValue 2012
semester hasValue spring
courseOpening hasValue cmpe354_spring_2012_gr1

instance reg_res_jane_cmpe354_spring_2012 memberOf
RegistrationResult
student  hasValue ayse
courseOpening hasValue cmpe354_spring_2012_gr1
```

Figure 4.36: Definition of RegistrationRequest and RegistrationResult instances

4.2.4. Relation Instances of Course Registration Ontology

Relation instance shows facts about relation [58]. In figure 4.37 we can see relation instances with the corresponding relation definition through WSML visualizer graph. As can be seen there are two ontologies, *courseRegistrationRelations* and *courseRegistrationRelationInstances* which are connected with each other by *prerequisite*, *takes*, *tookCourse* and teaches relations. *prerequisite* relation has two instances which are identified by *cmpe211PrerequisiteOfcmpe354* and *cmpe112Prerequisiteofcmpe318*. *takes* has two instances identified by *sengulTakescmpe354* and *ayseTakescmpe354*. *tookCourse* relation has four instances identified by *sengulTookCoursecmpe354*, *zainabTookcoursecmpe211*, *sengulTookcoursecmpe211*, *ayseTookCoursecmpe211*. Finally, *teaches* relation has two instances, identified by *aliTeachescmpe354* and *mehmetTeachescmpe354*.

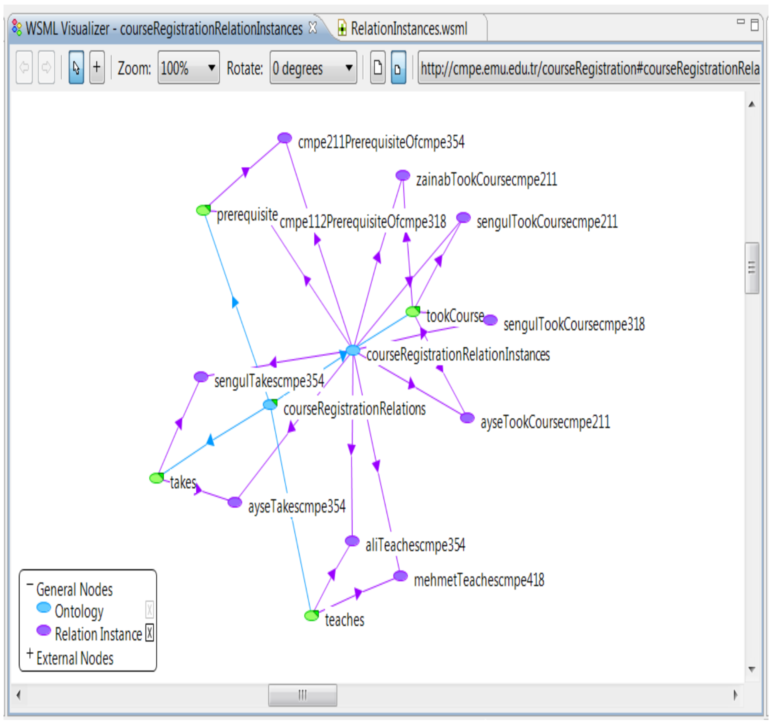

WSML Visualizer - courseRegistrationRelationInstances
RelationInstances.wsml
Zoom: 100%
Rotate: 0 degrees
http://cmpe.emu.edu.tr/courseRegistration#courseRegistrationRela

Figure 4.37: Relation Instances of Course Registration Ontology Concepts

```
relationInstance  aliTeachescmpe354
teaches(ali, cmpe354_spring_2012_gr1)

relationInstance  mehmetTeachescmpe318
teaches(mehmet, cmpe418_spring_2012_gr1)
```

Figure 4.38: Definition of "teaches" relation instances

Figure 4.38 shows the definition of *teaches* relation instance which specifies relation between instructor and opening course. As can be seen, it takes two parameters; student name and course opening. For example, *ali* teaches *cmpe354_spring_2012_gr1* and *mehmet* teaches *cmpe318_spring_2012_gr1*.

```
relationInstance ayseTakescmpe354
takes(ayse, cmpe354_spring_2012_gr1
relationInstance sengulTakescmpe354
takes(sengul, cmpe354_spring_2012_gr1)
```

Figure 4.39: Definition of “takes” relation instances

Figure 4.39 shows instance of *takes* relation, which defines the relation between student and opening course. It states that *sengul* and *ayse* take the opening course *cmpe354_spring_2012_gr1*.

```
relationInstance cmpe211PrerequisiteOfcmpe354
prerequisite(cmpe211,cmpe354)
relationInstance cmpe112prerequisiteOfcmpe318
prerequisite(cmpe112,cmpe318)
```

Figure 4.40: Definition of “prerequisite” relation instances

The above figure 4.40 depicts the instances of *prerequisite* relation which defines the relation between two courses. For example, it says that *cmpe211* is prerequisite of *cmpe354* and *cmpe112* prerequisite of *cmpe318*.

Figure 4.41 describes the relation instances of *tookCourse* between student and course. Remember, *ayse* is taking opening course *cmpe354_spring_2012_gr1* which has prerequisite course *cmpe211*, in order to let *ayse* take *cmpe354_spring_2012_gr1*, there must be *tookCourse* relation showing that *ayse* has taken *cmpe211*.

```
relationInstance ayseTookCoursecmpe112
tookCourse(ayse,cmpe112)

relationInstance ayseTookCoursecmpe211
tookCourse(ayse,cmpe211)

relationInstance sengulTookCoursecmpe318
tookCourse(sengul,cmpe318)

relationInstance zainabTookCoursecmpe211
tookCourse(zainab,cmpe211)

