

# **Re-engineering Service Registry for Fast Service Search and Composition**

*A thesis submitted in 2015 to University of Hyderabad in partial  
fulfillment of the award of a Ph.D. degree*

*by*

**LAKSHMI.H.N**



**SCHOOL OF COMPUTERS & INFORMATION SCIENCES  
UNIVERSITY OF HYDERABAD  
(P.O.) CENTRAL UNIVERSITY  
HYDERABAD - 500 046, INDIA**

**JUNE 2015**



## **CERTIFICATE**

This is to certify that the thesis entitled “**Re-engineering Service Registry for Fast Service Search and Composition**” submitted by **Lakshmi.H.N** bearing Reg. No. 10MCPC09 in partial fulfillment of the requirements for the award of **Doctor of Philosophy** is a bonafide work carried out by her under my supervision and guidance.

The thesis has not been submitted previously in part or in full to this or any other University or Institution for the award of any degree or diploma . It is also free from any plagiarism.

**Prof Arun K.Pujari**  
Dean  
School of Computer and  
Information Sciences  
University of Hyderabad

**Prof Hrushiksha Mohanty**  
Supervisor  
School of Computer and  
Information Sciences  
University of Hyderabad

## DECLARATION

I, **Lakshmi.H.N**, hereby declare that this thesis entitled “**Re-engineering Service Registry for Fast Service Search and Composition**” submitted by me under the guidance and supervision of **Prof. Hrushikesh Mohanty** is a bonafide research work and is free from any plagiarism. I also declare that it has not been submitted previously in part or in full to this University or any other University or Institution for the award of any degree or diploma. I hereby agree that my thesis can be submitted in Shodganga/INFLIBNET.

**A report on plagiarism statistics from the University Librarian is enclosed.**

Date:

Name: **Lakshmi.H.N**

Regd. No. **10MCPC09**

//Countersigned//

Signature of the supervisor:

## DEDICATION

*I dedicate this thesis to my beloved soulmate*

***Vasuki.H.N***

*you have been my prime inspiration for pursuing research work, but for your confidence in me I wouldn't have dared to start this. You have been my guru, taught me what self-respect, persistence, hard-work and of-course, love, are all about. You have always been my motivation, through your eyes I have seen myself as a capable, intelligent woman who could do anything once I made up my mind. You always have instilled confidence in me to achieve my goals. I can never forget your humour, friendly sarcasm and witty dialogues that not only kept me laughing, but also lightened my perspective. There are no words that can express my gratitude and appreciation for all you've done and been for me. I am forever indebted to you Vasu, for making me independent and strong enough to face all facets of life. Without you, I may not have gotten to where I am today, at least not sanely. I surmise that you will be there with me in every walk of life, come what may.*

## ACKNOWLEDGEMENTS

First and foremost, I wish to thank my supervisor **Prof. Hrushikesh Mohanty**, who has been a tremendous mentor for me. It has been my privilege to work closely with you sir, a lifetime opportunity wherein you have helped me grow not only as a researcher, but also as a better person. This research would have not been possible but for your thought provoking discussions, timely suggestions and constant supervision. Thanks sir, for being there with me in all my tough times and grooming me to face tougher ones. I would also like to thank you for making this journey memorable, be it the de-stressing tea breaks, interesting stories and poems, or deciphering your facebook comments, I have enjoyed those all and am going to miss it a lot.

I am grateful to my doctoral review committee members **Dr. Alok Singh** and **Dr.T. Sobha Rani** for sagaciously analyzing my research work and giving me apt suggestions. Their valuable comments have helped in improvising my contributions.

I would like to thank dean, **Prof. Arun.K.Pujari** for providing facilities and infrastructure to carry out this research.

I would like to thank **Prof. Madhusudan Reddy**, advisor of CVR college of engineering and, **Dr. C.V Raghava**, chairman of CVR college of engineering for extending their support to complete my research, by providing weekly offs and study leaves without which completing this research work would have been nearly impossible. I am extremely grateful for their constant encouragement and moral support they extended at times most needed. I would also like to thank **Prof. L.C.Siva Reddy**, vice-principal, CVR college of engineering for motivating me to complete my research work and providing me the right advice at times when I needed them the most.

I greatly appreciate my little angels **Neeharika** and **Niveditha** who have been eagerly waiting for this day, for the maturity they have displayed all along. They have been my inspiration and driving force throughout and thanks to them for patiently tolerating my never ending (as it seemed to them) work schedule, with a hope to find me free someday soon. Thank you kids for setting a goal for me, for giving me a reason to live and smile.

Words can't describe how grateful I am for my parents, **Mr.H.K Nagaraja** and **Mrs.B.N. Krupalini**, for their eternal support. Thanks *amma*, for standing by me when I needed you the most, for not letting my kids miss me, for being their mother when they needed me, for those constant reminders of my responsibilities and deadlines to be met. Thanks *appa*, for having faith in me in my tough times and for those relentless efforts to revive my confidence. Completion of this research would have not been possible without you both. I wouldn't have come this far without the efforts and blessings of my grand parents, **Late.B.Narayan Rao** and **Mrs.B.N.Padma**. I thank them heartily for educating me and teaching me values. I take this opportunity to thank my aunt **Dr.B.N.Manjula**, who has been a source of great moral support throughout.

I am extremely thankful to my mother-in-law **Mrs.H.N.Nagalakshmi** for her perpetual encouragement and support, which has helped me focus on my goal. Your prayers for me was what sustained me thus far. A special thanks to my uncle **Mr.H.C Anantharamiah** for motivating me to complete my research, for those judicious advices at the time of crisis. Thanks for being with me throughout this journey, which would have been tougher without your support.

Another staunch supporter is my the best friend **A. Vani Vathsala**. Her love, support, and belief in me is a treasure. Through the precious memories we have shared together she has made a place in my heart only she will ever occupy. We have laughed, cried and among other things, cursed together. Being my senior in research, Vani had first hand knowledge of the research process and what I was experiencing, an understanding priceless for both of us. Vani has played many roles - of a friend, elder sister, colleague, confidant, humorist, phone pal, etc., etc. Thanks Vani, for being persistent

and encouraging, for believing in me, for being tolerant of my mood swings, for reinforcing my confidence, for reminding me of my abilities and for the many precious memories along the way.

My close friends **Bhagya Sree** and **B.Rajitha** have been motivating me all along, not letting me fail. I thank them for constantly reminding me my schedules, most of which I would forget. Their concern have helped me to sustain all the stress throughout and keep me focused. Thanks for your encouragement, support and most of all your humor that kept things light and me smiling.

My acknowledgment will be incomplete without the mention of my lab seniors **V.Supriya, A. B. Sagar, Deepak**, who have taught me the lab culture and for always being there and bearing with me the good and bad times during my wonderful days of Ph.D. My heartfelt thanks to my fellow lab mate, **Abhaya Pradhan** for his timely help and support through out.

Last but not the least, I heartily thank my *sarathy*, **Mr.T.Nagaraju**, who has driven me to success. Thanks for being with me in ups and downs of life during the last one and half years. You have been a silent spectator and at times a victim of my mood swings. Thanks for tolerating it all patiently, for changing my perception towards life.

June 2015

**Lakshmi.H.N**

# ABSTRACT

Web services are self-contained, self-describing, modular applications that can be published, searched, and invoked across the Web. Research on web service search and composition has become increasingly important in recent years due to the growing number of web services over Internet and the challenge of automating the process of composition. Since there is a large growth in number of available web services and possible interactions among them are huge, searching for desired set of services to satisfy a user query becomes very difficult. Thus service search and composition problems are research problems of importance.

Various approaches can be used for service search, such as, searching in UDDI, Web and service portals. We take up the issue of input/output parameter based search at service registries and improved upon the existing techniques by making use of meta information made available by pre-processing of services in registry. The repository of services with meta information make ESR: *extended service registry*, which we propose in this thesis.

In order to improve the performance of parameter based search in our *extended service registry*, we propose to pre-process our registry in two-levels. In the first level, to widen the scope of input/output parameter matching, we cluster service I/O parameters into semantic groups, based on their term equivalence. This is done by making use of semantic similarity and co-occurrence of terms in parameter names, computed using WordNet as the underlying ontology. These clusters are further utilized in the second level to cluster services based on their output parameter pattern similarity. Our clustering approach makes use of the co-occurrence of output parameters to cluster web



services on frequent output parameter pattern.

When a user queries for a service with a required output parameter pattern, initially, a service cluster that best matches the queried pattern is searched for. One may obtain many services in the matched cluster, from which best matching services need to be selected. Hence, as a next step, we propose a bi-level service selection approach that models both the functional and non-functional requirements of users. In first level services matching the functional requirements are shortlisted, which are further filtered in second level based on given QoS requirements, thus providing a list of web services that best matches a given user query.

For a given user query, when service selection fails to find a matching service, we propose to find possible compositions that satisfy a given user query. Utilizing the three types of matches: exact, super and partial, we widen the scope of composition by defining possibly three types of service composition: *exact*, *collaborative* and *super* composition. Service composition process results to a *composition search tree* that arranges services in levels showing the way service compositions can be made to meet a user requirements, i.e, such a tree can be viewed as a result to a user query. The paths of a tree depicts chaining of services due to matching of input and output parameters.

The proposed approaches are analyzed experimentally and based on the developed concept a tool has been designed to build extended service registry and automate both service search and composition. In a nutshell, this thesis proposes a variant of input/output parameter based search and composition that uses service meta information and offer results better than the existing ones following the same technique.

# TABLE OF CONTENTS

<b>DECLARATION</b>	<b>i</b>
<b>ACKNOWLEDGEMENTS</b>	<b>iii</b>
<b>ABSTRACT</b>	<b>vi</b>
<b>LIST OF TABLES</b>	<b>xiii</b>
<b>LIST OF FIGURES</b>	<b>xvi</b>
<b>ABBREVIATIONS</b>	<b>xvi</b>
<b>NOTATIONS</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Introduction to web services . . . . .	1
1.1.1 Web services technologies . . . . .	2
1.2 Web service search and composition . . . . .	3
1.2.1 UDDI . . . . .	4
1.2.2 WSDL . . . . .	6
1.2.3 Publishing and finding WSDL descriptions in UDDI . . . . .	7
1.2.4 QoS based service selection . . . . .	9
1.3 Thesis organization . . . . .	11
1.3.1 Motivating Example . . . . .	12
1.3.2 Problem Statement . . . . .	14
1.3.3 Proposed Solutions . . . . .	17
1.3.4 List of Publications . . . . .	24

<b>2</b>	<b>Literature Survey</b>	<b>26</b>
2.1	Introduction . . . . .	26
2.2	Web services search and composition techniques . . . . .	28
2.2.1	Composition approaches . . . . .	30
2.2.2	Graph based service composition techniques . . . . .	32
2.2.3	RDBMS based service composition techniques . . . . .	35
2.3	Clustering web services . . . . .	37
2.4	QoS Based service selection and ranking . . . . .	39
2.5	Conclusion . . . . .	43
<b>3</b>	<b>Extended Service Registry</b>	<b>44</b>
3.1	Introduction . . . . .	44
3.2	Types of service matches based on input/ output parameters . . . . .	47
3.3	Object Relational Database for service registry . . . . .	48
3.3.1	WSDL and UDDI Relationship . . . . .	48
3.3.2	Object relational database as service registry . . . . .	50
3.3.3	Registry Meta Information . . . . .	54
3.3.3.1	Registered services details . . . . .	56
3.3.3.2	Parameter clusters . . . . .	57
3.3.3.3	Web service clusters . . . . .	58
3.3.3.4	User query . . . . .	58
3.3.3.5	Keyword based service search . . . . .	59
3.3.3.6	I/O and QoS based service search . . . . .	59
3.3.3.7	CST Construction . . . . .	62
3.4	Performance and Scalability of ESR . . . . .	63
3.4.1	Experimental setup . . . . .	63
3.4.2	Comparison of basic keyword search on conventional registries vs ESR approach . . . . .	64
3.4.3	Process of parameter based service search . . . . .	66
3.4.4	Performance of output parameter based service search . . . . .	67

3.5	Conclusion . . . . .	67
<b>4</b>	<b>Accelerating parameter based service search</b>	<b>69</b>
4.1	Introduction . . . . .	69
4.2	Semantics for service search . . . . .	73
4.2.1	Similarity metrics . . . . .	73
4.2.1.1	Tokenizing web service input/output parameters .	74
4.2.1.2	Parameter similarity function ( $pSim$ ) . . . . .	75
4.2.1.3	Maximization function ( $maxMeas$ ) . . . . .	75
4.2.2	Term equivalence based parameter clustering . . . . .	79
4.2.2.1	Definitions . . . . .	79
4.2.3	Parameter clustering algorithm . . . . .	80
4.2.3.1	Cluster validation . . . . .	81
4.3	Output parameter pattern based service clustering . . . . .	82
4.3.1	Definitions . . . . .	83
4.3.2	Frequent output parameter pattern based clustering algorithm	86
4.3.3	Accelerating output parameter based service search . . . . .	87
4.4	Experiments . . . . .	88
4.4.1	Experimental Setup . . . . .	88
4.4.2	Parameter clustering results . . . . .	89
4.4.2.1	Evaluating quality of parameter clustering . . . . .	90
4.4.2.2	Performance of parameter clustering . . . . .	90
4.4.3	Web services clustering results . . . . .	92
4.4.3.1	Performance Analysis . . . . .	92
4.4.3.2	Impact of Preference factor on Number of Clusters	93
4.4.3.3	Impact of Preference Factor on Cluster Overlap .	93
4.4.3.4	Performance improvement obtained in output parameter based service search . . . . .	94
4.5	Conclusion . . . . .	95

<b>5</b>	<b>Parameter based service selection and composition</b>	<b>98</b>
5.1	Introduction . . . . .	99
5.2	I/O parameter and QoS based service selection . . . . .	101
5.2.1	Functional Match . . . . .	102
5.2.1.1	Output parameter deviation measure . . . . .	103
5.2.1.2	Input parameter deviation measure . . . . .	104
5.2.1.3	Combining input/output deviation measures . . . . .	105
5.2.2	QoS based service selection . . . . .	106
5.2.2.1	QoS model used . . . . .	107
5.2.2.2	Selection method . . . . .	108
5.3	Query processing: construction of composition search tree (CST) . . . . .	110
5.3.1	Service composition types . . . . .	111
5.3.2	Process of CST Construction . . . . .	113
5.3.3	CST Construction Algorithm . . . . .	118
5.3.4	Finding a composition . . . . .	121
5.4	Finding best composition . . . . .	122
5.4.1	Observations . . . . .	125
5.5	Experimental results . . . . .	125
5.5.0.1	Experimental setup . . . . .	126
5.5.0.2	Quality of service matches . . . . .	126
5.5.0.3	Performance comparison . . . . .	127
5.6	Conclusion . . . . .	128
<b>6</b>	<b>Tool Implementation</b>	<b>130</b>
6.1	Introduction . . . . .	130
6.2	System Architecture . . . . .	131
6.2.1	Pre-processing Module . . . . .	132
6.2.2	Service Search Module . . . . .	136
6.2.2.1	<i>ParClustering</i> Package . . . . .	136

6.2.2.2	<i>WSDLToClusters</i> Package . . . . .	136
6.2.2.3	<i>SearchInCl</i> Package . . . . .	141
6.2.3	Service Composition Module . . . . .	142
6.2.4	User Interface . . . . .	145
6.2.4.1	Software Requirements for Tool Implementation .	149
6.3	Tool Usage . . . . .	150
6.3.1	Screen-shots of Example 1 . . . . .	151
6.3.2	Screen-shots of Example 2 . . . . .	156
6.4	Conclusion . . . . .	162
<b>7</b>	<b>Conclusion and Future Work</b>	<b>163</b>
7.0.1	Future Work . . . . .	166
	<b>References</b>	<b>167</b>

## LIST OF TABLES

1.1	<b>Example web services</b> . . . . .	12
1.2	Services Matching ws8 . . . . .	13
2.1	<b>Example web services</b> . . . . .	29
2.2	Size of Graph generated for WSC data set . . . . .	34
2.3	Size of Graph generated for WSC data set . . . . .	35
3.1	<b>Example web services</b> . . . . .	51
3.2	Relational Schema for web services . . . . .	52
3.3	Relational Tables after further normalization . . . . .	52
4.1	Different placements of <i>Hotel</i> in parameters . . . . .	75
4.2	Illustration of Algorithm <i>FOPC</i> for Sample web services . . . . .	88
4.3	Average Precision and Recall of Clusters . . . . .	92
5.1	QOS Model of web service . . . . .	107
5.2	Abbreviations used in BNF . . . . .	114
5.3	Notations used in CST Algorithm . . . . .	118
5.4	<b>Example web services</b> . . . . .	120
5.5	<b>Solutions in CST</b> . . . . .	122