relationInstance sengulTookCoursecmpe211
tookCourse(sengul,cmpe211)
```

Figure 4.41: Definition of "tookCourse" relation

4.2.5. Axiom of Course Registration Ontology

Axioms are logical constraints which are directly related with the relations instances that are defined in figures 4.38, 4.39, 4.40 and 4.41. In the domain, we have defined eight axioms; *Clashes* checks if there is clash in teaching times of courses, *noClashStudent* checks if the student has a clash with other courses. *prerequisisteTaken* checks if the course has prerequisite and prerequisite of course has been taken, *prerequisiteNotTaken* checks if student has not taken the prerequisite of course, *teachCourse* specifies constraints about only which teachers can teach courses, *noClashTeacher* checks if the teaching times of teacher clashes, *noClashRoom* enables classroom to not clash, *classSizeExceeded* checks if the capacity of the class has exceeded the number of students that are registered to the course, *registrationRules* combines all rules needed for registration such as *prerequisiteNotTaken*, *classSizeExceeded*. In the figure below you can see all the definitions of axioms through WSML visualizer.

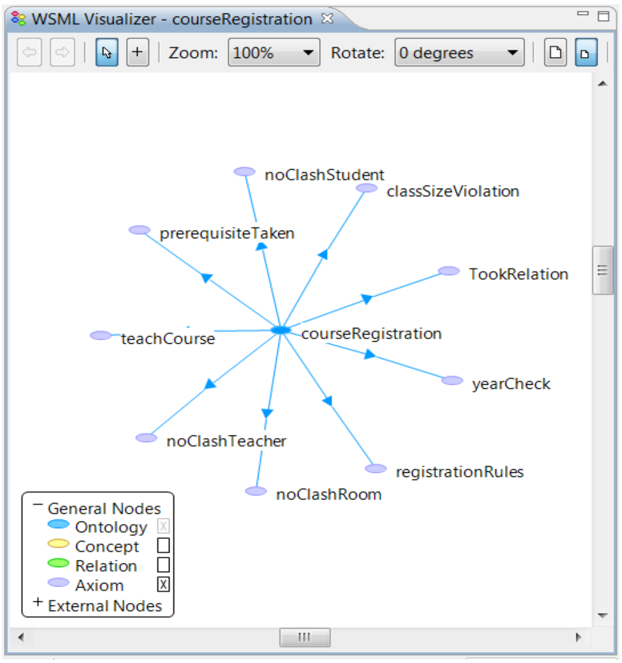


Figure 4.42: Axioms of CourseResgistartion Ontology

```
axiom registrationRules definedBy
clashes(?co1,?co2):- ?co1 memberOf CourseOpening and
?co2 memberOf CourseOpening and ?co1 != ?co2 and
?co1[ teaching_times  hasValue ?tt1,
      year hasValue ?y1,
      semester  hasValue ?s1] and

?co2[ teaching_times  hasValue ?tt1,
      year hasValue ?y1,
      semester  hasValue? s1].
```

Figure 4.43: The “Clashes” axiom

Figure 4.43 shows the *Clashes* axiom. According to *Clashes* rule, teaching times of courses must not be clashed, in the given example in figure axiom depicts that there are two courses ?co1 and ?co2, the *teaching_times*, *year*, *semester* of ?co1 should not be same as *teaching_times*, *year* and *semester* of ?co2. For example, in figure 4.32, I have defined some instances for *CourseOpening* , if we look at “cmpe354 spring 2012 gr1” and “cmpe318 spring 2012 gr1” instances, we can see that “cmpe354 spring 2012 gr1” is open 2012 spring semester and teaching times are “lecture cmpe128 monday per2 2”, “lecture cmpe128 wednesday per2 2”, “lab cmpelab5 friday per4 2”. In addition, “cmpe318 spring 2012 gr1” is open 2012 spring semester and teaching times are “lecture cmpe128 tuesday per2 2”, “lab cmpelab5 wednesday per6 2”, “lecture cmpe126 thursday per6 2”. There is not any clash in this case, but if change “lecture cmpe128 monday per2 2” to “lecture cmpe128 tuesday per2 2”, the clash will be automatically recognized and we should get a noClashRoom error message.

```
prerequisiteNotTaken(?student,?course,?precourse):-
takes(?student, ?courseOpening) and
?courseOpening[ofCourse  hasValue ?course] memberOf CourseOpening
               and prerequisite(?pre,?course) and
               naf tookCourse(?student,?precourse).
```

Figure 4.44: The “prerequisiteNotTaken” axiom

Figure 4.44 shows the *prerequisiteNotTaken* axiom. According to *prerequisiteNotTaken* axiom if students did not take prerequisite of course, he cannot take the course, *prerequisiteNotTaken* axiom is related with “*takes*” relation. The expression of this axiom as following, it takes tree parameters, ?student, ?course and ?precourse, in order to make this expression satisfy able, there should be student taking course and course must be instance of *CourseOpening*. If *takes* relation is satisfied and course is a member of *CourseOpening*, then it checks if the course has prerequisite, if there is prerequisite instance exist for course, then it checks if there is not any relationship shows that student took the prerequisite course, it shows prerequisite consistency violation. For example, remember ayse is taking course “cmpe354 spring 2012 gr1” denoted by takes relation, also you can see from *prerequisite* relation instance, cmpe211 prerequisite course of cmpe354, and *tookCourse* relation says ayse has been taken cmpe211. In this example there is not any prerequisite consistency violation. But if you remove the *tookCourse* relation instance which is saying that ayse has taken cmpe211, there will be prerequisite consistency violation.

```
classSizeExceeded:-
?co[teaching_times  hasValue ?tt,
     current_size  hasValue ?s] memberOf CourseOpening and

?tt[room  hasValue ?room] memberOf RoomDayPeriodDuration and
                   ?room[capacity  hasValue?maxCap] and ?s>?maxCap.
```

Figure 4.45: The “classSizeExceeded” Axiom

Figure 4.45 describes the *classSizeExceeded* axiom which defines constraints between classroom capacity and the total number of students taking the course; quota of the course must not exceed the class capacity. For example in figure 4.32 we have defined some instances for *CourseOpening*, let’s take “cmpe354 spring 2012 gr1” as an example. Our aim is to compare current size of the course with the class capacity. We can see that current size of “cmpe354 spring 2012 gr1” is 4. Now we have to find out the class capacity of “cmpe354 spring 2012 gr1”. In order to find out the class capacity of this course, we have to use *teaching_times* instances. There are four teaching time instances as following, “lecture cmpe128 monday per2 2”, “lecture cmpe128 wednesday per2 2”, “lab cmpelab5 friday per4 2”, after we reach the teaching times, we will go to each instance and check the *room* for obtaining classroom name, after we found room name we will go to instance of classroom. For example, classroom of “lecture cmpe128 monday per2 2” is cmpe128, then we go to cmpe128 classroom instance and compare the *capacity* with current size of course, if the capacity is less than current size, the system gives error. This example is satisfied but if we change current size of any course to more than the class capacity, classSizeViolation error message will occur.

```
axiom noClashRoom definedBy
!-clashes(?x,?y).
```

Figure 4.46: Definition of “noClashRoom” axiom

```
axiom prerequisiteTaken definedBy
!- prerequisiteNotTaken(?student,?course,?pre).
```

Figure 4.47: Definition of “prerequisiteTaken” axiom

```
axiom classSizeViolation
definedBy
!- classSizeExceeded.
```

Figure 4.48: Definition of “ClassSizeViolation” axiom

Figure 4.46, figure 4.47 and figure 4.48 show the *noClashRoom*, *prerequisiteTaken*, and *classSizeViolation* axioms which are used to invoke *clashes*, *prerequisiteNotTake* and *classSizeExceeded* axioms with respectively in the negation form, because we do not want these axioms to be satisfied by any circumstances.