## LIST OF FIGURES

1.1	Service Oriented Architecture . . . . .	2
1.2	UDDI Core Data Structure . . . . .	5
1.3	WSDL Data Model . . . . .	6
1.4	Example WSDL document . . . . .	7
1.5	WSDL to UDDI Mapping . . . . .	8
1.6	Compositions Satisfying User Query . . . . .	14
1.7	Thesis Objective . . . . .	16
1.8	Thesis Contributions . . . . .	18
1.9	Example CST . . . . .	24
2.1	Parameter based composition example . . . . .	29
3.1	ER Diagram for Extended Service Registry . . . . .	56
3.2	jUDDI vs ESR for 500 services . . . . .	65
3.3	jUDDI vs ESR for 1000 services . . . . .	65
3.4	Service Search Time in jUDDI vs ESR . . . . .	66
3.5	Output parameter based search in ESR . . . . .	67
4.1	Pre-processing registry for service search . . . . .	72
4.2	Illustration of $maxMeas$ function . . . . .	77
4.3	Cluster cohesion of generated clusters . . . . .	90
4.4	Precision of Generated Clusters . . . . .	91
4.5	Recall of Generated Clusters . . . . .	92
4.6	Preference Factor vs Number of Clusters in CC . . . . .	93
4.7	Preference Factor vs Average Cluster Overlap for clusters in CC . . . . .	94
4.8	Output Parameter based Search Using Clusters . . . . .	95



5.1	Bi-level service selection approach . . . . .	102
5.2	BNF of a CST Node . . . . .	114
5.3	Example CST . . . . .	120
5.4	Service matches of Constraint method v/s Chen's method v/s available services in registry . . . . .	128
5.5	Performance comparison of Constraint method v/s Chen's method .	128
6.1	ESR System Architecture . . . . .	132
6.2	Class Diagram for Pre-processing Module . . . . .	133
6.3	Sequence Diagram for MyDataSetToDB . . . . .	134
6.4	Sequence Diagram for QoSdataToDB . . . . .	135
6.5	Class Diagram for parameter Clustering Package . . . . .	137
6.6	Sequence Diagram for ParClustersToDB . . . . .	138
6.7	Class Diagram for Web Services Clustering Package . . . . .	139
6.8	Sequence Diagram for SCDToClFinalNoCO . . . . .	140
6.9	Class Diagram for Service Search Package . . . . .	143
6.10	Sequence Diagram for SearchIOQoSMatch . . . . .	144
6.11	Class Diagram for service composition Module . . . . .	146
6.12	Sequence Diagram for CSTreeConstruction . . . . .	147
6.13	Sequence Diagram for TreeConstruction method . . . . .	148
6.14	Main Page . . . . .	150
6.15	Main Page of Example 1 . . . . .	152
6.16	Values Provided in User Query . . . . .	153
6.17	Web Services Matching User Query . . . . .	154
6.18	Compositions Satisfying User Query . . . . .	155
6.19	Main Page of Example 2 . . . . .	157
6.20	Values Provided in User Query . . . . .	158
6.21	Web Services Matching User Query . . . . .	159
6.22	Web Services Matching User Query . . . . .	160
6.23	Compositions Satisfying User Query . . . . .	161

## ABBREVIATIONS

<b>API</b>	Application Programming Interface
<b>BPEL</b>	Business Process Execution Language
<b>CST</b>	Composition Search Tree
<b>DBSCAN</b>	Density-Based Spatial Clustering of Applications with Noise algorithm
<b>ESR</b>	Extended Service Registry
<b>FOPC</b>	Frequent Output Parameter based Clustering algorithm
<b>IDE</b>	Integrated Development Environment
<b>QoS</b>	Quality of Service
<b>RTD</b>	Round Trip Delay
<b>SCSP</b>	Similarity based CLustering of Service Parameters algorithm
<b>SOA</b>	Service Oriented Architecture
<b>SOAP</b>	Simple Object Access Protocol
<b>UDDI</b>	Universal Description, Discovery and Integration
<b>URL</b>	Uniform Resource Locator
<b>WSDL</b>	Web Service Description Language
<b>WSM</b>	Weighted Sum Model
<b>WS4J</b>	WordNet Similarity for Java package
<b>XML</b>	Extensible Markup Language
<b>QWS</b>	Quality of Web Services dataset

## NOTATIONS

$ws$	A web service
$ws^I$	Input parameters of a web service
$ws^O$	Output parameters of a web service
$Q$	User query
$Q^I$	Input parameters specified in user query
$Q^O$	Output parameters specified in user query
$Q^{QoS}$	QoS attribute values specified in user query
$R$	Service registry
$P$	Set of parameters in a service registry
$W$	Set of registered web services in a service registry
$WS$	Set of web services participating in a service composition
$I/O$	Input/Output
$ws_P$	Preceding web service
$ws_S$	Succeeding web service
$pf$	preference factor
$CCl$	Candidate Cluster
$CC$	Covering Cluster
$WS_D$	Desired web service
$WS_M$	Matched web service
$DM_P^I$	Input parameter deviation measure
$DM_P^O$	Output parameter deviation measure
$DM_P^{IO}$	Input output parameters deviation measure
$q_{RT}$	Response Time of a web service
$q_R$	Reliability of a web service
$q_A$	Availability of a web service
$q_{RT}$	Cost of a web service
$\epsilon_{RT}$	Response time value fed in user query
$\epsilon_R$	Reliability time value fed in user query
$\epsilon_A$	Availability time value fed in user query
$\epsilon_C$	Cost value fed in user query
$NWS$	Number of web services in a CST node
$SWS$	Set of web services in a service composition
$R_I^{EC}$	Required input parameters for exact composition
$R_I^{SC}$	Required input parameters for super composition
$R_I^{CC}$	Required input parameters for collaborative composition

# CHAPTER 1

## Introduction

### Abstract

Various approaches can be used for service search, such as, searching in UDDI, Web and service portals. We take up the issue of searching at service registries, that are based on UDDI, for its practicality in business world. In this research, we explore the feasibility of input/output parameter based web service search and composition, as an extension to UDDI, to support varying requirements of a user. We propose to build an *extended service registry(ESR)* system capable of offering parameter based web service search and composition operations. In this chapter we first give a brief introduction to web services in section 1.1. Basics of service search, service composition and their underlying technologies like WSDL, UDDI, etc are explained in section 1.2. In section 1.3 that follows, we present thesis organization, problem statement and a brief introduction to the proposed solutions.

### 1.1 Introduction to web services

Web services are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services are driven by the SOA paradigm, which describes the relationships that exist among service providers, consumers and service brokers, thereby providing an abstract execution environment for web services. Service-Oriented Architecture (SOA) is an architectural style that supports a way of thinking in terms of services and service-based development and the outcomes of services. A service is a logical representation of a repeatable business activity that has a specified outcome (SOA (2013)). Figure 1.1 depicts the Service Oriented Architecture with its three components : *service provider*, *service consumer* and *discovery agency*. A *service provider* is responsible for building a service and publishing it,

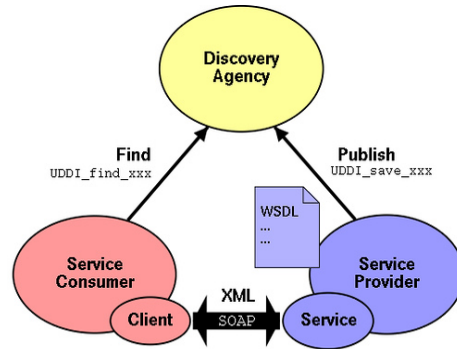


Figure 1.1: Service Oriented Architecture

thereby making it available for the consumer requests. A *discovery agency* is responsible for maintaining the public registry in which the description of services are stored, allowing users to find the services that best meet their needs. A *service consumer* represents a potential user for published services. By means of the *find operation*, users interact with the *discovery agency* to find the services that best meet their needs.

### 1.1.1 Web services technologies

Web services framework is built on three main technologies (Curbera *et al.* (2002)):

1. **Simple Object Access Protocol (SOAP):** The simple object access protocol (SOAP) enables communication among web services. A SOAP message has a very simple structure: an XML element having two child elements - a header and the body, which are themselves arbitrary XML elements.
2. **Universal Description Discovery and Integration (UDDI):** UDDI (Version 3) (Carolgeyer (2013)) offers an industry specification that is used for building flexible, inter operable XML web services registries useful in private as well as public deployments. It offers clients and implementers a comprehensive and complete blueprint of description and discovery foundation for a diverse set of web services

architectures.

3. **Web Service Description Language (WSDL):** WSDL is a XML language for describing web services. It contains service interface defined as a set of operations and messages, their protocol bindings, and the deployment details.

The advent of semantics led to two main categories of specifications for web services viz. ontology based (OWL-S and WSMO) and keyword based(WSDL). A service of former category is built with domain ontology whereas of latter case keywords are used for specifying inputs and outputs. WSDL is popular and adopted by the industry due to its simplicity, while OWL-S (formerly DAML-S) and WSMO are well accepted by researchers as they offer much structured and detailed semantic mark-ups. The term web services is used for services described in WSDL and whereas semantic web services mean those described using either OWL-S or WSMO. A clear separation of both is necessary since the techniques, be it service search, service selection or service composition, for the two types can be quite different. Though, semantic web services are versatile in nature still keyword based service specification is popular for its fast response avoiding reference to domain ontology that could be at times large and complex enough for the purpose. Hence **we have taken up keyword based service specification** for our research.

## 1.2 Web service search and composition

As growing number of services are being available, selecting the most relevant web service fulfilling the requirements of a user query is indeed challenging. *service discovery* (or *service search*) is a process of searching web services matching a given set of user functional and non-functional requirements. *Service composition* can be defined as creating a composite service, obtained by combining available web services. If no single web service in the registry can satisfy the functionality required by the user, there is a need to combine existing services together in order to fulfill the user request, resulting

in a service composition. Web service composition still is a highly complex task due to the dramatical increase in the number of services available during the recent years and also due to dynamic nature of the services.

Various approaches can be used for service search, such as, searching in UDDI, web and service portals. **We take up the issue of web service searching and composition at service registries** for its practicality in business world as providers would like to post their services centrally, as searching there is less time consuming than searching on world wide web.

### 1.2.1 UDDI

Universal Description, Discovery and Integration (UDDI) (Carolgeyer (2013)) is an industry standard for service registries, developed to solve the web service search problem. A UDDI information model is composed of four primary data structures:

1. *businessEntity* : used to describe a business or service provider that typically provides web services.
2. *businessService* : used to represent business descriptions for a web service. Describes a collection of related web services offered by a businessEntity.
3. *bindingTemplate* : contains the technical information associated to a particular service. Each bindingTemplate describes an instance of a web service offered at a particular network address, typically given in the form of a URL. The bindingTemplate also describes the type of web service being offered using references to tModels, application-specific parameters, and settings.
4. *tModel* : used to define the technical specification for a web service, such as a web service type, a protocol used by web services, or a category system.

A *businessEntity* can contain one or more *businessServices*. A *businessService* can have several *bindingTemplates* and each *bindingTemplate* contains a reference to one or more tModels. The data structures and their relationships(Carolgeyer (2013)) are shown in figure.1.2.

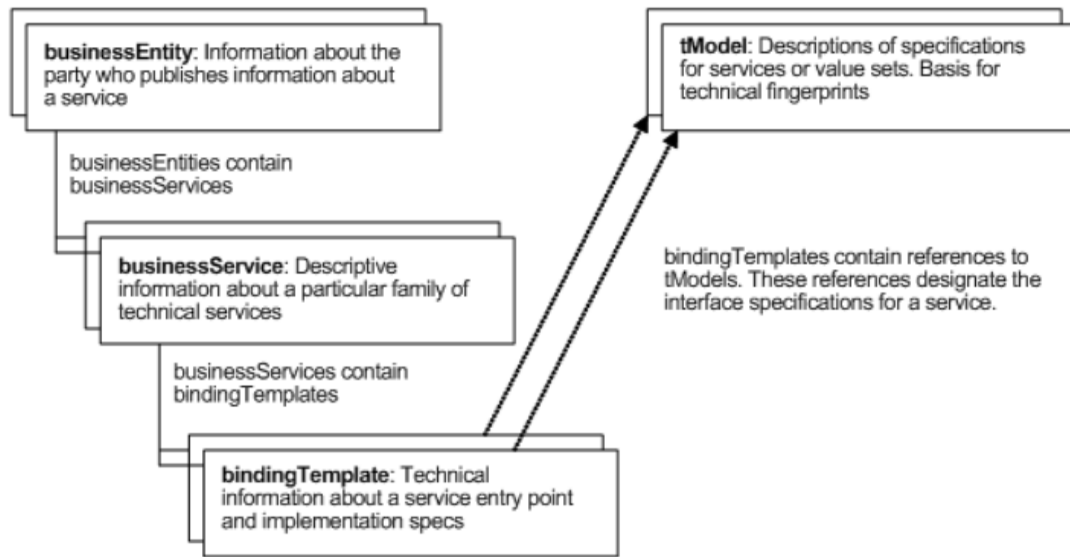


Figure 1.2: UDDI Core Data Structure

Basically, UDDI supports service search by name, location, business, bindings or tModels, and binds two services based on composability of their protocols(Carolgeyer (2013)). The search API is limited by the kind of information that is available and searchable in UDDI entries and do not provide any support for complex searches like input/output parameter based search and automatic composition of web services. Also, our experiments show that average response time of current UDDI implementations increase with a substantial increase in number of services registered.



## 1.2.2 WSDL

WSDL complements the UDDI standard by providing a uniform way of describing the abstract interface and protocol bindings of arbitrary network services. Figure.1.3 depicts the WSDL data model and an example WSDL document is shown in figure.1.4. WSDL document contains six major elements that defines web services(WSDL (2001))

:

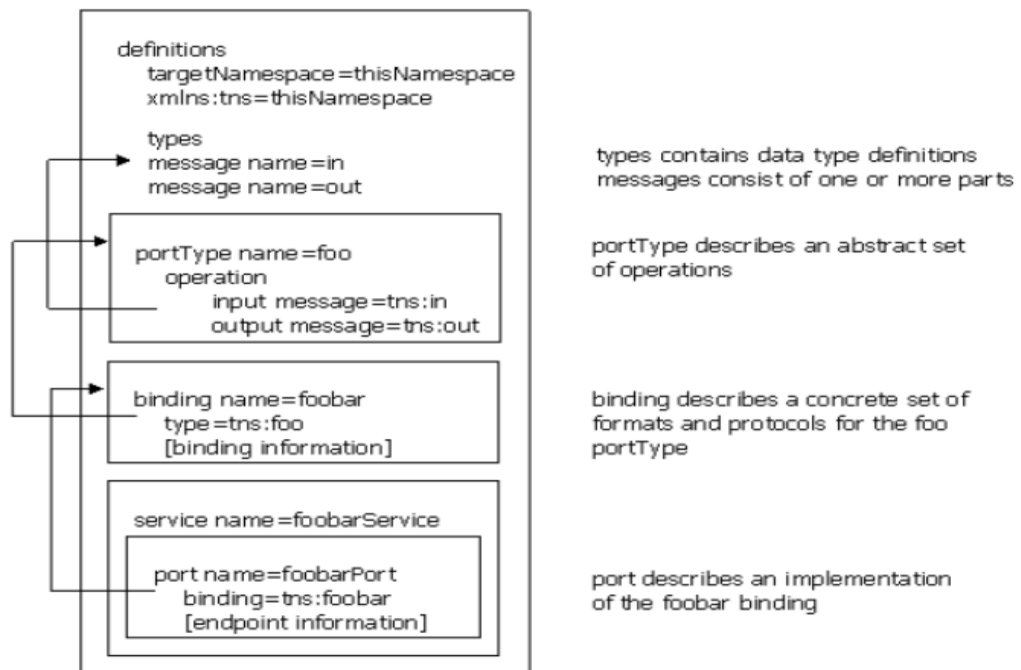


Figure 1.3: WSDL Data Model

1. *types* : provides data type definitions used to describe the messages exchanged.
2. *message* : provides an abstract definition of the data being transmitted and consists of logical parts.
3. *portType* : defines a set of abstract operations each referring to an input message and output message.
4. *binding* : specifies concrete protocol and data format specifications for the operations and messages defined by a particular portType.

5. *port* : specifies an address for a binding, defining a single communication end-point.
6. *service* : aggregates a set of related ports.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions name="StockQuote" targetNamespace="http://example.com/stockquote/" xmlns:tns=http://example.com/stockquote/
xmlns:xsd1="http://example.com/stockquote/schema" xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema
      targetNamespace="http://example.com/stockquote/schema/"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="TradePriceRequest">
        <complexType><all><element name="tickerSymbol" type="string"/></all></complexType>
      </element>
      <element name="TradePrice">
        <complexType><all><element name="price" type="float"/></all></complexType>
      </element>
    </schema>
  </types>
  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>
  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>
  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>
  <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    ....
  </binding>
  <service name="StockQuoteService"> <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
    <soap:address location="http://location/sample"/> </port>
  </service>
</definitions>
```

Figure 1.4: Example WSDL document

### 1.2.3 Publishing and finding WSDL descriptions in UDDI

Here we describe the process for publishing and searching a service, described by a WSDL document. The WSDL document of a service describes instances of the service using a WSDL service element. Each service element in a WSDL document is

used to publish a UDDI businessService. The service interface described using WSDL portType and binding is published as a UDDI tModel before publishing a businessService. Figure 1.5 depicts how the various elements of WSDL are mapped to UDDI data structures. A summary of the mapping is as follows:

- WSDL portType element is mapped to UDDI tModel.
- WSDL binding element is mapped to UDDI tModel.
- WSDL port element is mapped to UDDI bindingTemplate.
- WSDL service element is mapped to UDDI businessService.

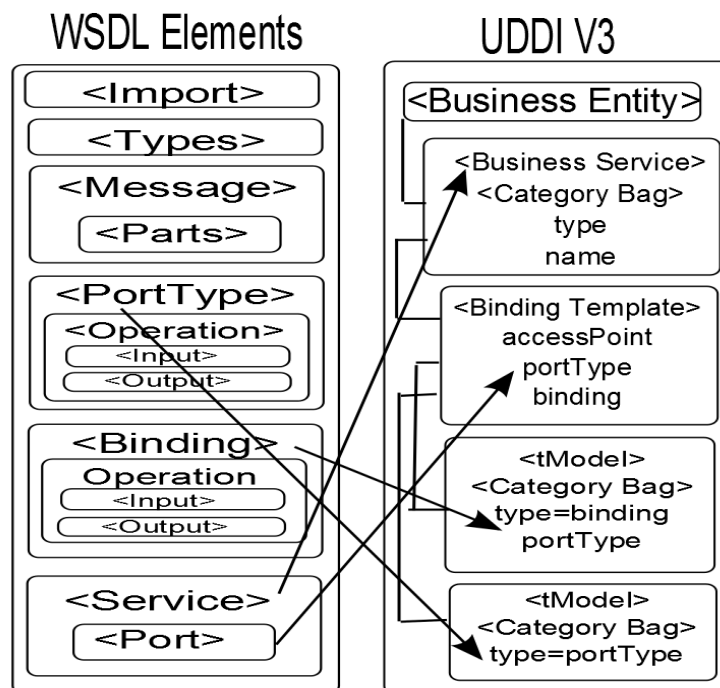


Figure 1.5: WSDL to UDDI Mapping

On contemplating the relationship between WSDL and UDDI, we observe that there are many sub-elements in WSDL that are not mapped to UDDI types like input message, output message, message parts, etc. Although these details can be added in tModel of the service, there are no supporting search functionalities provided in UDDI standards.

This leads to the limited and poor search functionalities supported by UDDI. **We hence propose to include these details in an *extended service registry*, enabling the registry to support additional search functionalities.**

#### 1.2.4 QoS based service selection

One of the critical challenges in web service search and composition is the selection of appropriate services that best match the user requirements, either to be executed independently (when an individual service matches the requirements) or to be bound to the service composition, from the pool of matching web services in the service registry. The current UDDI model limits the service discovery to functional requirements only. Often there are many web services available that meet the given user functional requirements. Non-functional features of web services play an important role in differentiating services with similar functionalities. Services non-functional features can be best represented with the Quality of Service(QoS) offered by the web services, such as price, response time, reliability , and so on(Ran (2003)). Hence incorporating quality of service into service selection process becomes very important.

There are different Quality of service attributes defined in the literature, such as, performance, reliability, scalability, capacity, robustness, exception handling, accuracy, integrity, accessibility, availability, interoperability, security, and network-related QoS requirements(QoS (2003)). The most popular of these quality aspects is defined and their desired requirements are described as below:

1. *Performance* : The performance of a web service represents how fast a service request can be completed and can be measured in terms of throughput, response time, latency, execution time, etc.
  - *Throughput* is the number of web service requests served in a given time

interval.

- *Response time* is the time required to complete a web service request.
- *Latency* is the round-trip delay (RTD) between sending a request and receiving the response.
- *Execution time* is the time taken by a web service to process its sequence of activities.

In general, web services with high quality provide higher throughput, faster response time, lower latency, lower execution time.

2. *Reliability* : Reliability is the quality aspect of a web service that represents the degree of being capable of maintaining the service and service quality. The number of failures per day, week, month or year represents a measure of reliability of a web service. In another sense, reliability refers to the assured and ordered delivery for messages being sent and received by service requesters and service providers.
3. *Availability* : Availability is the quality aspect of whether the web service is present or ready for immediate use. Availability represents the probability that a service is available. Larger values represent that the service is always ready to use while smaller values indicate unpredictability of whether the service will be available at a particular time.
4. *Price* : Price is the amount of money the requester has to pay for using the service.

QoS attributes are either *positive*, for which higher values indicates better quality, Eg: availability, reliability, etc or *negative*, for which lower values indicate better quality, Eg: price, response time, etc. **We present a QoS model considering 4 QoS attributes - *response time, availability, reliability* and *price*, for our proposed service selection approach.**

## 1.3 Thesis organization

As growing number of services are being available, selecting the most relevant web service fulfilling the requirements of a user query is indeed challenging. Various approaches can be used for service search, such as, searching in UDDI, web and service portals, as discussed before. **We take up the issue of searching at service registries, that are based on UDDI**, for its practicality in business world as providers would like to post their services centrally, as searching there is less time consuming than searching on world wide web.

Often consumers may be unaware of exact service names that's fixed by service providers. Rather consumers being well aware of their requirements would like to search a service based on their commitments(inputs) and expectations (outputs). UDDI offers limited search functionalities due to the limited information that is available and searchable in UDDI entries and do not provide any support for complex searches like input/output parameter based search and automatic composition of web services. Also, average response time of current UDDI implementations increase with a substantial increase in number of services registered.

The limitations of UDDI motivated us to explore the feasibility of input/output parameter based web service search and composition, as an extension to UDDI, to support varying requirements of a user. We propose to build an *extended service registry*(ESR) system capable of offering parameter based web service search and composition operations. The various research problems we encountered and the corresponding solutions proposed are presented in this section. We first present a motivating example.

### 1.3.1 Motivating Example

Web services composition gives us a possibility to fulfill the user query when no single web service in the registry can satisfy a user requirement. Each web service in the registry has a web service name, input and output parameters. It is assumed that a web service when invoked with all input parameters returns the output parameters.

Table 1.1 depicts example web services considered for service composition. Let us assume that a user wants to find a web service that takes  $\{Date, City\}$  as input and outputs  $\{HotelName, FlightInfo, CarType, TourCost\}$ . From web services in Table 1.1 it is seen that there is no single web service that satisfies the user query, hence service composition becomes inevitable.

service No	service Name	input parameters	output parameters
ws1	HotelBooking	Period, City	HotelName, HotelCost
ws2	AirlineReservation	Date, City	FlightInfo, FlightCost
ws3	TaxiInfo	Date, City	CarType, TaxiCost
ws4	DisplayTourInfo	HotelName, FlightInfo, CarType	TourInfo
ws5	TaxiReservation	CarType, Date, City	TaxiCost
ws6	TourPeriod	Date, City	Period
ws7	TourCost	TourInfo	TourCost
ws8	AgentPackage	PackageID	Period, TourInfo
ws9	TourPackages	Date, City	PackageID
ws10	TourReservation	Period, TourInfo	HotelName, FlightInfo, CarType, TourCost
ws11	PackageDetails	PackageID	HotelName, Hotelcost, FlightInfo, FlightCost, CarType, TaxiCost, TourCost

Table 1.1: Example web services

A web service,  $ws$ , has typically two sets of parameters - set of inputs  $ws^I$  and set of outputs  $ws^O$ . When a user is looking for a web service for a given input and desired output parameters and there is no single web service in the service registry satisfying the request, service composition becomes necessary. By service composition, we mean making of a new service(that does not exist on its own) from existing services registered in UDDI. Conventionally two services  $ws_i$  and  $ws_j$  are said to be composable iff

$ws_i^O = ws_j^I$ , i.e,  $ws_j$  receives all the required inputs from outputs  $ws_i$  has. A service composition is formed by constructing a chain of such composable services. However, the making of a service composition chain may fail at a point when the output parameters of a preceding service ( $ws_P^O$ ) does not match exactly with the input parameters of a succeeding service( $ws_S^I$ ).

If this classical definition of service composability is relaxed to include partial composable services having  $ws_i^O \subset ws_j^I$  and super composable services having  $ws_i^O \supset ws_j^I$ , then many compositions can be formed that satisfy a given query. As an example, considering the services in Table 1.1, services that are composable with web service 'ws8', i.e, web services whose inputs are composable with outputs of 'ws8', are listed in Table 1.2.

service Matched	Matching parameters	Composability Type
ws1	Period	Partial Composable
ws7	TourInfo	Super Composable
ws10	Period,TourInfo	Exact Composable

Table 1.2: Services Matching ws8

In order to find the various compositions satisfying a given query, initially services satisfying the desired output parameters is found and classified according to the types of composability. A best matching service in each type is selected for further composition. The selected services may need some extra input parameters, other than those provided in user query. These input parameters now become desired output parameters for the next step. This process is repeated until there are no more extra input parameters required for a selected service. This process of finding compositions satisfying a given query can be visualized as a *composition search tree*.

The tree in Figure 1.6, graphically represents the compositions satisfying the example query, where  $Q^O$  represents the desired output parameters. In the tree, the path from a leaf node to root node represents a composition that satisfies the given user requirement. As seen there are five such paths with each path involving services with different



composability types at different levels.

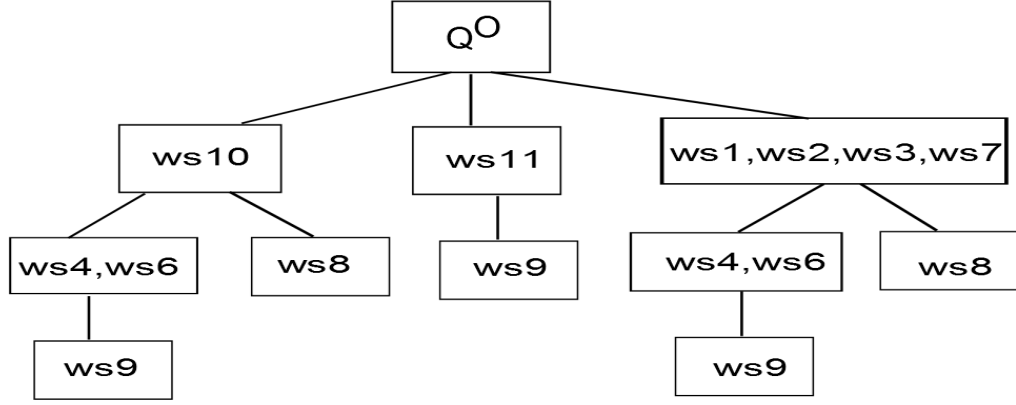


Figure 1.6: Compositions Satisfying User Query

### 1.3.2 Problem Statement

**Objective:**

*To build an extended service registry(ESR) capable of offering parameter based web service search and composition operations.*

**Formalizing Problem Statement :**

Given a service registry  $R = \langle P, W \rangle$  and a query  $Q = \langle Q^I, Q^O, Q^{QoS} \rangle$ , we need to find set of web services,  $WS \subseteq W$ ,  $WS = \{ws_1, ws_2, \dots, ws_n\}$ , such that services in WS can be composed to obtain  $Q^O$ , i.e.,  $\{ws_1^O \cup ws_2^O \cup \dots \cup ws_n^O\} \supseteq Q^O$ , where

- P is a set of parameters,  $P = \{P_1, P_2, \dots, P_n\}$ . In the motivating example,  $P = \{ \text{Period, City, HotelName, HotelCost, } \dots \}$ .
- W is a set of web services in the registry,  $W = \{ws_1, ws_2, \dots, ws_n\}$ . In the motivating example,  $W = \{ \text{HotelBooking, AirlineReservation, TaxiInfo, } \dots \}$ .
- $ws_i^O$  is a set of output parameters of web service  $ws_i$ , and  $\{ws_1^O \cup ws_2^O \cup \dots \cup ws_n^O\}$  is the union of output parameters of  $ws_1^O, ws_2^O, \dots, ws_n^O$ .

- $Q^I \subset P$  is set of initial input parameters.
- $Q^O \subset P$  is set of desired output parameters.
- $Q^{QoS}$  is a set of desired QoS attribute values.

### **Research Problems:**

We assume that a user query  $Q$  is built with the following components:

- A set of input parameters  $Q^I$ , that are provided by the user.
- A set of output parameters  $Q^O$ , that a user expects from the searched web service.
- A set of QoS attribute values  $Q^{QoS}$ , that the user expects from the searched web service.

We propose to add *extended service registry* to UDDI architecture such that the augmented UDDI can support the following functionalities:

1. Extended service registry for I/O parameter based service search.
2. Searching for services that satisfy a user query  $Q$ .
3. Finding service Compositions satisfying a user Query  $Q$ .

The usages of the proposed architecture is also presented pictorially in figure1.7.

To realize such an *extended service registry* we had to find solutions for the following problems -

- How do we extend the service registry to store the input/output parameter information of the registered web services?
- How do we provide support for parameter based web service search in the extended registry?

For this problem, we had to first think of solutions to the following sub-problems:

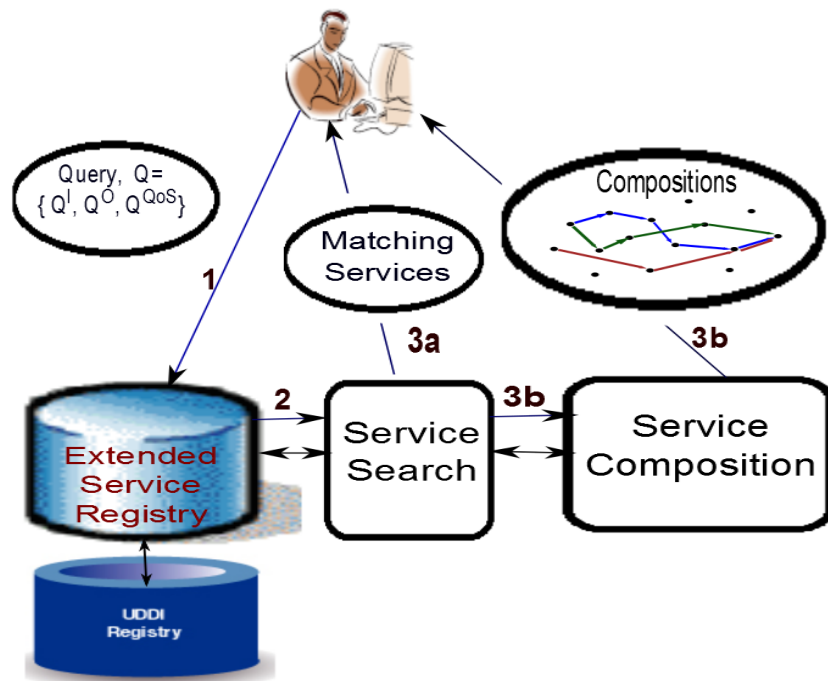


Figure 1.7: Thesis Objective

- How do we pre-process the service registry to group similar parameters?  
This step is essential to extend the scope of search, since parameter names are not standardized and are usually built using terms that are highly varied due to the use of synonyms, hypernyms, and different naming conventions.
- Can we improve parameter based service search in terms of time taken for search?
- How do we select the best matching web service from the pool of services that matches the given requirements?
- How do we find various compositions (involving the different types of service composabilities as seen in the motivating example) satisfying a given user query?

Solutions to the problems listed above make the contributions of this thesis. We briefly explain these solutions and describe the thesis organization in the next subsection.

### 1.3.3 Proposed Solutions

We have worked out to achieve our objective by finding solutions for the problems listed above. We propose the following approaches for each of them :

- **Extending service registry:** We propose the use of Object Relational database as repository of web services.
- **Parameter based web service search:** The extended service registry as such supports parameter based service search. In order to widen the scope of search and to improve the performance of parameter based search we propose the following approaches -
  - Term equivalence based parameter clustering.
  - output parameter Pattern based service clustering.
- **Service selection:** We propose a lexical model considering both functional and non-functional requirements for service selection.
- **Parameter based service composition:** Adopting the three types of service composability as introduced in the motivating example, we propose an algorithm for finding compositions satisfying a given user query. The composition process is visualized as a *Composition search Tree*.

The thesis contributions are depicted in fig1.8.

#### 1. Extending service registry

The limitations of UDDI, as discussed before, motivated us to **build an extended service registry(ESR) system capable of offering parameter based web service search and Composition operations**. We propose the use of Object Relational Database as repository of web services, in chapter 3. Information about the web services, extracted from their WSDLs, are stored in tables and relational algebraic operators are used for

service search and composition.

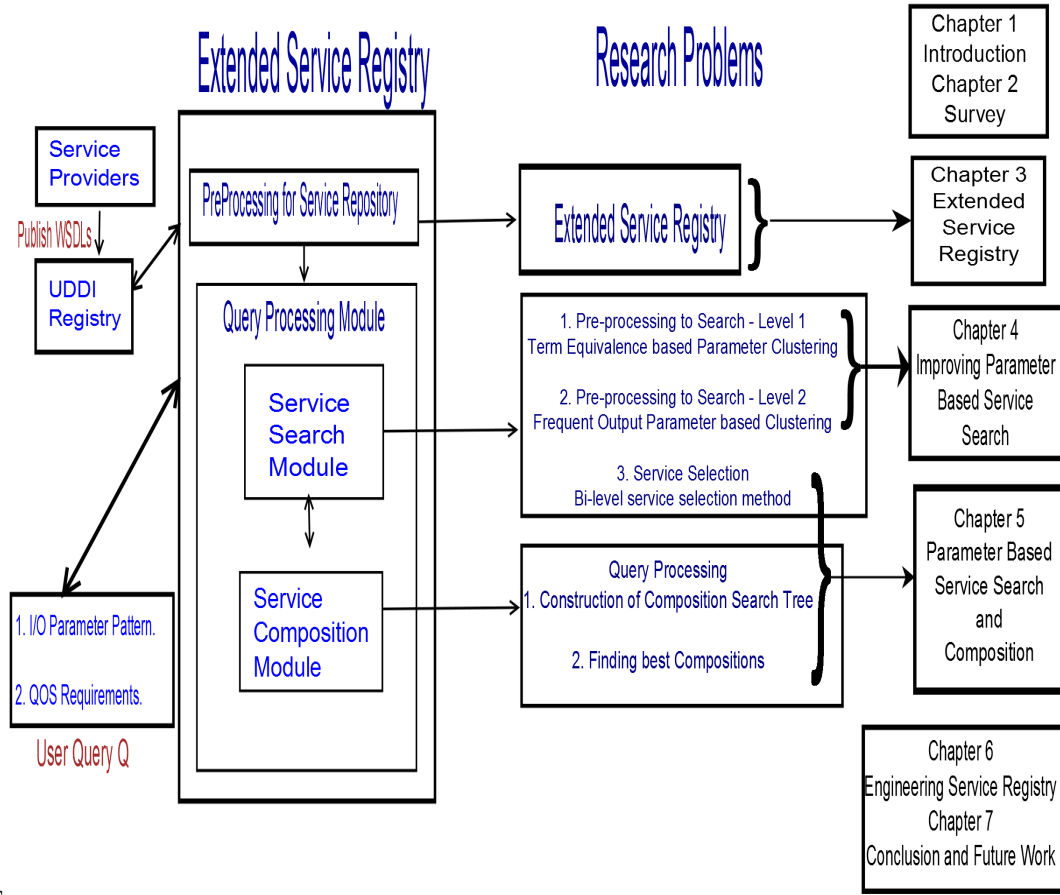


Figure 1.8: Thesis Contributions

A web service,  $ws$ , has typically two sets of parameters - set of inputs  $ws^I$  and set of outputs  $ws^O$ . By service composition, we mean making of a new service(that does not exist on its own) from existing services registered in UDDI. Conventionally two services  $ws_i$  and  $ws_j$  are said to be composable iff  $ws_i^O = ws_j^I$ , i.e,  $ws_j$  receives all the required inputs from outputs  $ws_i$  has. A service composition is formed by constructing a chain of such composable services.

When a user is looking for a web service for given input and desired output parameters and there is no single web service in the service registry satisfying the request, service composition becomes necessary. However, the making of a service composi-

tion chain may fail at a point when the output parameters of a preceding service ( $ws_P^O$ ) does not match exactly with the input parameters of a succeeding service( $ws_S^I$ ). **We propose an approach to alleviate this problem by making match criteria flexible. In addition to *exact* match we allow *partial* as well as *super* match for conditions  $ws_i^O \subset ws_j^I$  and  $ws_i^O \supset ws_j^I$  respectively**(Lakshmi and Mohanty (2012)). Further, to avoid the need for multiple joins and to speed up querying, **we propose to use multi-valued attributes for storing input and output parameters in our database design**, and hence use an object relational database for our proposed **extended service registry**. Based on this concept we have **explored the feasibility of input/output based web service search and composition in our proposed ESR system**, to support varying requirements of the consumer. The proposed **extended service registry** is explained in detail in chapter 3.

### **Pre-processing for parameter based web service search**

To widen the scope of search and to further improve the performance of parameter based search in our ESR we propose a two-level pre-processing of service registry.

#### ***Level 1: Term equivalence based parameter clustering.***

A parameter name,(eg: FlightInfo, CheckHotelCost), is typically a sequence of concatenated words,(eg: Flight,Check,Hotel,Cost), referred to as terms. For effectively matching I/O parameters of web services, it is essential to consider their underlying semantics. However, this is hard since parameter names are not standardized and are usually built using terms that are highly varied due to the use of synonyms, hypernyms, and different naming conventions. Hence, to widen the scope of I/O parameter matching, it is useful to cluster service I/O parameters based on their similarity.