```
axiom yearCheck definedBy
!- ?co[ year  hasValue ?ye,
        semester  hasValue ?se] memberOf CourseOpening and

   ?se[syear  hasValue ?maxyear] memberOf Semester and

    ?ye != ?maxyear.
```

Figure 4.49: Definition of “yearCheck” axiom

Figure 4.49 shows the *yearCheck* axiom which defines constraint about year of opening course such as opening year of course must be same as semester year. In order to make year check, first, we go to instance of *CourseOpening* concept, we see that it has attribute values *year* and *semester*, we check the *semester* attribute value then go to specified semester instance and find out *year* that is defined for *Semester* instance. Finally, we can compare if the specified year of semester is equal to year of *courseOpening*. For example "cmpe354 spring 2012 gr1" opened 2012 in spring semester. Firstly, we go to *Semester* instance which is spring and check the attribute value of semester year. For instance, spring semester year is 2012. If we change the semester year to 2011, yearCheckedViolation error message will occur because year of course opening and year of semester must be the same.

```
axiom TookRelation
definedBy
?x[tookCourse_  hasValue ?course] memberOf Student:-
tookCourse(?x,?course).
```

Figure 4.50: Definition of "TookRelation" axiom

```
axiom noClashTeacher definedBy
!- ?t1 memberOf Instructor and
?co1 memberOf CourseOpeningand
?co2 memberOf CourseOpening and
teaches(?t,?co1)and teaches(?t,?co2)and
?co1 != ?co2 and
?co1[ teaching_times hasValue? tt1,
      year hasValue ?y1,
      semester   hasValue ?s1] and
?co2[ teaching_times hasValue ?tt2,
      year hasValue ?y1,
      semester hasValue ?s1] and
?tt1[ day hasValue ?d1,
       period hasValue ?p1]and
?tt2[ day hasValue ?d1,
      period hasValue ?p1].
```

Figure 4.51: Definition of "noClashTeacher" axiom

Figure 4.50 and figure 4.51 shows the *TookRelation* and *noClashTeacher* axioms respectively. *TookRelation* axiom checks if the student has been taken prerequisite course or not and *noClashTeacher* axiom defines constraint about teacher, teacher can teach many courses but the *year*, *semester* ,*day* and *period* of taught courses should not clash. For example, mehmet teaches course "cmpe318 spring 2012 gr1", teaching times are "cmpe128 monday per2 2", "lecture cmpe128 wednesday per2 2", "lab cmpelab5 friday per4 2". If mehmet teaches "cmpe418 spring 2012 gr1" which has teaching times as "lecture as208 friday per4 2". As we can see "friday period 2" clashes and noClashTeacher error message will occur.

```
axiom noClashStudent definedBy
!- ?t1 memberOf Student and
?co1 memberOf CourseOpening and
?co2 memberOf CourseOpening and
takes(?t1,?co1)and
takes(?t1,?co2)and
?co1[ teaching_times hasValue?tt1,
      year hasValue?y1,
      semester hasValue?s1] and
?co2[ teaching_times hasValue?tt2,
      year hasValue?y1,
      semester hasValue?s1] and
?tt1[ day hasValue?d1,
      period hasValue?p1]and

?tt2[ day hasValue?d1,
      period hasValue?p1]and
?co1 != ?co2.
```

Figure 4.52: Definition of "noClashStudents" axiom

Figure 4.52 depicts *noClashStudent* axiom. *noClashStudent* axiom defines constraint between courses that the student takes. The student cannot take the course if the day and period of course clashes with his other day and period of course. There is a student ?t1 and two courses ?c1 and ?c2 which are member of *courseOpening* concept, ?t1 takes both ?c1 and ?c2. the *teaching_times*, *year* and *semester* of ?c1 must not be equal to the *teaching_times*, *year* and *semester* of ?c2

4.2.6. Web Service specification for Course Registration

We use WSML-Rule to describe the functionality provided by "course registration" semantic web service. We can say it is a service which the user interacts in order to invoke "course registration" semantic web service with the aim of registering for the course.

Although we can create many web services within domain, we have created only one web service which is named as web service *courseRegistration***.** Figure 4.53 shows the definition of web service through the WSML visualizer graph.

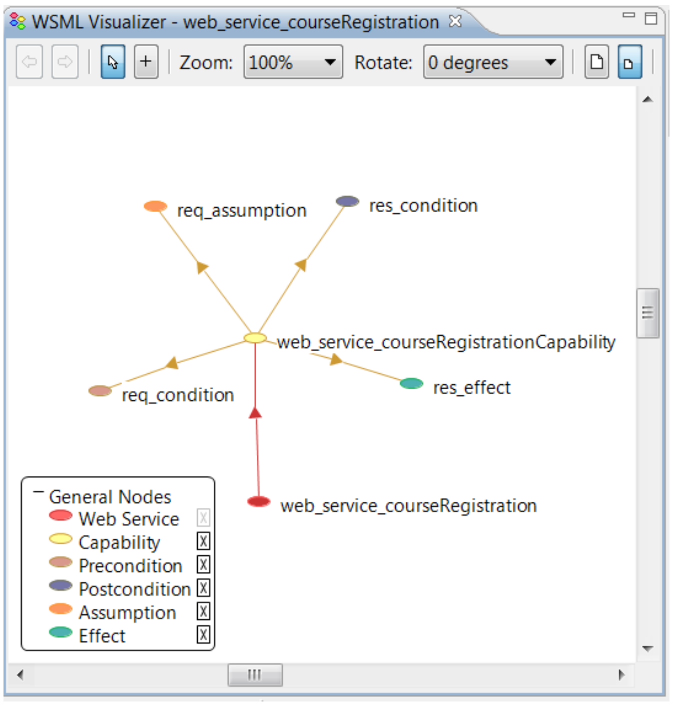


Figure 4.53: Definition of Web service through WSML visualizer

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

webService web_service_courseRegistration

importsOntology {courseRegistration, courseRegistrationAxioms,
courseRegistrationRelations,courseRegistrationInstances,
courseRegistrationRelationInstances}
```

Figure 4.54: Prelude of web service

Figure 4.54 shows the prelude of web service. In order to define functionality of web service, firstly, at the beginning of the WSML file we have to define WSML variant which is specified as "WSML-Rule" in the university course registration model. After that at the second line we have to declare namespaces and continue definition by importing ontologies, actually we have defined only one ontology in our domain and it is divided parts according to concepts, instances, relation instances and axioms in order to simplify ontology so that you might see as there are five ontologies imported. As well as importing ontologies we might define some nonfunctional properties under capability declaration which provide description about web service.