We make use of semantic similarity and co-occurrence of terms in parameter names, computed using WordNet as the underlying ontology, to cluster service parameters into semantic groups. We also propose a semantic similarity measure for computing similarity of parameters and an approach for clustering service parameters based on the

classical DBSCAN algorithm(Ester *et al.* (1996)). The key idea in our clustering is that a group of parameters having  $\epsilon$  similarity with a core parameter(representing a cluster center) forms a cluster. The minimum number of parameters that a cluster should contain is governed by *MinSP* and  $\epsilon$  is the minimum similarity value that cluster parameters should have with respect to core parameter of a cluster. The parameter clustering serves as a pre-processing step for I/O parameter based web service search. Experimental evidence shows that the clusters generated by our algorithm has a better Precision and Recall values when compared with the clusters generated by K-means algorithm(MacQueen *et al.* (1967)). This approach is discussed in detail in chapter 4.

***Level 2: Output parameter pattern based service clustering.***

To improve the performance of parameter based search in our extended service registry, we propose an approach to cluster services based on their output parameter pattern similarity, in chapter 4. This approach utilizes the parameter clusters obtained in the previous step. Each output parameter of a web service is represented by the *cluster id* to which it belongs to, and hence the output parameter pattern is now reduced to a pattern of parameter cluster ids. It is on these patterns that we apply our clustering process.

Clustering of services in a web registry reduces search space while looking for a service from the registry to meet a search requirement. The current approaches(Ma *et al.* (2008); Elgazzar *et al.* (2010); Liu and Wong (2009); Nayak and Lee (2007); Dong *et al.* (2004)) employ clustering during the service search process, to cluster search results for a given user query. Mostly, the approach(Ma *et al.* (2008)) of query processing follows two basic steps - reducing the domain of search by service domain e.g. hotel, transport etc. and then grouping the services based on query requirements. As these are done during processing of a query, it becomes expensive in terms of time. Instead, we propose an approach that does a book keeping of services following an order so that the pre-processing helps to reduce search time.

Our clustering approach makes use of the co-occurrence of output parameters to cluster web services. Set of output parameters that co-occur in more than a threshold number of web services make a *frequent output parameter pattern*. A set of services having the same frequent output parameter pattern make a *candidate cluster*. From the many candidate clusters generated we need to choose a subset of candidate clusters such that selected clusters form a *covering cluster*. A *covering cluster* covers all web services in the registry and also have a minimum overlap between them. Inclusion of a candidate cluster into *covering cluster* is governed by its *cluster overlap* and *preference factor*. We define cluster overlap of a candidate cluster based on the number of frequent output parameter patterns that the services in a candidate cluster is supporting. Further, a cluster's preference factor suggests its inclusion into *covering cluster*, based on the number of non-overlapping web services that it contributes to final clustering. A detailed description of the clustering approaches is given in chapter 4.

## 2. Service search and selection

When a user queries for a service with a required output parameter pattern, initially, a cluster from *covering cluster*, that best matches the queried pattern is searched for. This matched cluster contains many services whose parameter pattern matches with queried pattern. And these services are further classified into three groups: exact, partial and super, based on the type of match the output parameter pattern of a service has with queried output parameter pattern. One may obtain many services in each match type, from which best matching services need to be selected. Hence we propose a service selection approach as a next step.

One of the critical challenges in the area of service search and composition is to define a **service selection approach** that selects the most appropriate web services from the pool of services discovered. Most of the current approaches(Cai and Xu (2014); El Hadad *et al.* (2010); Li *et al.* (2014b); Mobedpour and Ding (2013); Yager *et al.* (2011)) select services based on their QoS values from a set of web services that are



functionally similar. These approaches usually apply a weighted sum model(WSM) as an evaluation method, represented as -

$$Score(WS) = \sum (q'_i * w_i) \quad (1.1)$$

where  $q'_i$  is a normalized QoS attribute value and  $w_i$  is the weight given to the QoS attribute. Such methods require users to express their preference over different (and sometimes conflicting) quality attributes as numeric weights and may fail to model user preferences precisely.

On the contrary, in order to model both the functional and non-functional requirements of users, we propose a **bi-level service selection approach**, explained in detail in chapter 5. The functional requirements are provided by the user as a set of input parameters provided for and output parameters desired from the web service. The user also provides a set of desired QoS values and the order of their preference for selection. In first level services matching the functional requirements are shortlisted, which are further filtered in second level based on given QoS requirements, thus providing a list of web services that best matches a given user query. Experiments were conducted using QWS dataset (Al-Masri and Mahmoud (2008)) to compare the second level(QoS based selection) of our approach with that of Chen's(Mobedpour and Ding (2013)) approach. Various sets of queries were fed for both the approaches and the results were analyzed on the quality of services selected and the execution time taken by both approaches. From the results obtained we can infer that our approach performs better and returns quality web services as compared with Chen's approach.

### 3. Parameter based service composition

Most of the existing algorithms for service composition(Kwon *et al.* (2007); Lee *et al.* (2011); Zeng *et al.* (2010)) consider services based on exact matches of input/output parameters for composition. However, for service composition the construction of such

a chain of services fails at a point when the output parameters of a preceding service ( $ws_P^O$ ) does not match exactly with the input parameters of a succeeding service ( $ws_S^I$ ). In chapter 3 we propose three types of service match : *exact match*, *super match* and *partial match*, extended from the conventional definition of a service match, to alleviate this problem.

Utilizing these types of matches, **we widen the scope of composition by defining possibly three types of service composition: exact, collaborative and super composition**. For a given user query, when service selection fails to find a matching service, we propose to find the possible compositions that satisfy the given user query. **The process of service composition is visualized as generation of a *composition search tree*** that arranges services in levels showing the way service compositions can be made to meet the user requirements, i.e, such a tree can be viewed as a result to consumer query. Further, **we propose the utility of *composition search tree* for finding best service compositions like *leanest composition* and *shortest depth composition***, from the many compositions found in *composition search tree*. The process of service composition and its utility for finding best compositions are explained in chapter 5.

Figure 1.9 depicts *composition search tree* generated when the web services in Table1.1 are used to construct the tree for a query with  $Q^I = \{Date, City\}$  and  $Q^O = \{HotelName, FlightInfo, CarType, TourCost\}$ .

Based on the developed concept a tool has been designed to build extended service registry and automate both service search and composition. The tool is explained in detail in chapter 6.

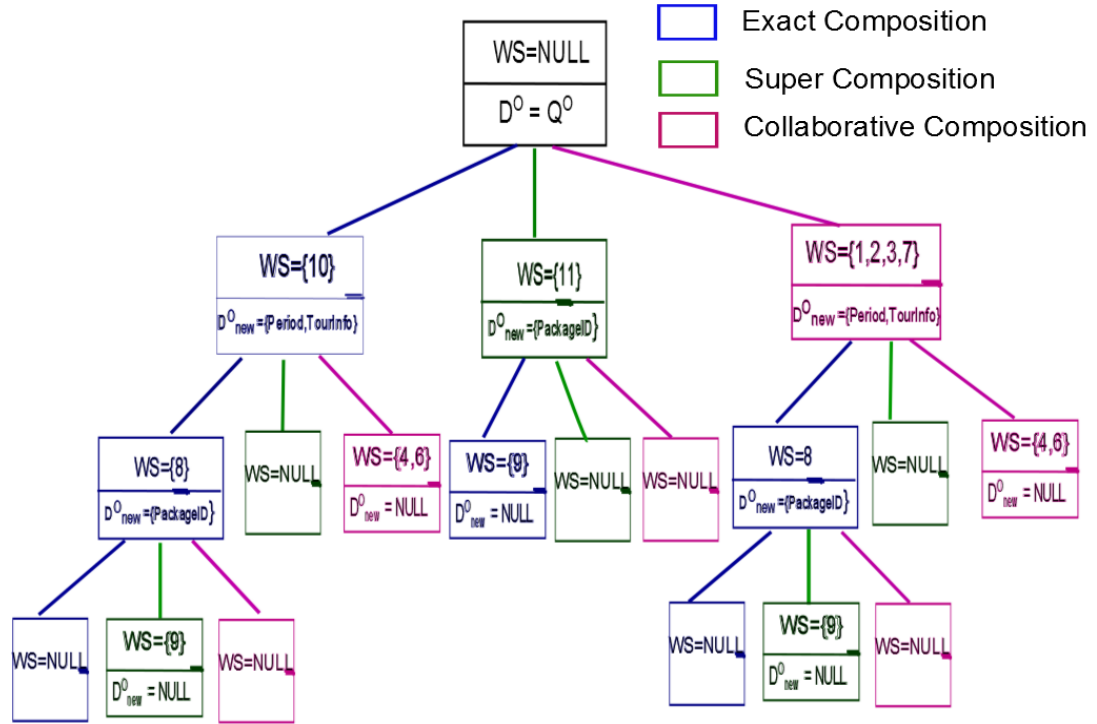


Figure 1.9: Example CST

### 1.3.4 List of Publications

The contributions discussed above have led to the following publications:

1. Lakshmi H. N., Hrushiksha Mohanty, A Preprocessing of service Registry: Based on I/O Parameter Similarity, published in proceedings of 11th International Conference on Distributed Computing and Internet Technology (ICDCIT 2015), Lecture Notes on Computer Science, Vol. 8956, 2015, pp 220-232.
2. Lakshmi H. N., Hrushiksha Mohanty, Clustering web services on Frequent Output Parameters for I/O Based service Search, published in proceedings of MIWAI 2014, Lecture Notes on Computer Science, Vol. 8875, pp 161-171.
3. Lakshmi.H.N, Hrushiksha Mohanty, Extended Service Registry to Support I/O Parameter-Based service Search, published in proceedings of 1st International Conference on Intelligent Computing, Communication and Devices, ICCD 2014, Advances in Intelligent Systems and Computing, Springer Publications, Volume 308

AISC, Issue VOLUME 1, 2015, Pp 145-155.

4. Lakshmi.H.N, Hrushiksha Mohanty, Utility of Composition Search Tree for Searching Optimal service Compositions , published in proceedings of Eighth International Conference on Data Mining and Warehousing (ICDMW-2014), Elsevier Science, ISBN: 9789351072515, 2014 , Pp 36- 42.
5. Lakshmi.H.N, Hrushiksha Mohanty, RDBMS for service repository and composition, published in IEEE Xplore, Proceedings of 4th International Conference on Advanced Computing, ICoAC 2012, Chennai, India, 13 to 15 December 2012.

# CHAPTER 2

## Literature Survey

### Abstract

In this chapter we review the research publications on various approaches proposed for the following: *web service search and composition*, emphasizing input/output based approaches, *clustering web services* and *QoS based service selection and ranking* with an emphasis on optimization based methods.

### 2.1 Introduction

Specifications for web services fall into two main categories viz. ontology based (OWL-S and WSMO) and keyword based(WSDL). A service of former category is built with domain ontology whereas of latter case keywords are used for specifying inputs and outputs. WSDL is popular and adopted by the industry due to its simplicity, while OWL-S (formerly DAML-S) and WSMO are well accepted by researchers as they offer much structured and detailed semantic mark-ups. The term web services is used for services described in WSDL and whereas semantic web services mean those described using either OWL-S or WSMO. A clear separation of both is necessary since the techniques, be it service search, service selection or service composition, for the two types can be quite different. Though, semantic web services are versatile in nature still keyword based service specification is popular for its fast response avoiding reference to domain ontology that could be at times large and complex enough for the purpose. Hence we have taken keyword based service specification and are interested in input/output parameter based service search and composition in UDDI based service registries.

The simplest approach to query a UDDI registry is the keyword-based information retrieval process of matching a query string against the textual description in the UDDI catalog and in the tModel. To address the limitation of keyword-based queries, more sophisticated service search approaches are available in the literature. When services satisfying a user query is not readily available in the registry service composition becomes inevitable. Manual composition of web services is time consuming, error-prone, generally hard and not scalable. Hence, many approaches proposed for web service composition aims to fully or partially automate the composition process.

Given a user query, web services to satisfy the given requirements are obtained following the steps below:

1. **Service discovery (service search):** Web services that match the users functional and non-functional requirements are located by searching in a service registry that holds information about registered web services. It is very likely that many candidate services may be found for the given requirements. While satisfying the basic functional requirements, these matching services may differ in their QoS attribute values, i.e. different levels of response time, availability, price, etc. As the number of services available are rapidly increasing, various approaches for faster service search are being proposed in the literature. Recently, clustering of web services has become a popular area of research, due to the potential benefits that can be achieved from clustering, like reducing the search space of a service search task, thereby making service search faster.
2. **Service selection:** This stage follows service search. The goal of this stage is to select a set of web services that best match the user specified requirements from the pool of candidate web services that the first stage returns.
3. **Service composition:** Service composition can be defined as creating a composite service, obtained by combining available web services. If no single web service in the registry can satisfy the functionality required by the user, there is

a need to combine existing services together in order to fulfill the user request, resulting in a service composition. Web service composition still is a highly complex task due to the dramatical increase in the number of services available during the recent years and also due to dynamic nature of the services.

In this chapter we survey the various approaches proposed for the various steps of user query processing as explained above. We first review the various approaches proposed for service search and composition in section 2.2. We mainly discuss input/output parameter based approaches, especially those based on graphs and relational models, since our proposal is motivated by these approaches. In section 2.3 that follows, we survey various approaches proposed for clustering web services to improve the performance of service search. Following is section 2.4 where we discuss the approaches for QoS based service selection and ranking. Conclusion is given in section 2.5.

## 2.2 Web services search and composition techniques

A web service,  $ws$ , has typically two sets of parameters, as set of inputs  $ws^I$  and set of outputs  $ws^O$ . Conventionally two services  $ws_i$  and  $ws_j$  are said to be composable iff  $ws_i^O = ws_j^I$ , i.e,  $ws_j$  receives all the required inputs from outputs  $ws_i$  has. Table 2.1 depicts example web services considered for service composition. Fig 2.1 gives an example of input/output parameter based service composition considering these services. In the example, when a user queries for a web service that takes  $\{PackageID\}$  as input and provides  $\{HotelName, FlightInfo, TourCost\}$  as output, then a composition of services *PackageDetails*, *DisplayTourInfo* and *TourCost* satisfies the query. A query that takes  $\{Date, City\}$  as input and expects  $\{TourCost\}$  as output can be satisfied by two compositions: *TourPackage*, *AgentPackage*, *TourCost* and *TourPackage*, *AgentPackage*, *TourReservation*.

Service Name	Input parameters	Output parameters
Package Details	PackageID	HotelName, FlightInfo
Display TourInfo	HotelName, FlightInfo	TourInfo
Tour Cost	TourInfo	TourCost
Tour Package	Date, City	PackageID, Period
Agent Package	PackageID, Period	TourInfo, Date
Tour Reservation	TourInfo, Date	TourCost

Table 2.1: Example web services

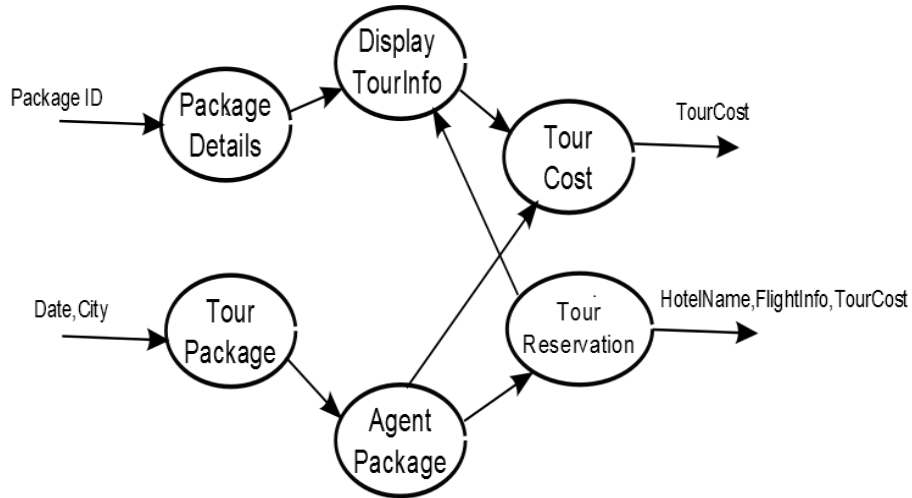


Figure 2.1: Parameter based composition example

We are interested in building an *extending service registry* capable of handling input/output parameter based service search and composition(automatic), along with the traditional keyword based search supported by service registries. Our work greatly depends on the input/output parameter match of services and hence in this section, we survey current efforts related to web services composition considering input/output parameters of web services. We begin by giving a brief classification of service composition approaches available in literature. Of the many approaches proposed for parameter based search and composition, we review two popular and active research approaches: graph based approaches and relational database approaches, which are closely related to our proposed approach.



### 2.2.1 Composition approaches

Web services search and composition is an active research topic although many approaches have been proposed in the literature. This is due to the complexity involved in the problem and a variety of solutions are still being proposed. Web services composition techniques can be categorised based on many aspects (Syu *et al.* (2012); Dustdar and Schreiner (2005)) as below:

- *Static vs dynamic composition:* Service composition can be classified as static and dynamic composition based on when service selection is scheduled. Composition can be done statically where the user builds an abstract model of the tasks that should be carried-out during the execution of the web service. The model is to be finalized before the composition planning starts. Web services matching each task is found and one of them is selected for execution of the composed web service. This may work fine as long as the component web services does not, or only rarely change.

The service environment is highly dynamic in that new services are added on a daily basis and constantly growing number of service providers. To enable service processes to adapt to environment changes, dynamic composition techniques have been proposed. Dynamic composition is achieved by creating the abstract model of tasks and selecting the atomic web services automatically without the interference of the user in the composition process.

- *Manual vs automatic composition:* Web service composition techniques can be categorized based on degree of user intervention in the composition process as manual or automatic. Manual composition done by human composers is a mature technique and already has been applied universally in the industry. Since manual composition relatively demands higher costs (Dustdar and Schreiner (2005)), automatic composition is becoming more popular and hence a larger proportion of research is dedicated to automatic web service composition, trying to eliminate the user intervention completely.

Web services are often described by the inputs, outputs, preconditions, and effects (IOPEs) and the non-functional properties given by QoS attributes. Following is a list of popular approaches that are based on input and output parameters of service:

- *AI Planning based approaches*: Approaches that treat the service composition as a planning problem start off from an initial state (inputs and preconditions) followed by a sequence of actions leading to a goal state (required outputs)(Rodriguez-Mier *et al.* (2011); Rao and Su (2005); Medjahed *et al.*). However, most of these proposals have some drawbacks: high complexity, high computational cost and inability to maximize the parallel execution of web services.