The core element of Web service definition is capability. Each web service must include one capability which specifies the provided features of web service to the user. In order to define web service capability in WSMO, the definition of the *preconditions*, *post conditions*, *effect* and *assumption* are required. In addition, capabilities also might include shared variables. From figure 4.58 to figure 4.63 we can see definition of capability.

```
capability web_service_courseRegistrationCapability
nonFunctionalProperties
discovery#discoveryStrategy  hasValue discovery#HeavyweightDiscovery
endNonFunctionalProperties
```

Figure 4.55: Definition of nonfunctional properties of web service.

```

sharedVariables {?student, ?course,?year, ?semester, ?oldsize, ?co}
```

Figure 4.56: Definition of shared variables of web service.

Figure 4.55 and figure 4.56 shows the definition of nonfunctional properties and shared variables respectively. While nonfunctional properties define description about web services, shared variables part defines the shared variables among assumptions, pre-conditions, post-conditions and effects. In the given example, there are six shared variables. Shared variables link various parts of the capability.

```
Precondition
definedBy
?rr[
student hasValue ?student,
course hasValue ?course,
year hasValue ?year,
semester hasValue ?semester] memberOf RegistrationRequest.
```

Figure 4.57: Precondition definition for the web service capability

Figure 4.57 shows the precondition definition of web service. *Pre-Condition* defines the condition of the web service in order to service users. In the given example, the precondition says that an instance of the *RegistrationRequest* concept is required for successful provision of "course registration" web service. There is no other information that can be accepted by web service.

```
Assumption
definedBy
?co[
    ofCourse hasValue ?course,
    year hasValue ?year,
    semester hasValue ?semester,
    groupNo hasValue ?groupno,
    current_size hasValue ?oldsize] memberOf CourseOpening.
```

Figure 4.58: Assumption definition for the web service capability

Figure 4.58 depicts the Assumption of web service. *Assumption* describes the expectation before executing the web service. If the expectation is not meet, successful execution of web service is not guaranteed. Within the given example, assumption is saying that before the web service can be called, there must be a course opening object for the course requested.

Figure 4.59 defines the Effect of Web service. *Effect* describes what the student will reach after the execution of the Web service successfully. In the given example, effect is saying that if student satisfies the precondition and assumption and execute the web service, it is guaranteed that the student will be registered to the course and the number of students registered to the course which is specified by ?current_size will be increased one.

```
Effect
definedBy
takes(?student,?co) and
?co[current_size  hasValue (?oldsize+1)].
```

Figure 4.59: Effect definition for the web service capability

Figure 4.60 shows the Post condition of web service. *Post Conditions* describe the state of world that is reached after the execution of the Web service successfully; in other words, post condition describes the relation between the information that is provided to the Web service, and its results. In the given example, post condition is specifying which information will be provided after invoking web service successfully, as you can see student and *CourseOpening* will be an instance of the Registration Result and provided to the student.

```
postconditiondefinedBy
?aResult[
student hasValue ?student,
courseOpening hasValue ?co] memberOf RegistrationResult.
```

Figure 4.60: Post condition defined for capability

4.2.7. Goal of Course Registration

Goals are similar to as web services, but they are used to describe expectation of users when invoking the web service. Figure 4.61 depicts capability, precondition and post condition through WSML visualizer. In the following examples we present goals for user in order to register for a course. More precisely, through the goal, the student attempts to invoke an appropriate web service for doing the registration.

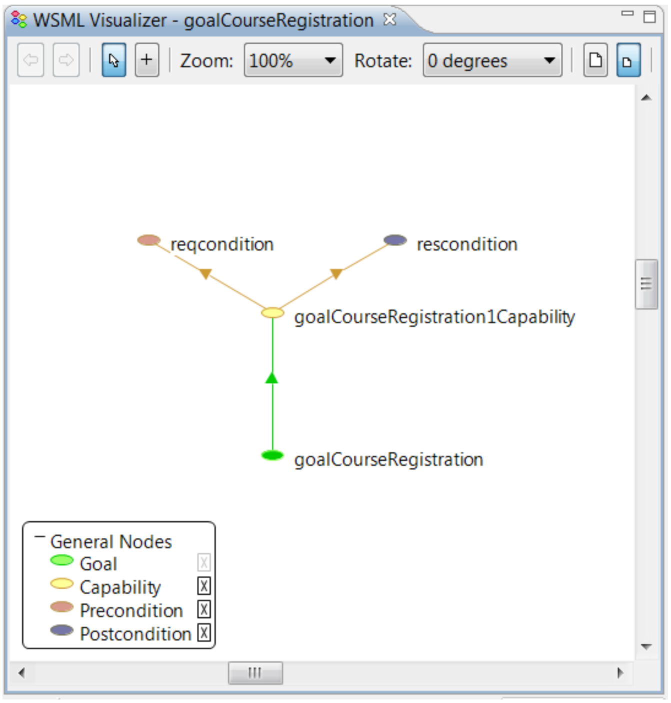


Figure 4.61: Instances of CourseResgistartion Ontology Concepts

In figure 4.62 you can see goal definition of web service through WSML editor. At the beginning of the specification of Goal, we have specified WSML variant, namespaces, and imported ontologies as we specified in web services. Then we have to define capability elements such as precondition and post condition.

Precondition has logical formalism what student provides before executing web service in order to register for a course. In the given example, student requests service with attribute values of *student*, *course*, *year* and *semester*. *Postcondition* defines expectation of user after executing web service. In the given example user desires that will be instance of *registrationResult* concept.

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


goal goalCourseRegistration

importsOntology
{ _"http://cmpe.emu.edu.tr/courseRegistration#courseRegistration",

_"http://cmpe.emu.edu.tr/courseRegistration#courseRegistrationInstan
ces",

_"http://cmpe.emu.edu.tr/courseRegistration#courseRegistrationRelati
onInstances"}

capability goalCourseRegistrationAliCapability
nonFunctionalProperties
discovery#discoveryStrategy  hasValue
{discovery#HeavyWeightRuleDiscovery, discovery#NoPreFilter}
endNonFunctionalProperties

sharedVariables {?student, ?co, ?groupno}

precondition reqcondition
definedBy

?rr[
      student hasValue?student,
      course hasValue ?course,
      year hasValue ?year,
      semester hasValue ?semester] memberOf RegistrationRequest.

Postcondition rescondition
definedBy

?aResult[
      student hasValue ?student,
      courseOpening hasValue ?co] memberOf RegistrationResult
and
      ?co [
      ofCourse hasValue ?course,
      year hasValue ?year,
      semester hasValue ?semester,
      groupNo hasValue ?groupno ] memberOf CourseOpening.
```

Figure 4.62: Specification of goal for course registration

```
_"http://cmpe.emu.edu.tr/courseRegistration#courseRegistration",
_"http://cmpe.emu.edu.tr/courseRegistration#courseRegistrationInstan
ces",
_"http://cmpe.emu.edu.tr/courseRegistration#courseRegistrationRelati
onInstances"}

capability goalCourseRegistration1Capability

nonFunctionalProperties
          discovery#discoveryStrategy  hasValue
{discovery#HeavyWeightRuleDiscovery, discovery#NoPreFilter,
discovery#HeavyweightDiscovery}
endNonFunctionalProperties

sharedVariables {?student, ?co, ?groupno}

precondition reqcondition
definedBy

?rr[
      year  hasValue 2012,
      student  hasValue Jane,
      semester  hasValue spring,
      course hasValue cmpe354] memberOf RegistrationRequest.

postcondition rescondition
definedBy

?aResult[
      student  hasValue Jane,
      courseOpening  hasValue ?co] memberOf RegistrationResultand

?co[
      year  hasValue 2012,
      semester  hasValue spring,
      groupNo  hasValue ?groupno,
      ofCourse hasValue cmpe354] memberOf CourseOpening.
```

Figure 4.63: Requesting course registration for Jane

Figure 4.63 shows example goal which makes course registration request for Jane. Jane submits *name*, *year*, *student*, *semester* and *course*, and web service returns to Jane with student *name*, *course name*, *year*, *semester* and *group no*.

4.2.8. Mediator of Course Registration Ontology

Mediators enable to connect Ontologies, Goals and Web Services that are defined in different platforms without interoperability problems between them. WSMO studio provides extendible editor that enables us to create Mediators as well as the Ontologies, Goal and Web service. In the domain, I have not defined any mediator because requesters and providers would use the same domain ontologies for the description of their goals and Web services, respectively. However, we have to define mediators in the real world because in an open environment the same ontologies and web services might not be used.

Chapter 5

EVALUATION OF WSMO AND WSML

This Chapter discusses some of the important deficiencies discovered in WSML as we critically analyze and evaluate WSML language while creating university course registration specification. In addition, we have also recommended possible improvements which help WSML to achieve covering many aspects of semantic web service. We have discovered eleven WSML issues with some recommendations.

5.1. General Overview

WSMO provides a framework by in which WSML is used to semantically describe all relevant aspects of Web services (ontologies, mediators, goals and web services) in order to automatically discover, combine and invoke web services over the Web [10]. In general, the idea of WSMO is really exciting and in the future it may be a very reliable framework in modeling ontologies, creating web services, interacting with web services automatically and exchanging data between them. However, currently formal language of WSMO is not issue-free and there are some deficiencies that WSML has to overcome in order to allow WSMO to achieve its goals. For example, complex syntax definition of WSML, unspecified WSML variants is one of the most important issues to be resolved in WSML.

5.2. Implementation Issue

WSML supports some tools to develop WSML specification in an easy way such as WSMT which is used in this thesis to investigate WSML language. WSMT provides graphical user interface framework and tools (WSML visualizer) to specify description of web services semantically through WSML [66]. Although WSMT provides user-friendly tools to describe ontologies, axioms, mediators, goals and web services semantically, it does not provide an effective reasoner. Currently, WSML must be supported with an external reasoner in order to implement WSML files. Deficiency of WSML reasoner leads us to an unreliable manner of developing semantic web application.

Another issue that we faced with reasoning is testing the consistency of the ontology. For example, if we want to test the whole ontology to see if there is any logical or semantic problem in any part of the WSML file, we do not have any specific query to perform over model. I have to write some irrelevant query expressions in order to see if everything is well in ontology. In this thesis, to test the whole ontology, we used 3<5 as a query which is unrelated with the ontology and just gives results as true or false. For an alternative solution we can add debug button to the WSML reasoner which will test the whole ontology if there is any syntactic mistake.

Furthermore, when we want to test a goal, the WSML reasoner does not enable testing of goal implementation; it does not even provide any related error on it, this means that there is no reasoning mechanism for goals. Reasoning goals must be provided by the WSML reasoner.

5.3. Error Provider Issue

WSML does not have proper error provider mechanism when they arise. For instance, when any constraints are violated, a small window appears in an annoying way and says “Reasoning has encountered a problem. It was not possible to execute the given query”. However, it does not show or communicate a specific error and does not guide us to the error line. We would suggest that it should provide the line number or when we click over an error it links us to a place where the problem occurred. Therefore, we can add new functionality for error provider. For example, on the bottom side of window, errors can be shown to the user with red color and with line number, when the user clicks on the line number it may guide them to the point where the error occurred.

5.4. Variable Issue (Syntax)

WSML reasoner does not allow us to declare variables using underscore character. For example, ?var1_1 is not allowed as variable, it should be defined as ?var1. When I want to write huge expression having many variables, we have to use meaningful variable names in order to remember functionality of variable, at this point ” _” is very useful. Such as “?student_id, department_id, ?eng_undergrad”. However, WSML does not support variable that contains underscore. Although defining numbers as variable is allowed, using “_” character is not allowed in WSML.

5.5. Weak Error Detection Mechanism

Even though WSML can catch the WSML syntactic mistakes, WSML has a very weak detection mechanism for catching structural mistakes, for example when we created instance for the wrong concept, it did not catch that instance that shows it has a mismatch problem with concept. Let’s recall a previous example, we have University concept in figure 4.3 which is structured with two attributes, university name and address only. When we attempt to create instance for university and add another attribute which does not have definition in concept, such as foundation year, WSML would not catch the mistake. Furthermore, also consider that University concept address has attribute value as *Address* which is multivalued , at the instance definition if we set attribute value as “TRNC” (string), it does not match the mistake, however, attribute value should be instance of *Address* concept not string. For the structural errors, WSML does not provide any error catching mechanism.

5.6. Data Type Issue

WSML forces us to define attribute type when we are defining attribute for concept. There may be such cases that we do not know about the type of attribute. Therefore, there should be data type wrappers. Such as for the *University* concept, when we leave *Address* type definition empty in concept, define in instance definition as “TRNC”, during the implementing of ontology, WSML reasoner will arise error about the undefined attribute value, however there may be some cases that we could not decide about data type such as *Address* might be string or multivalued data type, same as foundation year can be both integer and string. Which one will be used as data type ? Therefore WSML should provide data type wrappers when we do not define the type of attribute. According to attribute value defined in instance, WSML should wrap the data type and should not arise error.

5.7. Attribute Value Definition of Instance

In WSML, defining super Concept and sub Concept does not have any meaning. WSML assumes like there is an imaginary concept and every concept is sub concept of it. Therefore, defining sub concept for some certain classes is meaningless. For example, as shown in the example below, *Instructor* concept is sub concept of *Person*; it includes all inherited attributes of *Person* concept as well as its own specific attributes. However, when we define instance for instructor such as ali, it must contain attribute definitions of *Person* and *Instructor* only. There should be restriction. In the given example below we have defined nationality value for instance of ali, although *nationality* attribute has never been defined in both *Person* and *Instructor* concept, why WSML allows us to set value for attribute, that was never defined. In my opinion, this means that there is an imaginary concept and every concept is a sub concept of it. That makes it useless for super-Concept and sub-Concept concepts. Even this makes it meaningless for attribute definition of concept, because whenever we want, we can directly define any attribute value in instance for certain concepts. This is useless of defining attribute type in concept. Since it is useless, why should we define super concept, sub concept and their attributes. We can shortly and straight forward create instances.