- *Graph based approaches*: Approaches, such as (Hashemian and Mavaddat (2006); Talantikite *et al.* (2009); Arpinar *et al.* (2004); Gekas and Fasli (2005)), view the service composition problem as a graph search problem, where services in the registry are visualized as a dependency graph and graph path finding algorithms are applied for finding compositions satisfying the user requirements. However, most of these approaches utilize complex dependency graphs that are not optimized to reduce service redundancy (equivalent services and equivalent combination of services).
- **RDBMS based approaches**: Recently, many researchers (Kwon *et al.* (2007); Lee *et al.* (2011); Zeng *et al.* (2010); Li *et al.* (2014a); Srivastava *et al.* (2006)) have utilized relational database technology for solving service composition problem. However, most of these approaches (Kwon *et al.* (2007); Lee *et al.* (2011); Zeng *et al.* (2010)) are constrained to usage of multiple joins. Also these algorithms construct chains of services for composition based on **exact matches of input/output parameters** to satisfy a given query.

We survey in detail the graph based composition approaches and RDBMS based approaches in the following subsections since they are closely related to our proposed approach.

### 2.2.2 Graph based service composition techniques

A great deal of research (Hashemian and Mavaddat (2006); Talantikite *et al.* (2009); Arpinar *et al.* (2004); Gekas and Fasli (2005)) has been done in recent years on graph based approaches for web services composition that consider input/output parameter match of the web services. Graph based approaches view web services as vertices and edges are drawn between composable services, thus reducing web service composition search into path finding problem between vertices in the graph.

Hashemian et al.(Hashemian and Mavaddat (2006)) work on a web service repository that contains information for a set of existing web services. They introduce dependency graphs to capture dependencies among the input and output of services in the repository. The nodes of the graph capture inputs and outputs of services, while its edges represent the input-output dependencies imposed by each web service. Composite services are built by applying a graph search algorithm. However, the dependency graph has some restrictions as it captures only one-to-one input-output dependencies;i.e. services that receive a single input and return a single output and hence is unable to represent the dependencies when multiple inputs/outputs are involved.

Arpinar et al.(Arpinar *et al.* (2004)) propose an Ontology driven web services composition platform. They present an approach which uses weighted graphs for web service composition. The weight of every edge is a function of QoS attributes and semantic similarity between service parameters. The composition technique aims to find an optimal composition of services considering QoS and semantic matching of parameters. They propose a modified Bellman-Ford shortest-path dynamic programming algorithm to find the shortest sequence from initial stage at node SI(a web service in the graph) to the termination node SF(a web service in the graph).

Gekas et al.(Gekas and Fasli (2005)) propose an approach considers web service in-

tegration as a graph search problem, where the search space consists of all the potential web service operations that can be part of a work-flow. The service registry is viewed as a hyper linked graph network consisting of web services linking to other web services. Heuristics regarding the connectivity structure of the repository and how  $\S$ tightly $\checkmark$  various types of web services are linked together are derived from the service repository. These heuristics are used in order to generate the  $\S$ Priority Queues $\checkmark$  used by the composition algorithm. The composition is done using a recursive depth-first algorithm, which starts from the initial state and tries to reach the goal state following the shortest route possible. The major drawback of this approach is that searching all possible compositions needs multiplying adjacency matrices  $N$  times, which can be very huge and time consuming as the number of service in the repository increase.

Talantikite et al(Talantikite *et al.* (2009)) propose to pre-compute and store a network of services that are linked by their input/output parameters.They use graph exploration algorithms and chaining algorithms to find a solution for the composition. Their approach utilizes backward chaining and depth-first search algorithms to find sub-graphs that contain services to accomplish the requested task. At the end of the network exploration, several composition plans can be found which are further categorized as : Simple composition, Serial composition, Independent parallel composition and Dependent parallel composition.

Pablo et.al.(Rodriguez-Mier *et al.* (2011)) present an  $A^*$  algorithm for matching semantic input-output message structure for web service composition. A service dependency graph is dynamically generated for a given request from services in a repository and a minimal composition satisfying the request is found using  $A^*$  search algorithm.

We compare the graph based approaches discussed above in table2.2.

Approach	Algorithm used for service composition search	Weighted Graph	Limitation
Hashemian et al (Hashemian and Mavaddat (2006))	Breadth first search (BFS)	No	Unable to represent the dependencies when multiple inputs/outputs are involved.
Arpinar et al (Arpinar <i>et al.</i> (2004))	Modified Bellman-Ford shortest-path algorithm	Yes	Not Scalable (Performance of search decreases with increase in number of services )
Gekas et al (Gekas and Fasli (2005))	Recursive Depth first search (DFS)	No	Not Scalable (Searching all possible compositions needs multiplying adjacency matrices N times )
Talantikite et al (Talantikite <i>et al.</i> (2009))	Backward chaining and Depth first search (DFS)	No	Not Scalable (Performance of search decreases with increase in number of services )
Pablo et al (Rodriguez-Mier <i>et al.</i> (2011))	A* algorithm	No	Graph generation is done for a user query, hence not efficient in terms of query response time.

Table 2.2: Size of Graph generated for WSC data set

Most of the approaches discussed above are not scalable due to the size of graph involved. To further understand the complexity of the service composition problem, we analyze the size of the graph that is generated using the WSC data set (Blake *et al.* (2006)). WS-Challenge(WSC) data set is a benchmark data set which consists of web services with a complex structure. Each web service in WSC data sets has multiple input parameters and output parameters. The data set "composition-50-32" means as follows: 50 denotes the number of web services composition and 32 means that a web service has 32-36 input and output parameters. The number of web services in the "composition-20-32" and "composition-50-32" are 1000. Table2.3 shows the number of edges and the number of paths generated by WSC data set.

It can be seen from Table2.3 that the number of paths generated for "composition-

Data Set	No of Edges	No of Paths
composition-20-32	11,024	930,394
composition-50-32	58,270	29,363,370

Table 2.3: Size of Graph generated for WSC data set

50-32" data set are more than 29 million, showing the complexity of the graph involved in a service registry of just 1000 web services. Most of the graph based approaches for service composition create the graph at the time of composition which incurs substantial overhead and use in-memory algorithms for web services composition search. The scalability of in-memory approach is limited by size of physical memory of the system, thus making this kind of algorithms non-scalable. **We hence propose to use Object Relational Database, for its huge success in data management, as a scalable repository for storing web services in UDDI framework.**

### 2.2.3 RDBMS based service composition techniques

Recently, many researchers have utilized relational database technology for solving service composition problem. Lee et al. (Kwon *et al.* (2007); Lee *et al.* (2011)) proposed a scalable and efficient web service composition system based on a relation database system. They pre-compute all possible web service compositions and store it in tables to be used later for web service composition search. Pre-computing web service composition is done by applying multiple joins on the tables maintained. They also propose a web service composition search algorithm called PSR(Pre-computing solutions for WSC in a RDBMS) that uses simple SQL statements. PSR system supports web services having single input and output parameters and can handle web services having multiple input and output parameters.

C. Zheng et al. (Zeng *et al.* (2010)) put forward a new storage strategy for web services which can be flexibly extended in relational database. They present a matching algorithm SMA between web services of multiple input/output parameters, which

considers the semantic similarity of concepts in parameters based on WordNet. The matching relationship between different web services based on their input/output parameters are calculated and stored into a One-way Matching Table(OMT). OMT is then viewed as a weighted directed graph where each node denotes a web service and each edge denotes the semantic matching degree between two web services. Thus, the service composition problem is simplified to find all reachable paths of two nodes in the graph. They propose a Fast-EP algorithm to reduce the time taken by multiple-joins of the table to find the service composition.

Jing Li et.al (Li *et al.* (2014a)) propose a system called FSIDB (Full Solution Indexing using Database) which uses a relational-database approach for automatic service composition. All possible service combinations are generated as paths and the QoS attribute values of the combinations are also pre-calculated before the search process and stored in a relational database. On receiving a user query the system uses a single SQL query to find a composition satisfying the user requirements.

Utkarsh (Srivastava *et al.* (2006)) propose the development of a web service Management System (WSMS): a general-purpose system that enables clients to query multiple web services simultaneously in a transparent and integrated fashion. They build virtual tables for input/output parameters of web services to manage service interfaces, and uses multi-thread pipeline executive mechanism to improve the efficiency of web services search, so the service composition problem is transformed into query optimization in database. Query processing over web services is visualized as a workflow or pipeline: input data is fed to the WSMS, and the WSMS processes this data through a sequence of web services. The output of one web service is returned to the WSMS and then serves as input to the next web service in the pipeline, finally producing the query results.

Daniele et.al (Braga *et al.* (2008)) propose a formal model for the optimization and

the execution of multi-domain queries over heterogeneous web services. The model uses query plans that schedule the invocations of web services and the composition of their inputs and outputs. The main operations of the query plans are joins between web service results, whose execution depend on the join strategies. Given a query over a set of web services their approach finds a query plan that minimizes the expected execution cost according to a given metric in order to obtain the best k answers.

Though there have been some techniques that use database technology for service matching and compositions, still those are constrained to usage of multiple joins. Also the existing algorithms construct chains of services based on **exact matches of input/output parameters** to satisfy a given query. However, the making of such a service composition chain may fail at a point when the output parameters of a preceding service ( $ws_P^O$ ) does not match exactly with the input parameters of a succeeding service ( $ws_S^I$ ). **We propose an approach in chapter 3, to alleviate this problem by making match criteria flexible. In addition to exact match we allow partial as well as super match for conditions  $ws_i^O \subset ws_j^I$  and  $ws_i^O \supset ws_j^I$  respectively** (Lakshmi and Mohanty (2012)). Further, to avoid the need for multiple joins and to speed up querying, **we propose to use multi-valued attributes for storing input and output parameters in our database design**, and hence use an Object Relational database for our proposed **extended service registry**. Based on this concept we have **explored the feasibility of input/output based web service search and composition in our proposed ESR system**, to support varying requirements of the consumer.

## 2.3 Clustering web services

Research in clustering web services has recently gained much attention due to the popularity of web services and the potential benefits that can be achieved from clustering web services like reducing the search space of a service search task. There are a number of approaches proposed in recent years for non-semantic web service search (Elgazzar



*et al.* (2010); Liu and Wong (2009); Nayak and Lee (2007); Liu *et al.* (2010)). In this section we summarize various clustering approaches for non-semantic web services.

Xin Dong *et al.* (Dong *et al.* (2004)) develop a clustering based web service search engine, Woogle. Their search approach consists of two main phases: first, the search engine returns a list of web services that match a given user query, specified as a set of keywords. Second, their tool extracts a set of semantic concepts, by exploiting the co-occurrence of terms in web service inputs and outputs of operations, to cluster terms into meaningful concepts by applying an agglomerative clustering algorithm. This makes it possible to combine the original keywords with the extracted concepts and compare two services on a keyword and concepts level to improve precision and recall. This approach leads to significantly better results than a plain keyword-based search.

Richi Nayak and Bryan Lee (Nayak and Lee (2007)) propose "Semantic web services clustering (SWSC)" as an extension to the existing UDDI registry. They utilize OWL-S ontology and WordNet lexicon to enhance the WSDL description of web services with semantics. The approach attempts to handle the service search problem by suggesting to the current user with other related search terms based on what other users had used in similar queries by applying clustering techniques. web services are clustered based on search sessions instead of individual queries.

For the calculation of non-semantic similarity between web services, WSDL-based approaches are the most representative work (Elgazzar *et al.* (2010); Liu and Wong (2009)). Khalid Elgazzar *et al.* (Elgazzar *et al.* (2010)) propose a technique for clustering WSDL documents into functionally similar web service groups to improvise web service search in service search engines. They use the Quality Threshold (QT) clustering algorithm to cluster web services based on similarity of 5 features in WSDL documents which include - WSDL content, WSDL types, WSDL messages, WSDL ports and the web service name.

Liu and Wong (Liu and Wong (2009)) propose to extract 4 features, i.e., content, context, host name, and service name, from the WSDL document to cluster web services. They take the process of clustering as the preprocessor to search, hoping to help in building a search engine to crawl and cluster non-semantic web services.

The approaches discussed above employ clustering during the service search process, to cluster search results for a given user query. Mostly, the approach (Ma *et al.* (2008)) of query processing follows two basic steps - reducing the domain of search by service domain e.g. hotel, transport etc. and then grouping the services based on query requirements. As these are done during processing of a query, it becomes expensive in terms of time. **Instead, in chapter 4, we propose an approach that does a book keeping of services following an order so that the pre-processing helps to reduce search time.** The order here is formed as patterns of output parameters i.e co-occurrence of output parameters. Though, time is spent in pre-processing still, the gain in the proposed scheme is better than the time due to the approach followed earlier (Ma *et al.* (2008)). **Our approach is useful when we are looking for a web service with a desired output parameter Pattern.**

## 2.4 QoS Based service selection and ranking

Web service composition (WSC) offers a range of solutions for user query when web services satisfying the user requirements are not readily available in the service registry, by composing a set of already existing web services. One of the critical challenges in web service search and composition is the selection of appropriate services that best match the user requirements, either to be executed independently or to be bound to the service composition, from the pool of matching web services in the service registry. In this section we provide a brief survey of the existing service selection approaches.

Of the many service selection methods proposed in web service literature, we review here the approaches that model selection as an optimization problem. Optimization can be performed at two levels: *Local Optimization*, for an individual web service selection and *Global Optimization*, for a given business process.

Chia Lin et.al (Lin *et al.* (2011)) propose a QoS-based service selection (RQSS) algorithm to discover feasible web services based on functionalities and QoS criteria of user requirements. The QoS constraints are classified as relaxable and non-relaxable constraints and the approach not only discovers web services fulfilling the functional requirements and non-functional QoS constraints, but also recommends solutions which could satisfy the non-relaxable QoS constraints by relaxing the relaxable QoS constraints.

Karim et.al (Karim *et al.* (2011)) propose to use an enhanced PROMETHEE model for QoS-based web service selection. They take into account the QoS interdependency by using Analytical Network Process (ANP) to calculate the priority associated with each QoS criterion. In their original PROMETHEE model they do not consider user's QoS requirement due to which the model may end up in listing services that optimizes the overall QoS criteria but fail to satisfy the user requirements. Hence they enhance their approach to rank the web services listed in the search to assess how well a service satisfies the user requirement.

Huang (Huang *et al.* (2009)) applies a weighted sum model (WSM) to help service requesters evaluate services numerically. QoS-based optimization of service composition is then transformed into an integer programming problem by deriving the objective functions of constituent workflow patterns. User needs to provide a workflow of the service composition and the approach searches for services that best matches the given

workflow and the QoS constraints.

Ronald et.al (Yager *et al.* (2011)) propose a simple but effective selection approach for finding the most suitable web services fitting user's requirements. The user needs to identify the QoS criteria of interest, provide ranking of those criteria, from which constraint satisfaction functions are constructed. They use lexicographic method for multi criteria decision making : to order the QoS criteria according to the preference provided by the user, this ordering ensures that some QoS criteria must be satisfied before considering the others.