```
concept Person
     ID  ofType  (0 1) _string
     gender ofType  (0 1) _string
     Date_of_Birth  ofType  (0 1) _date
     name ofType  (0 1) _string
     lastName ofType  (0 1) _string
     address ofType Address

concept Instructor subConceptOf Person
    works_in ofType  (1 *) Department

instance ali memberOf Instructor
     works_in hasValue dept_computer_engineering
     name hasValue"ali"
     lastName hasValue"Can"
     gender hasValue"male"
     nationality hasValue"turkish"
     address hasValue Address
```

5.8. Matching between Relation and Relation instances Issue

In WSML in order to create relation instance, first of all we create definition of Relation. However, definition of relations is meaningless since WSML allows creating any relation instance. For example, *teaches* relation defined as follow relation teaches(**ofType** Student, **ofType** CourseOpening) and relation instance defines **relationInstance** teaches(ali, cmpe354_spring_2012_gr1). If we remove the definition of relation, WSML does not give any error. Checking whether relation is defined or not, is an important issue because we have to constrain the type of relation parameters. Therefore, in axioms when we are using relations we have to define the type of relations again as shown in the following example

```
?t1 memberOf Student
And ?co1 memberOf CourseOpening
And ?co2 memberOf CourseOpening
and takes(?t1,?co1)
and takes(?t1,?co2)
```

Although we defined the type of relation parameters in the definition of relation, we have to define again in axioms because there is no matching between definition of relation and relation instances. If WSML would match the data type of relation parameters, we would not need to define them again in axiom and this would provide simplicity and greater usability.

5.9. Aggregate Function Issue

None of the rule language of WSML allows us the use of the aggregate functions which are very useful in adding and averaging data, finding the largest and smallest values, and counting the records about specific criteria in the domain. For example; it is not possible to define logic rules (axiom) or constraints to restrict the number of students that can take a certain course. For this case, we need a counter to count how many students have been registered in the course and use it to prevent registering more students for the course. In addition, finding which class has the largest capacity is not possible. Therefore, Building predicates in WSML for the aggregate functions can be useful and efficient.

5.10. Missing Semantics for WSML-Full

From the language development point of view, the semantic of WSML-Full which is extension of WSML DL and WSML rule, does not have a complete semantics yet.

5.11. Capability Issue

Capability defines the functionality of a web service. WSML web services and goals may only have one capability. However, web service might have more functionalities, linking many request to web service means web service can have many functionalities.

In addition to that and in order to invoke the web services, preconditions under capability of goal must be matched with preconditions of web service exactly. If the user does not satisfy the conditions of the web service, it does not serve. In the following example there are preconditions of goal and web service

Precondition of Goal

```
?rr[year  hasValue ?year,
        student hasValue ?student,
        semester hasValue ?semester,
        course hasValue ?course] memberOf RegistrationRequest.
```

Precondition of web service

```
?rr[student  hasValue ?student,
course hasValue ?course,
year hasValue ?year,
semester hasValue ?semester] memberOf RegistrationRequest.
```

What will be happen if the user tries to invoke web service with the following pre condition?

Precondition of Goal

```
?rr[student  hasValue ?student,
course hasValue ?course] memberOf RegistrationRequest.
```

The service would not respond to the user’s desire because of missing attributes. Therefore, definition of more than one capability in web service is really required in order to make web service more accessible.

5.12. Choreography Issue

Choreography provides the necessary information in order to establish communication with the Web service from the user’s point of view [10]. For communications between web services and users, existence of choreography must be defined. Specification of Choreography in WSML is too high level and abstract to be of practical use.

5.13. Orchestration Issue

Orchestration is a sequence of rules and conditions, in which web service should follow to invoke other web services in order to perform some functionality. However, how web services discover and interact with each other, how to deal with web services that provides similar functionality, how to compose web services are questionable.

In this thesis, web service does not interact with other web services to provide functionality, because of that we do not need to define orchestration. If web service would use other web services, the sequence and activities between web service’s requester and providers must be defined. However, this is not possible because definition of the dynamic behavior of web services in the context of WSML is currently under investigation and has not been integrated in WSML at this point.

Chapter 6

CONCLUSION and FUTURE WORK

In this thesis, we have deeply studied Web service modeling language WSML and all elements relevant to the Web service modeling ontology (WSMO) framework through the specification of university course registration web service in WSML rule. Through our study we have identified some issues, namely the reasoning issue, the error provider issue, the definition of variable issue, the weak detection mechanism issue, the definition of data type issue, the definition of attribute value issue, the aggregate function issue, the capability issue and finally the choreography and orchestration issues. We proposed solutions to some of the discovered issues that we believe will improve WSML.

For future research, we wish to work on improving the weak points of WSML that we obtained in this thesis. We believe that WSMO is a promising framework for specifying semantic web services, and with the solution to the issues introduced in this thesis, it will be a viable approach also.

REFERENCES

[1] The World Wide Web (2001, January 24). Retrived January 23, 2013 from http://www.w3.org/WWW/.

[2] Cao, S. T., Honglei, Z., & Mcilraith, S. (2001, April-March). Semantic Web services, *Intelligent Systems, IEEE,* 16(2) , pp. 46-53..

[3] Accessing Health Information Through the Internet (2005). Retrieved January 18, 2012 from http://www.prb.org/pdf04/AccessHealthInfoInternet.pdf.

[4] Kanellopous, D., Kotsiantis S. (2007, September). Semantic Web: A state of the Art Survey, *International Review on Computers and Software,* 2(5), 428-442.

[5] Amin, A., Joa, A., Shayeganfar, F. & Wagner R., (2005, August 26-26). Semantic Web Challenges and new Requirements.1160-1163.

[6] Hendler, J., Lassila, O., & Berners Lee, T., (2011, May). The Semantic Web, *Scientific American.* Retrieved June 29, 2012 from https://courses.ischool.berkeley.edu/i202/f12/sites/default/files/SemanticWeb.pdf.

[7] Jeckle M., Zhang L., (2003). *Web Services - ICWS-Europe 2003*, Berlin: Springer Heidelberg, 183-197.

[8] Bussler, C., Dieter, F. (2002).The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications,* 1(2), 113-137.

[9] Dieter, F., Jos, D. B., Kifer, M., Krummenacher, R., Lausen, H., Polleres, A., Predoiu, L. & Uwe, K. (2005, June 3). Web Service Modeling Language (WSML). Retrieved November 1, 2012 from http://www.w3.org/Submission/WSML/.

[10] Web Service Modeling Ontology (WSMO) (2005, June 3). Retrieved November 20, 2012 from http://www.w3.org/Submission/WSMO/.

[11] Web Service Execution Environment (WSMX) (2005, June 3). Retrived Sepetember 18, 2012 from http://www.w3.org/Submission/WSMX/.

[12] Kerrigan, M. (2006, july). WSMOViz: An Ontology Visualization Approach for WSMO, *Intelligent Systems, IEEE,411-418.*

[13] HTML tutorial (2012). Retrieved January 20, 2013 from http://www.w3schools.com/html/default.asp.

[14] Anonymous author (2008). European Organization for Nuclear Research. Retrieved January 21, 2013 from http://public.web.cern.ch/public/en/about/name-en.html.

[15] Berners Lee T. (2009, October 16). Retrived September 26, 2012 from http://tr.wikipedia.org/wiki/Tim_Berners-Lee.

[16] Matthews B., Wilson W., (2006).The Semantic Web:Prospects and Challenges, *CCLRC Rutherford Appleton Laboratory,* 1(4), 26-29.

[17] Web 1.0, 2.0, and 3.0 (2009, May 7). Retrived 25 January,2013 from http://benramsey.com/blog/2009/05/web-10-20-and-30-defined/.

[18] Agarwal, P. R. (2009, May 9). *Semantic Web in Comparison to Web 2.0: Intelligent Systems, Modelling and Simulation (ISMS), 2012 Third International Conference on Intelligent Systems Modelling and Simulation.* 558-563.

[19] Austin, D., Barbi,r A., Ferris, C., & Garg, S. (2001, October 11). Web Services Architecture Requirements. Retrieved January 19, 2013 from http://www.w3.org/TR/2002/WD-wsa-reqs-20021011.

[20] Mathew S., McGovern J., Stevens M., & Tyagi S. (2003, May 12). *Java Web Services Architecture*. (2nd ed.). San Francisco: Morgan Kaufmann publishers.

[21] Daconta, M. C., Obrst, L. J., Smith, K. T. (2003, May 19). The Semantic Web.(2nd ed.). Indianapolis, Indiana: Wiley Publishing.

[22] Curbera, F., Duftler, M., Khalaf, R., Nagy, W, Mukhi, N. & Weerawarana, S. (2002, March). Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *Internet Computing, IEEE,* 6, 86-93.

[23] Box, D., Ehnebuske D., Kakivaya G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte S. & Winer D. (2000, May 8). Simple Object Access Protocol (SOAP) 1.1. Retrived 24 December 2012 from http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[24] Eric, N. (2001). Understanding Web Services: XML, Wsdl, Soap, and UDDI. (3rd ed.). pp 81-110.

[25] HTTP - Hypertext Transfer Protocol (2012 ,April 11). Retrieved December 15, 2012 from: http://www.w3.org/Protocols/.

[26] Christensen, E., Curbera, F., Meredith, G. & Weerawarana, S. (2001,March 15). Web Service Description Language (WSDL). Retrieved July 22, 2012 from http://www.w3.org/TR/wsdl.

[27] Jakhar, J., Parashar, A.(2001, August). A test Based Web Service Selection Approach. *International Journal of Research and Reviews in Computer Science*. 2(4), 1014-1017.

[28] Rouse, M. (2005, September). UDDI (Universal Description, Discovery, and Integration). Retrived July 22, 2012 from http://searchsoa.techtarget.com/definition/UDDI.

[29] Bai, J., Hao, Y. & Miao, G. (2011, November). Integrating Building Automation Systems based on Web Services. *Journal Of Software*. 6(11), 2209-2215.

[30] Berners-Lee, T., Hall, W., Hendler, J. A., O'Hara, K., Shadbolt, N. & Weitzner, D. J. (2006). A Framework for Web Science. *Foundations and Trends in Web Science*, 1(1), 1-30.