Mohammad et.al (Alrifai *et al.* (2012)) propose a hybrid solution that combines global optimization with local selection techniques, visualizing the problem as an instance of multi-dimensional multiple choice knapsack problem(MMKP). The approach selects web services for a given composition request from a collection of candidate services satisfying the specified QoS constraints. They use MIP(Mixed Integer Programming) to find the optimal decomposition of global QoS constraints into local constraints and then distributed local selection is applied to find the best web services satisfying these local Constraints.

Chen Ding (Mobedpour and Ding (2013)) propose a selection model capable of handling both exact and fuzzy requirements. The model returns two categories of matching web services: super-exact and partial matches, which are ranked based on relaxation orders and then preference orders of the QoS attributes provided by the user, using MIP as the base algorithm. Symbolic dynamic clustering algorithm(SCLUST) is used to cluster services into 3 groups: good, medium, and poor, based on the values of QoS attributes of the web services.

Most of the current service selection proposals apply using a weighted sum model

(WSM) as an evaluation method for selection of services with the same functionality. This is represented as -

$$Score(WS) = \sum (q'_i * w_i) \quad (2.1)$$

where  $q'_i$  is a normalized QoS attribute value and  $w_i$  is the weight given to the QoS attribute. Such methods require users to express their preference over different (and sometimes conflicting) quality attributes as numeric weights. The objective function assigns a scalar value to each service based on the QoS attribute values and the weights given by the user. The service that has the highest value for the objective function will be selected and returned to the user. Such optimization techniques are unable to model user preferences precisely.

We hence propose a **service selection approach**, in chapter 5, that selects the most appropriate web services from the pool of matching services based on a **bi-level model that considers both the functional and non-functional requirements for service selection**. The functional requirements are provided by the user as a set of input parameters provided for and output parameters desired from the web service. The user also provides a set of desired QoS values and the order of their preference for selection. Our model for service selection lists the web services that best matches the functional and non-functional requirements and constraints specified by the user. Experiments were conducted to compare our approach with that proposed by Chen's (Mobedpour and Ding (2013)) approach. Various sets of queries were fed for both the approaches and the results were analyzed on the quality of services selected and the execution time taken by both approaches. From the results obtained we can infer that our approach performs better and returns quality web services as compared with Chen's approach.

## 2.5 Conclusion

We provide a brief definition of each stage involved in query processing in a service repository: service discovery, service selection and service composition. A review of the recent works in each of these stages is provided. The survey is restricted to input/output parameter based methods since our proposal is a parameter based approach. Also we confine our survey in service composition to graph based and relational database based approaches since these are closely related to our proposed approach. Recent works on QoS based service selection and ranking are also discussed, giving emphasis to optimization based methods. In all these sections, we have compared our proposed work with available literature and established the need and applicability of our work.

In the next chapter we propose an extended service registry, an object relational database approach for storing web service information in the registry. The objective of such a registry is to provide additional functionalities like input/output parameter based web service search, along with the traditional keyword based service search. The registry also provides automatic composition of services for satisfying a given user query, when there are no services readily available in the registry matching the given requirements.

# CHAPTER 3

## Extended Service Registry

### Abstract

Service registries and web service engines are the main approaches for discovering web services. Current service directories are mainly based on Universal Description, Discovery and Integration (UDDI), which is an industry standard for service registries, developed to solve the web service search problem. However, UDDI offers limited search functionalities due to the limited information that is available and searchable in UDDI entries and do not provide any support for complex searches like input/output parameter based search and automatic composition of web services. We propose a new system called *extended service registry*(ESR)<sup>1</sup>capable of offering powerful and efficient web service search and composition operations, by extending UDDI with meta information<sup>2</sup>. Implementation of ESR uses object relational database technology. The experimental results demonstrate the efficiency of service search in our ESR and the variety of user queries supported.

### 3.1 Introduction

Research on web service search and composition has become increasingly important in recent years due to the growing number of web services over the Internet and the challenge of automating the process of composition. Since there is a large growth in number of available web services and possible interactions among them are huge, searching for desired set of services to satisfy a user query becomes very difficult. This in-turn means service search and composition problem has become increasingly sophisticated

---

<sup>1</sup>Lakshmi.H.N, Hrushiksha Mohanty;Extended Service Registry to Support I/O parameter-Based Service Search,Proceedings of 1st International Conference on Intelligent Computing, Communication and Devices, ICCD 2014, Springer Publications,Volume 308 AISC, Issue Volume 1, 2015, Pages 145-155.

<sup>2</sup>Lakshmi.H.N, Hrushiksha Mohanty;RDBMS for service repository and composition, Proceedings of 4th International Conference on Advanced Computing, ICoAC 2012, published in IEEE Xplore.

and complicated for finding a solution.

Various approaches can be used for service search, such as, searching in UDDI, Web and service portals, as discussed previously in the chapter 2. **We take up the issue of web service searching and composition at service registries** for its practicality in business world as providers would like to post their services centrally, as searching there is less time consuming than searching on world wide web.

Universal Description, Discovery and Integration (UDDI) (Carolgeyer (2013)) is an industry standard for service registries, developed to solve the web service search problem. Basically, UDDI supports service search by name, location, business, bindings or tModels, and binds two services based on composability of their protocols (citeuddi). Traditional UDDI based service search lacks the ability to recognize all the features described in WSDL files. Hence the search API(Application Program Interface) is limited by the kind of information that is available and searchable in UDDI entries and do not provide any support for complex searches like input/output parameter based search and automatic composition of web services. Also, our experiments show that average response time of current UDDI implementations increase with a substantial increase in number of services registered.

The above shortcomings of UDDI motivated us to **build an extended service registry(ESR) system capable of offering powerful and efficient web service search and composition operations**, by extending UDDI with meta information. We propose the use of Object Relational Database as repository of web services. Information about the web services, extracted from their WSDLs, are stored in tables and relational algebraic operators are used for service search and composition.

A web service,  $ws$ , has typically two sets of parameters - set of inputs  $ws^I$  and set



of outputs  $ws^O$ . By service composition, we mean making of a new service(that does not exist on its own) from existing services registered in UDDI. Conventionally two services  $ws_i$  and  $ws_j$  are said to be composable iff  $ws_i^O = ws_j^I$ , i.e,  $ws_j$  receives all the required inputs from outputs  $ws_i$  has. A service composition is formed by constructing a chain of such composable services.

Recently, many researchers have utilized relational database techniques to solve the service composition problem (Kwon *et al.* (2007); Lee *et al.* (2011); Zeng *et al.* (2010)) as discussed previously in chapter 2. These relational techniques can be applied to a large number of web services in that their scalability is not limited by size of physical memory. However the **current approaches use multiple joins** (Kwon *et al.* (2007); Lee *et al.* (2011); Zeng *et al.* (2010)) on tables to pre-compute all possible web service compositions, which may be unnecessary. Also most of the existing algorithms (Kwon *et al.* (2007); Lee *et al.* (2011); Zeng *et al.* (2010)) construct chains of services based on **exact matches of input/output parameters** to satisfy a given query, as explained earlier. However, this approach fails when the available services satisfy only a part of the input/output parameters in the given query.

When a user is looking for a web service for given input and desired output parameters and there is no single web service in the service registry satisfying the request, service Composition becomes necessary. However, the making of a service composition chain may fail at a point when the output parameters of a preceding service ( $ws_P^O$ ) does not match exactly with the input parameters of a succeeding service( $ws_S^I$ ). **We propose an approach to alleviate this problem by making match criteria flexible. In addition to Exact match we allow Partial as well as Super match for conditions  $ws_i^O \subset ws_j^I$  and  $ws_i^O \supset ws_j^I$  respectively** (Lakshmi and Mohanty (2012)). Further, to avoid the need for multiple joins and to speed up querying, **we propose to use multi-valued attributes for storing input and output parameters in our database design,** and hence use an Object Relational database for our proposed *ESR*. Based on this con-

cept we have **explored the feasibility of input/output based web service Search and Composition in our proposed ESR system**, to support varying requirements of the consumer.

In this chapter we first describe the types of service matches, extended from the classical definition of service match in section 3.2. In section 3.3 that follows, we propose an *extended service registry* using ORDBMS approach. The database schema is explained in detail and an argument on the need for ORDBMS is also given in this section. Following is section 3.4 that provides a discussion on the performance and scalability of our *extended service registry* in comparison with the standard UDDI service registry. Conclusion is given in section 3.5.

## 3.2 Types of service matches based on input/ output parameters

A web service,  $ws$ , has typically two sets of parameters - set of input parameters  $ws^I$  and set of output parameters  $ws^O$ . When a user searches for a service based on its commitments(inputs) and expectations(outputs), there may be many matching services in the registry. Most of the existing algorithms for service search and composition consider services based on Exact matches of input/output parameters to satisfy a given query. However, for service composition the construction of a chain of services fails at a point when the output parameters of a preceding service ( $ws_P^O$ ) does not match exactly with the input parameters of a succeeding service ( $ws_S^I$ ). We propose an approach to alleviate this problem by making service match criteria flexible.

The classical definition of service matching, i.e one-to-one and onto mapping between input and output parameters of matching services, is extended to give rise to three types of service matches: *exact* match ( $O^P = I^S$ ), *super* match ( $O^P \supset I^S$ ) and *partial*

match ( $O^P \subset I^S$ ). Services with partial match collaborate to satisfy the input/output parameters of the given query. Following is a brief description of each of these type of service matches :

1. **Exact match** : A web service  $ws_i$  is an exact match of web service  $ws_j$  if the input/output parameters of  $ws_i$  exactly matches all the input/output parameters of  $ws_j$ , i.e.,

$$ws_i^O = ws_j^O \text{ or } ws_i^I = ws_j^I, i \neq j.$$

2. **Partial match** : A web service  $ws_i$  is a partial match of web service  $ws_j$  if the input/output parameters of  $ws_i$  partially matches the input/output parameters of  $ws_j$ , i.e.,

$$ws_i^O \subset ws_j^O \text{ or } ws_i^I \subset ws_j^I, i \neq j.$$

3. **Super match** : A web service  $ws_i$  is a super match of web service  $ws_j$  if the input/output parameters of  $ws_i$  is a superset of the input/output parameters of  $ws_j$ , i.e.,

$$ws_i^O \supset ws_j^O \text{ or } ws_i^I \supset ws_j^I, i \neq j.$$

### 3.3 Object Relational Database for service registry

In this section we first give an overview of how web services are registered in UDDI. We then argue for the need of Object Relational Database for storing web service information in a registry. We then propose an Object Relational schema for our ESR.

#### 3.3.1 WSDL and UDDI Relationship

The Web Services Description Language (WSDL) (WSDL (2001)) is a XML language for describing web services. It contains service interface defined as a set of operations

and messages, their protocol bindings, and the deployment details. UDDI (Version 3) (Carolgeyer (2013)) offers an industry specification that is used for building flexible, inter-operable XML web services registries useful in private as well as public deployments. It offers clients and implementers a comprehensive and complete blueprint of description and discovery foundation for a diverse set of web services architectures.

A UDDI information model is composed of four primary data structures (Carolgeyer (2013)):

1. *businessEntity* : used to describe a business or service provider that typically provides web services.
2. *businessService* : used to represent business descriptions for a web service. Describes a collection of related web services offered by a businessEntity.
3. *bindingTemplate* : contains the technical information associated to a particular service. Each bindingTemplate describes an instance of a web service offered at a particular network address, typically given in the form of a URL. The bindingTemplate also describes the type of web service being offered using references to tModels, application-specific parameters, and settings.
4. *tModel* : used to define the technical specification for a web service, such as a web service type, a protocol used by web services, or a category system.

A *businessEntity* can contain one or more *businessServices*. A *businessService* can have several *bindingTemplates* and each *bindingTemplate* contains a reference to one or more tModels.

WSDL complements the UDDI standard by providing a uniform way of describing the abstract interface and protocol bindings of arbitrary network services. WSDL document contains six major elements that defines web services(WSDL (2001)) :

1. *types* : provides data type definitions used to describe the messages exchanged.
2. *message* : provides an abstract definition of the data being transmitted and consists of logical parts.
3. *portType* : defines a set of abstract operations each referring to an input message and output message.
4. *binding* : specifies concrete protocol and data format specifications for the operations and messages defined by a particular portType.
5. *port* : specifies an address for a binding, defining a single communication endpoint.
6. *service* : aggregates a set of related ports.

*Publishing and finding WSDL descriptions in UDDI:*

Here we describe the process for publishing and searching a service, described by a WSDL document. The WSDL document of a service describes instances of the service using a WSDL service element. Each service element in a WSDL document is used to publish a UDDI businessService. The service interface described using WSDL portType and binding is published as a UDDI tModel before publishing a businessService. The various elements of WSDL are mapped to UDDI data structures as follows:

- WSDL portType element is mapped to UDDI tModel.
- WSDL binding element is mapped to UDDI tModel.
- WSDL port element is mapped to UDDI bindingTemplate.
- WSDL service element is mapped to UDDI businessService.

### **3.3.2 Object relational database as service registry**

On contemplating the relationship between WSDL and UDDI, we observe that there are many sub-elements in WSDL that are not mapped to UDDI types like input message,

output message, message parts, etc. Although these details can be added in tModel of the service, there are no supporting search functionalities provided in UDDI standards. This leads to the limited and poor search functionalities supported by UDDI. We hence propose to include these details in an ESR, enabling the registry to support additional search functionalities.

Each operation in a service has an input and output message, each of which in turn may have one or more parameters. For example a *HotelBooking* service may take  $\{Period, City\}$  as input parameters and provide  $\{HotelName, HotelCost\}$  as output parameters. Since current UDDI implementations store the UDDI data structures in a Relational Database, a first thought would be to include the service parameters in a Relational database. A normalized Relational Database solution to this requires that we store input and output parameters of each operation across multiple rows. Let us consider a simple example to explain it further - table 3.1 lists a few example web services. A relational database solution to this is to assign each parameter a unique parameter id(as in Pars Table) and list the various input and output parameters of web services across multiple rows as shown in table 3.2.

Service No	Service Name	Input parameters	Output parameters
ws1	HotelBooking	Period, City	HotelName, HotelCost
ws2	AirlineReservation	Date, City	FlightInfo, FlightCost
ws3	TaxiInfo	Date, City	CarType, TaxiCost
ws4	DisplayTourInfo	HotelName, FlightInfo, CarType	TourInfo
ws5	TaxiReservation	CarType, Date, City	TaxiCost

Table 3.1: **Example web services**

There are several advantages of normalization as -

- Provides indexing.
- Minimizes modification anomalies.
- Saves space.

- Enforces referential integrity.
- Provides High Performance.

Par ID	Par Name
1	Period
2	City
3	Date
4	HotelName
5	FlightInfo
6	CarType
7	TourInfo
8	TaxiCost
9	HotelCost
10	FlightCost

(a) Pars Table

WS ID	InParsId
ws1	1
ws1	2
ws2	2
ws2	3
ws3	2
ws3	3
ws4	4
ws4	5
ws4	6
ws5	2
ws5	3
ws5	6

(b) WSInput Table