[31] Hendler, J., Skall, M., Martin, D., Marcatte, V., McGuinness, D. L., Pollock, J., Roure, D. D., & Yoshida H. (2004, November). OWL Web Ontology Language for Services (OWL-S). Retrived July 22, 2012 from http://www.w3.org/Submission/2004/07/.

[32] Miller, J., Sheth, A., Sivashanmugam, K. & Verma, K.(2001). Adding Semantics to Web Services Standards. Retrieved July 24, 2012 from http://knoesis.wright.edu/library/download/SemanticWSOld.pdf.

[33] Grigoris, A., Frank, V. H. (2004*). A Semantic Web Primer*. (2nd ed.). England: The MIT Press. 213.

[34] Battle, S., Bernstein, A., Boley, H., Groso, B., Gruninger, M., Hull, R., Kifer, M., Martin, D., Mcilraith, S., McGuinness, D., Su, J. & Tabet, S. (2005, September 9). Semantic Web Services Framework (SWSF) Overview. Retrieved July 22,2012 from http://www.w3.org/Submission/SWSF/.

[35] Abraham, B., Benjamin, G., Deborah, M., Harold, B., Jianwen, S., Michael, G., Michael, K., Richard, H., Said, T., Sheila, M., & Steve, B. (2005, September

9). Semantic Web Services Language (SWSL). Retrieved July 25, 2012 from http://www.w3.org/Submission/2005/SUBM-SWSF-SWSL-20050909/.

[36] Abraham, B., Benjamin, G., Deborah, M., David, M., Steve, B., Harold, B., Jianwen, S., Michael, G., Michael, K., Richard, M., Said, T. & Sheila, M. (2005, September 9). Semantic Web Services Ontology (SWSO). Retrieved July 25, 2012 from http://www.w3.org/Submission/2005/SUBM-SWSF-SWSO-20050909/.

[37] Hendler, J., Horrocks, I., Parsia, B. & Patel, S. P. (2005). *Principles and Practice of Semantic Web Reasoning*. Semantic Web Architecture: Stack or Two Towers?. Germany: Springer Berlin Heidelberg, 3703, 37-41.

[38] Renato, I. (2010, September 8). Semantic Web Architectures. *White Paper*. Semantic Identity. 32. Retrieved January 26, 2013 from http://semanticidentity.com/Resources/Entries/2010/9/8_Semantic_Web_Architectures_(Whitepaper).html

[39] Anonymous author, (2013, February 2). Uniform resource identifier. Retrieved February 4, 2013 from http://en.wikipedia.org/wiki/Uniform_resource_identifier.

[40] GEIST Research Group (2011). Semantic Web Unicode and URI. AGH University of Science and Technology, POLAND. Retrieved September 18, 2012 from http://home.agh.edu.pl/~wta/semweb/geist-semweb-uri.pdf.

[41] Bray, T., Paoli, J., Maler, E. & Yergeau, F. (2008, November 26). Extensible Markup Language (XML) 1.0 (Fifth Edition). Retrieved July 23, 2012 from http://www.w3.org/TR/xml/.

[42] Rdf Working Group. (2004, February 10). Resource Description Framework (RDF). Retrieved August 3, 2012 from http://www.w3.org/RDF/.

[43] Prud'hommeaux, E., Seaborne, A.(2008,January 28). SPARQL Query Language for RDF. Retrieved July 12, 2013 from http://www.w3.org/TR/rdf-sparql-query/.

[44] Pérez, J. and Arenas, M. & Gutierrez, C. (2006). *The Semantic Web - ISWC 2006.* Semantics and Complexity of SPARQL. Springer Berlin Heidelberg. 30-43.

[45] Bizer, C., Breslin, J., Manjunath, G., Packard, H. & Seaborne, A. (2008, July 15). SPARQL Update. Retrieved July 23, 2012 from http://www.w3.org/Submission/SPARQL-Update/.

[46] Kifer, M., Boley, H. (2012, December 12). RIF Owerview. Retrieved January 21, 2013 from http://www.w3.org/TR/2012/NOTE-rif-overview-20121211/.

[47] Miller, E., Koivunen, M. R. (2001, November 2).W3C Semantic Web Activity. Retrieved Sptember 23, 2012 from http://www.w3.org/2001/12/semweb-fin/w3csw.

[48] Alesso, P., Smith, C. F. (2005, April). *Developing Semantic Web Services*, 178.

[49] RDF Schema (RDFS) (2012). Retrieved 18 August, 2012 from http://www.w3schools.com/rdf/rdf_schema.asp.

[50] Haytham, T. F., Koutb, M. & Suoror, H. (2008). Semantic Web on Scope: A New Architectural Model for the Semantic Web, *Journal of Computer Science,* vol. 4, pp. 623-624.

[51] XML DOM Tutorial (2013). Retrieved December 2, 2012 from http://www.w3schools.com/dom/default.asp

[52] Harmelen, F., Hendler, J. Horrocks, I., Lassila, O. & McGuinness, D.L. (2000, November-December). The semantic Web and its languages. *Intelligent Systems and their Applications, IEEE*, 15(6), 67-73.

[53] Resource Description Framework (2013, January 25). Retrived January 28, 2013 from http://en.wikipedia.org/wiki/Resource_Description_Framework.

[54] Connolly, D., Harmelen, F., Horrocks, I., McGuinness, D. L., Patel-Schneider P. F. & Stein L. A. (2001, December 18). DAML+OIL (March 2001) Reference Description. Retrieved December 18, 2012 from http://www.w3.org/TR/daml+oil-reference.

[55] Deborah, L., Harmelen, F. (2004, February 10).OWL Web Ontology Language. Retrieved July 15, 2012 from http://www.w3.org/TR/owl-features/.

[56] Obitko, M.(2007). Web Ontology Language OWL. Retrieved January 26, 2013 from http://www.obitko.com/tutorials/ontologies-semantic-web/web-ontology-language-owl.html.

[57] Bruijn, J. D., Bussler, C., Domingue, J., Fensel, D., Hepp, M., Keller, U., Kifer, M., Knig-Ries, B., Kopecky, J., Rubn, L., Lausen, H., Oren, E., Polleres, A., Roman, D., Scicluna, J. & Stollberg, M. (2005, June 3). Web Service Modeling Ontology (WSMO). Retrieved June 27,2012 from http://www.w3.org/Submission/WSMO/.

[58] Bruijn, J., Fensel D., Lausen, H. & Polleres, A. (2006). The Web Service Modeling Language WSML: An Overview, in The Semantic Web: Research and Applications, Sure, York, Domingue and John, Eds., Springer Berlin Heidelberg, pp. 590-604.

[59] Busslerb, C. F., Bruijna, J. D., Feiera, C., Kellera, U., Laraa, R., Lausena, H., Polleresa, A., Romana, D. & Stollberga, M. (2005, January 1). Web Service Modeling Ontology, Applied Ontology, pp. 77-106.

[60] Gruber, T. R. (1993, June). A translation approach to portable ontology specifications," *Knowl. Acquis.,* 5(2), pp. 199-220.

[61] Domingue, J., Roman, D., & Stollberg, M. (2005, June 9-10). Web Service Modeling Ontology (WSMO) - An Ontology for Semantic Web Services. Retrieved 8 July, 2012 from http://www.w3.org/2005/04/FSWS/Submissions/1/wsmo_position_paper.html.

[62] Axel, P., Bruijn, J. D., Fensel, D. & Lausen, H. (2005, April). WSML - a Language Framework for Semantic Web Services. Retrieved August 22, 2012 from http://www.w3.org/2004/12/rules-ws/paper/44/

[63] Angele, J., Axel, P., Boley, H., Bruijn, J. D., Fensel, D., Hitzler, P., Kifer, M., Krummenacher, R., Lausen, H., & Rudi, S. (2005). Web Rule Language (WRL). Retrieved 7 August, 2012 from http://www.wsmo.org/wsml/wrl/wrl.html.

[64] Steinmetz, N., Toma, I. (2008, August 8). D16.1v1.0 WSML Language Reference. Retrieved October 16, 2012 from http://www.wsmo.org/TR/d16/d16.1/v1.0/.

[65] Moran, M., Vitvar, T. & Zaremba, M. (2009). Instance-based Service Discovery with WSMO/WSML and WSMX, *Semantic Web Services Challenge*, (8 ed.), Springer US, pp. 168-183. Retrieved October 12, 2012 from http://dx.doi.org/10.1007/978-0-387-72496-6_10.

[66] Axel, P., Bruijn, J. D., Fensel, D., Lausen, H., Keller, U., Kifer, M., Krummenacher, R., Polleres, A. & Predoiu, L. (2005, June 3). Web Service Modeling Language (WSML). Retrived June 26, 2012 from http://www.w3.org/Submission/WSML/#cha:related-efforts.