WS ID	OutParsId
ws1	4
ws1	9
ws2	5
ws2	10
ws3	6
ws3	8
ws4	7
ws5	8

(c) WSOuput Table

Table 3.2: Relational Schema for web services

Normalizing the tables in 3.2 further, we get tables as shown in 3.3.

WSIP	WS ID	InParsId
1	ws1	1
2	ws1	2
3	ws2	2
4	ws2	3
5	ws3	2
6	ws3	3
7	ws4	4
8	ws4	5
9	ws4	6
10	ws5	2
11	ws5	3
12	ws5	6

(a) WSInput Table

WSOP	WS ID	OutParsId
1	ws1	4
2	ws1	9
3	ws2	5
4	ws2	10
5	ws3	6
6	ws3	8
7	ws4	7
8	ws5	8

(b) WSOuput Table

Table 3.3: Relational Tables after further normalization

One reason to carefully review the usage of normalization in a database design is the database's intended use. There are certain scenarios where the benefits of database

normalization are outweighed by its costs. Two of these scenarios are described below

-

1. Immutable data and append-only Scenarios - Database normalization may be unnecessary in situations where we are storing immutable data such as financial transactions or a particular day's price list.
2. When multiple joins are needed to produce a commonly accessed view - The biggest problem with normalization is that you end up with multiple tables representing what is conceptually a single item. With such design, it takes multiple SQL join operations to access and display the information about a single item. This implies reduced database performance. To make a long story short, a normalized database in such a scenario requires much more CPU, memory, and I/O to process database queries. A normalized database must locate the requested tables and then join the data from the tables to either get the requested information or to process the desired data.

Our database schema fits the second scenario. Consider the type of queries that we plan to use for web service composition -

- List all web services that matches(exact, partial and super) a given set of input parameters.
- List all web services that matches(exact, partial and super) a given set of output parameters.
- List all web services that are composable with a given set of output parameters.

Clearly the tables in 3.3 would require multiple joins to support these types of queries. Also the number of tuples in these tables increases drastically when the services have more input/output parameters, thereby generating more tuples in table joins, hence decreasing the database performance. Thus, we require a strategy to store these



multi-valued input and output parameters in the registry.

Object-relational database(ORDB) technology has emerged as a way of enhancing object-oriented features in relational database systems. In a relational model, multi-valued attributes are not allowed in the first normalization form. The solution to the problem is that each multiple-valued attribute is handled by forming a new table or distributed across multiple rows of the same table. To retrieve the data back in such a storage design, one has to do multiple joins across the tables (or the table, if stored as multiple rows in the same table). To avoid the need for multiple joins and to speed up the query, we propose to use multi-valued attributes for storing input and output parameters in our database design, and hence use an Object relational database for our proposed ESR. ORDBMs allow multi-valued attributes to be created in a database by using *collections type* or *nested tables*. Advantage of this approach is that it supports querying for services efficiently and also supports complex queries involving input and output parameters.

### **3.3.3 Registry Meta Information**

In this section we shall describe the object relational schema for storing web services in the registry. We first created a ESR to store the basic details of the web services registered in UDDI, by extracting the sub-elements of their WSDLs. Experiments were conducted to study the performance and scalability of ESR as explained in section 3.4.

Further to enhance the quality and performance of service search we proposed approaches for clustering service parameters on their similarity and to cluster web services in the registry on their frequent output parameter patterns, which is discussed in detail in chapter 4. The schema includes tables generated by these clustering approaches as well.

Our ESR supports two types of web service Search :

1. Name based web service Search.
2. Input and output parameter based service Search.

Both these searches generates a list of matching services, of which the best matching services are selected based on the QoS values specified in the user query. These search techniques use user query tables and dynamically generates few tables and views that stores the services matched and selected for the given user specifications. Tables and views supporting these searches are also explained in this section.

Finally, we propose a technique to generate various compositions that satisfy the given query requirements when the search fails to find a service matching the given query. This is done by constructing a *composition search tree*, as explained later in chapter 5. Tables to support the CST construction are also generated dynamically on querying and stored in the schema.

The relationship between the various tables in the proposed schema is depicted in the ER diagram in Fig 3.1. The various tables in the schema are categorized as below, based on the purpose of usage and explained further :

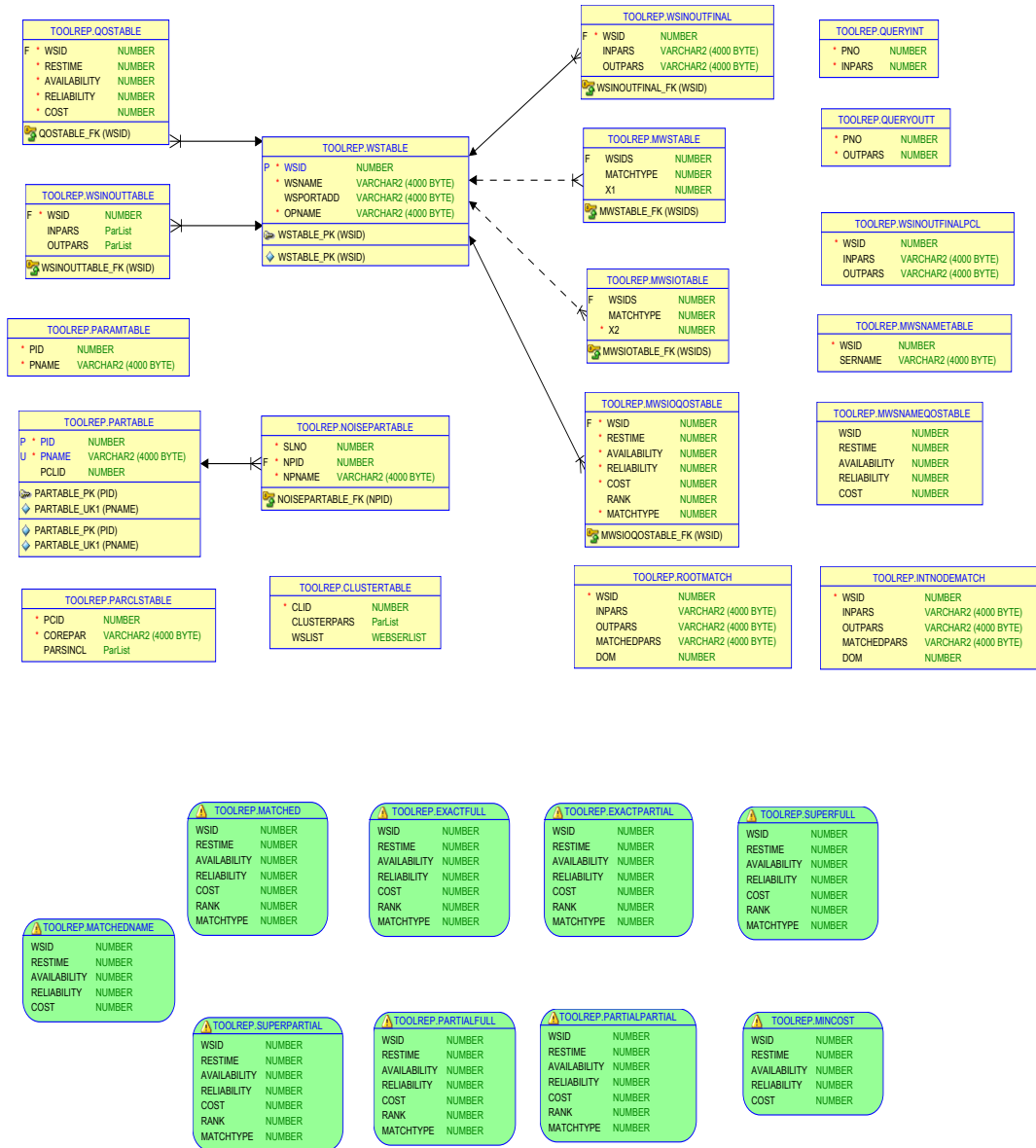


Figure 3.1: ER Diagram for Extended Service Registry

### 3.3.3.1 Registered services details

Following are the tables that store the basic details of all the web services registered in the registry. These information are extracted from the WSDLs of the respective services and stored in the respective tables as designed.

1. Each web service is given a unique ID (WSID) and stored with its name (WS-

Name), port address (WSPortAdd) and operation name (OPName) in a *Web Service Table (WSTable)*.

$$WSTable : \{WSID, WSName, WSPortAdd, OPName\}$$

2. The *Parameters Table (ParamTable)* contains all the parameters, that take part as either input or output in any of the web services in the registry, with their names in (PName). Each parameter is given a unique ID, (PID).

$$ParamTable : \{PID, PName\}$$

3. The *Web Service Input Output Table (WSInOutTable)* lists all web services in the registry with their respective input parameters (InPars) and output parameters (OutPars). InPars and OutPars are a collection of parameter Ids (Type - ParList) and are stored as nested tables.

$$WSInOutTable : \{WSID, InPars, OutPars\}$$

4. The *QoS Table (QoSTable)* contains the various quality of service values like response time, availability, reliability and cost for all web services registered in the registry.

$$QoSTable : \{WSID, ResTime, Availability, Reliability, Cost\}$$

### 3.3.3.2 Parameter clusters

The tables below are generated on clustering web service input and output parameters based on their similarity, as explained later in Chapter 4.

1. The *Parameter Clusters Table (ParClsTable)* stores the details of the clusters generated from the parameter clustering algorithm. Each cluster is given an unique Id (PCId), has a *core parameter* that acts as the cluster center (CorePar) and lists all the parameters in that cluster (ParsInCl). ParsInCl are a collection of parameter Ids (Type - ParList) and are stored as nested tables.

$$ParClsTable : \{PCId, CorePar, ParsInCl\}$$

2. The information as to which cluster a parameter belongs to is stored in a *Parameter Table(ParTable)*. Along with parameter Id and parameter Name for each parameter, the parameter Cluster Id of the cluster that it belongs to (PCId) is stored in this table.

$$ParTable : \{PIId, PName, PCId\}$$

3. The unclustered parameters in the registry are stored in a *Noise Table(NoiseTable)* with its parameter Id(NPId) and name(NPName).

$$NoiseTable : \{SlNo, NPId, NPName\}$$

### 3.3.3.3 Web service clusters

The *frequent output parameter pattern based clustering* algorithm, explained later in chapter 4, generates a *table of clusters(ClusterTable)* that stores the details of the web service clusters generated. Each web service cluster is given an unique Id(CId), has the list of output parameters supported by all the web service in the cluster(ClusterPars) and the list of web services in the cluster(WSList). ClusterPars are a collection of parameter Ids (Type - ParList) and are stored as nested tables. WSList is also a collection of WSIds of web services (Type - WebSerList), stored as nested tables.

$$ClusterTable : \{CId, ClusterPars, WSList\}$$

### 3.3.3.4 User query

The tables generated from the user Query is as follows :

1. The *Query input table(QueryInT)* stores the list of input parameters(InPars) specified in the user query along with its parameter Ids.

$$QueryInT : \{PNo, InPars\}$$

2. The *Query output table*(*QueryOutT*) stores the list of output parameters(*OutPars*) specified in the user query along with its parameter Ids.

$$QueryOutT : \{PNo, OutPars\}$$

### 3.3.3.5 Keyword based service search

The name based Search of services, searches for web services matching the given keyword and then selects the best matching services based on the the QoS values as specified in the user query. The search technique is explained further in Chapter 4. Following are the tables generated during this process.

1. Web services whose name matches with the given keyword are first listed in a *Matched Web Services Name Table*(*MWSNameTable*) as below :

$$MWSNameTable : \{WSId, SerName\}$$

2. The *MatchedName* view extracts the QoS values for the matched services in the *MWSNameTable*.

$$MatchedName : \{WSID, ResTime, Availability, Reliability, Cost\}$$

3. Finally, The best matches for the QoS values specified in the User Query are selected and stored in *Matched Web Services Name QoS Table*(*MWSNameQoSTable*).

$$MWSNameQoSTable : \{WSId, ResTime, Availability, Reliability, Cost\}$$

### 3.3.3.6 I/O and QoS based service search

Our ESR provides additional Search functionalities based on web service input and output parameters, to support varying requirements of the user. This is applicable when

users like to search for a web service based on their commitments(inputs) and expectations(outputs). Tables and views are created in our schema dynamically on querying the registry. The descriptions of the various tables and views that are preprocessed/generated for parameter based search are as below :

1. The *Web Service Input Output Final Table(WSInOutFinal)* lists all web services in the registry with their respective input parameters (InPars) and output parameters (OutPars). InPars and OutPars are comma seperated lists of parameter Ids of the respective parameters.

$$WSInOutFinal : \{WSID, InPars, OutPars\}$$

2. The *Web Service Input Output Final Parameter Cluster Id Table(WSInOutFinalPCL)* lists all web services in the registry with their respective input parameters (InPars) and output parameters (OutPars), where InPars and OutPars are comma seperated lists of parameter Cluster Ids(PCId) of the respective parameters.

$$WSInOutFinalPCL : \{WSID, InPars, OutPars\}$$

3. A set of web services satisfying the output parameters as specified in the Query is first stored in a *Matched Web Services Table(MWSTable)*. The output match type of the web service(MatchType) along with the value of output deviation measure (X1), is stored in this table.

$$MWSTable : \{WSIDs, MatchType, X1\}$$

4. Web services from *MWSTable* that further satisfy the input parameters as specified in the query is next stored in a *Matched Web Services Input Output Table(MWSIOTable)*. The input match type of the web service(MatchType) along with the value of input deviation measure (X2), is stored in this table.

$$MWSIOTable : \{WSIDs, MatchType, X2\}$$

5. Finally, web services satisfying the given QoS constraints are selected from *MW-SIOTable*, ranked and stored along with their QoS Values in *Matched Web Services Input Output QoS Table*(*MWSIOQoSTable*).

*MWSIOQoSTable* :

$\{WSIDs, ResTime, Availability, Reliability, Cost, Rank, MatchType\}$

6. *Matched* is a view listing the matched web services with their respective QoS values.

*Matched* :

$\{WSID, ResTime, Availability, Reliability, Cost, Rank, MatchType\}$

7. *Exact Full* view lists web services with *Exact output* match and *Full input* Match along with their respective QoS values.

*ExactFull* :

$\{WSID, ResTime, Availability, Reliability, Cost, Rank, MatchType\}$

8. *Exact Partial* view lists web services with *Exact output* match and *Partial input* Match along with their respective QoS values.

*ExactPartial* :

$\{WSID, ResTime, Availability, Reliability, Cost, Rank, MatchType\}$

9. *Super Full* view lists web services with *Super output* match and *Full input* Match along with their respective QoS values.

*SuperFull* :

$\{WSID, ResTime, Availability, Reliability, Cost, Rank, MatchType\}$

10. *Super Partial* view lists web services with *Super output* match and *Partial input* Match along with their respective QoS values.



*SuperPartial* :

$\{WSID, ResTime, Availability, Reliability, Cost, Rank, MatchType\}$

11. *Partial Full* view lists web services with *Partial output* match and *Full input* Match along with their respective QoS values.

*PartialFull* :

$\{WSID, ResTime, Availability, Reliability, Cost, Rank, MatchType\}$

12. *Partial Partial* view lists web services with *Exact output* match and *Full input* Match along with their respective QoS values.

*PartialPartial* :

$\{WSID, ResTime, Availability, Reliability, Cost, Rank, MatchType\}$

13. *MinCost* view lists the web services matching the Cost constraint specified in the user query.

*MinCost* :  $\{WSID, ResTime, Availability, Reliability, Cost\}$

### 3.3.3.7 CST Construction

Following are the tables generated in the process of construction of *composition search tree(CST)*, that visualizes various service Compositions satisfying the user Query.

1. The *Root Match Table (RootMatch)* lists all the web services that match the user query, has details of the input(InPars),output(OutPars) parameters of each service, along with list of matched output parameters(MatchedPars) and a Degree Of Match(DOM) value that specifies the type of output match of the web service.

*RootMatch* :  $\{WSID, InPars, OutPars, MatchedPars, DOM\}$

2. The *Internal Node Match Table (IntNodeMatch)* lists all the web services that match the set of desired output parameters of an internal node in the CST, has details of the input(InPars),output(OutPars) parameters of each service, along with list of matched output parameters(MatchedPars) and a Degree Of Match(DOM) value that specifies the type of output match of the web service.

$$IntNodeMatch : \{WSID, InPars, OutPars, MatchedPars, DOM\}$$

In the next section we discuss the performance and scalability of our ESR in comparison to the standard UDDI service registry.

### 3.4 Performance and Scalability of ESR

The effectiveness of our proposed extended service registry is shown by conducting two sets of experiments:

1. Performance of basic keyword search in service registries - jUDDI v/s ESR approach.
2. Scalability of service registries - jUDDI v/s ESR approach.

#### 3.4.1 Experimental setup

To evaluate the response time of UDDI based service directories, we queried a service directory that is implemented in jUDDI. Apache jUDDI (pronounced "Judy") is an open source Java implementation of the Universal Description, Discovery, and Integration (UDDI v3) specification for web services Kurt T Stam (2014). We conducted experiments on QWS Dataset (Al-Masri and Mahmoud (2008)). We varied the number of web services that were stored in service registries implemented in jUDDI and ESR, from 100 to 500 and then to 1000. Five queries were submitted to the service directory in both the sets of experiments, and the response times of the service directory were

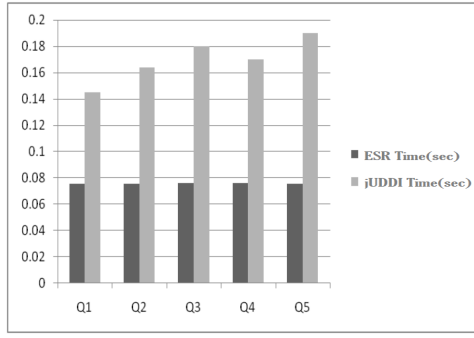
computed. We ran our experiments on a 1.3GHz Intel machine with 4 GB memory running Microsoft Windows 7. Our algorithms were implemented using Oracle 10g and JDK 1.6. Each query was run 5 times and the results obtained were averaged, to make the experimental results more sound and reliable.

### 3.4.2 Comparison of basic keyword search on conventional registries vs ESR approach

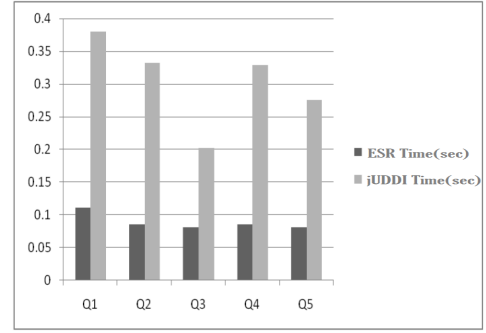
Web service search is performed in UDDI based service directories using predefined APIs. *find\_service()* function in inquiry API, of jUDDI, is used to locate specific services in service registry and returns a *serviceList* structure that matches the conditions specified in the arguments. The various arguments supported are : *authInfo*, *businessKey*, *findQualifiers*, *name* etc. The default behavior of UDDI with respect to matching is *exact match*.

In the first set of experiments we published 500 web services in jUDDI and ESR. We then queried jUDDI with five different user requirements. Experiments were done for both *approximateMatch* and *exactMatch* Qualifiers. Same set of requirements were then fed to ESR and their response times were recorded. The experiments were repeated by publishing 1000 web services in both the registries. Fig 3.2 and fig 3.3 shows the performance results for Queries  $Q_1$  to  $Q_5$  on both the service registries for *approximateMatch* and *exactMatch* Qualifiers respectively.

In the second set of experiments we varied the number of services published from 100 to 1000 in jUDDI and ESR. We then queried jUDDI with four different user requirements. Experiments were done for both *approximateMatch* and *exactMatch* qualifiers. The same set of requirements were then fed to ESR and their response times were recorded. Fig 3.4 depicts how the execution time for query  $Q_1$  varies on both the service registries, with an increase in the number of services registered.

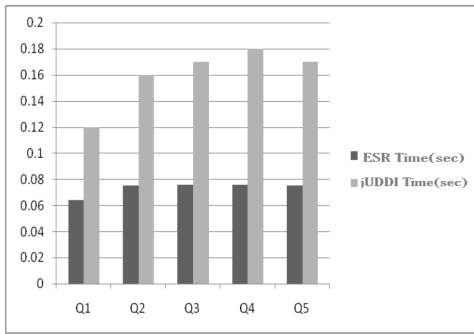


(a) Exact Match

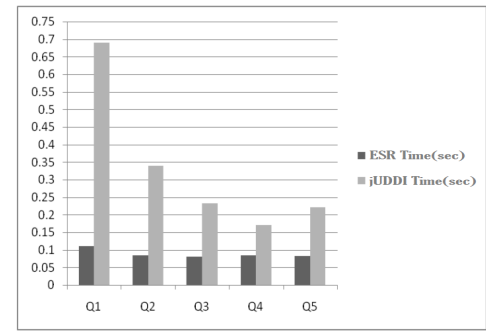


(b) Approximate Match

Figure 3.2: jUDDI vs ESR for 500 services



(a) Exact Match



(b) Approximate Match

Figure 3.3: jUDDI vs ESR for 1000 services

From the results obtained, we can infer that the time taken for search with approximate match is far far lesser in our ESR approach when compared to the time taken in jUDDI. Though the time taken for service search with exact match appears to be close to the time taken in jUDDI, its so only for the cases when search results have very few matching services, meaning, for search results that have many matching services, our approach works faster than jUDDI, both in the case of approximate match and exact match. Also, the search time becomes more efficient in our approach with a substantial increase in number of services in the registry. Thus we can finally conclude that service Search in ESR is more scalable and efficient than searching in UDDI.

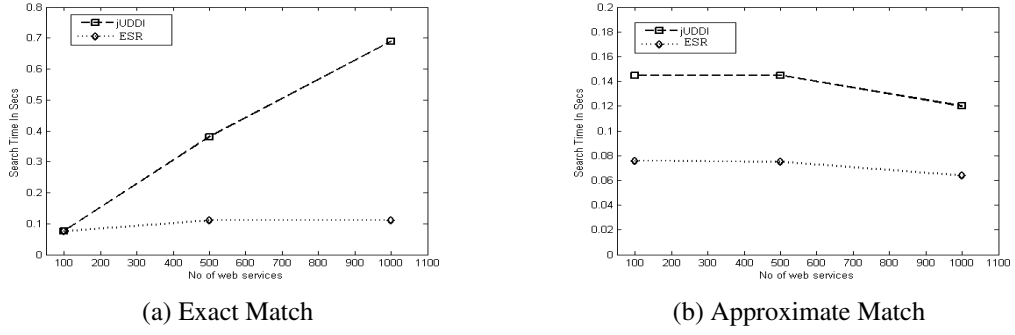


Figure 3.4: Service Search Time in jUDDI vs ESR

### 3.4.3 Process of parameter based service search

To empower our ESR with additional search capabilities, we define algorithms for input/output parameter based service search. Algorithm 1 presents pseudo-code for output parameter based service search.  $Q^O$  represent output parameters specified in user query. Similar algorithm is used for input parameter based service search. These algorithms were applied on ESR prior to parameter and web service clustering. Hence the performance of these algorithms show the efficiency of ESR prior to any pre-processing.

```

Input:  $Q^O$ , WSInOutTable : table
Output: MWSTable : table
foreach  $ParName$  in  $Q^O$  do
    Select PID from ParTable where PName =  $ParName$ 
    INSERT PID into the QueryT table
foreach  $ws$  in WSInOutTable do
    if  $ws^O = Q^O$  then
        INSERT  $ws$  as Exact Match in MWSTable
    else if  $ws^O \subset Q^O$  then
        INSERT  $ws$  as Partial Match in MWSTable
    else if  $ws^O \supset Q^O$  then
        INSERT  $ws$  as Super Match in MWSTable
    else
        continue

```

**Algorithm 1:** Output parameter based service Search

### 3.4.4 Performance of output parameter based service search

To empower the ESR with additional search capabilities we defined algorithms for input/output parameter based service search as discussed in section 3.4.3. To include input/output parameter based service search , we implemented Algorithm 1 and evaluated its performance using different output parameter patterns as user queries. In the first set of experiments we published 500 web services in ESR. The experiment was then repeated by publishing 1000 web services in ESR. Figure 3.5 shows the performance of output parameter based service Search in ESR.

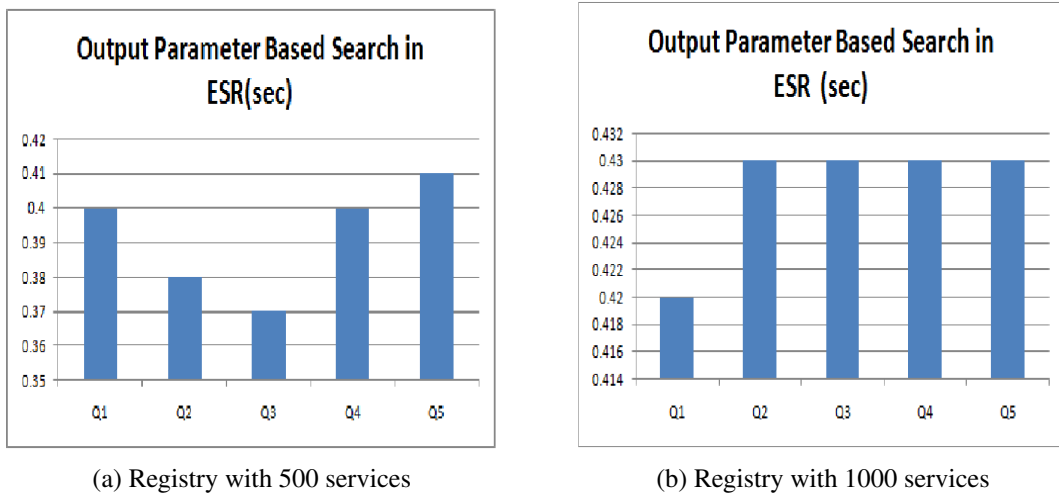


Figure 3.5: Output parameter based search in ESR

## 3.5 Conclusion

Query based web service search and composition is an important issue, especially for non-semantic web services. Traditional UDDI based service search lacks the ability to recognize all the features described in WSDL files. This leads to limited and poorly designed search operations that these registries offer. Experiments show that the performance of such service directories deteriorate with a substantial increase in number of services. Also, service search operations need to be extended to support varying requirements of the consumer. In order to improve the scalability and to empower service

registries with additional search capabilities we propose an ESR that uses Object Relational Databases as repository of web services as an extension to UDDI.

This chapter gives an introduction to the various service match types of web services that we have proposed. Further our ESR schema is explained in detail, giving the reader details of the tables and views that are used/generated by the various approaches that we propose in the following chapters.

We have simulated keyword based service search algorithms on ESR to study the performance and scalability of our ESR in comparison with jUDDI, using the standard QWS Dataset (Al-Masri and Mahmoud (2008)). The experimental results on varying user queries demonstrate that our ESR approach performs and scales better than jUDDI.

## CHAPTER 4

### Accelerating parameter based service search

#### Abstract

An *extended service registry*(ESR), to support varying requirements of the consumer, is proposed in the previous chapter. ESR as such supports parameter based service search. In order to widen the scope of search and to further improve the performance of parameter based search on ESR we propose a two-level pre-processing of service registry. The meta information extracted by this pre-processing are later used for searching services for a user query(i.e. user requirement). To widen the scope of input/output parameter matching, we initially cluster web service input/output parameters based on their similarity<sup>1</sup>. Further, to accelerate parameter based searching in our *extended service registry*, we propose an approach to cluster web services based on their output parameter pattern similarity<sup>2</sup>, utilizing the parameter clusters obtained in the previous step. This chapter explains both these clustering techniques in detail.

#### 4.1 Introduction

Among different techniques for service composition and service search, input / output parameter matching is a technique of often use(Hashemian and Mavaddat (2006); Talantikite *et al.* (2009); Arpinar *et al.* (2004); Gekas and Fasli (2005); Kwon *et al.* (2007); Lee *et al.* (2011); Zeng *et al.* (2010)), as discussed previously in survey chapter. The input and output parameters of a web service represents its functionality and can be utilized for finding similar and composable services. Based on this concept we have explored the feasibility of input/output based web service search in the previous chapter, where we propose an *extended service registry*, to support varying requirements of

---

<sup>1</sup>Lakshmi H. N., Hrushiksha Mohanty, A Preprocessing of service Registry: Based on I/O Parameter Similarity, published in proceedings of 11th International Conference on Distributed Computing and Internet Technology (ICDCIT 2015), Lecture Notes on Computer Science, Vol. 8956, 2015, pp 220-232

<sup>2</sup>Lakshmi H. N., Hrushiksha Mohanty, Clustering web services on frequent output parameters for I/O Based service search, published in proceedings of MIWAI 2014, Lecture Notes on Computer Science, Vol. 8875, pp 161-171



the consumer.

The extended service registry as such supports parameter based service search. In order to widen the scope of search and to further improve the performance of parameter based search in our ESR we propose a two-level pre-processing of service registry. **To widen the scope of input/output parameter matching, we initially cluster web service input/output parameters based on their similarity.** Further, **to accelerate parameter based searching in our *extended service registry*, we propose an approach to cluster web services based on their output parameter pattern similarity**, utilizing the parameter clusters obtained in the previous step. This chapter explains both these clustering techniques in detail.

Most of the current approaches for service search and service composition use semantic web services that have semantic tagged descriptions, e.g., OWL-S, WSDL-S, as discussed previously in survey. However, these approaches have many limitations: it is impractical to expect all services to have semantic tagged descriptions and descriptions of majority of existing web services are specified using WSDL and do not have associated semantics. Approaches that use non-semantic services often refer to domain ontology that could be at times large and complex enough for the purpose. Further, from the consumer's perspective, the consumer may be unaware of the domain and the exact terms to be used in the service request. As a result, such a search may overlook many relevant services.

In order to address these limitations a generic input/output parameter matching approach, not tied to a specific service description language, needs to be developed. Hence, **we propose to cluster web service input/output parameters based on their similarity**, computed using WordNet as the underlying ontology.

Further, **we propose an approach to cluster web services based on their output parameter pattern similarity**, that utilizes the clusters generated by the parameter clustering approach, to effectively match the output parameters of web services in our extended service registry. Feature based clustering of web services in a service registry reduces search space while looking for a web service from the registry to meet a search requirement.

As discussed previously in survey, the current approaches employ clustering of web services during the service search process, to cluster search results for a given user query. Mostly, the approach of query processing follows two basic steps(Ma *et al.* (2008)) - reducing the domain of search by service domain e.g. hotel, transport etc. and then grouping the services based on query requirements. As these are done during processing of a query, it becomes expensive in terms of time. Instead, **we propose an approach that does a book keeping of web services following an order so that the pre-processing helps to reduce search time.**

Our clustering approach makes use of the co-occurrence of output parameters to cluster web services. Set of output parameters that co-occur in more than a threshold number of web services make a *frequent output parameter pattern*. A set of services having the same frequent output parameter pattern make a *candidate cluster*. From the many candidate clusters generated we need to choose a subset of candidate clusters such that selected clusters form a *covering cluster*. A *covering cluster*(a set of clusters) covers all web services in the registry and also have a minimum overlap between them. Inclusion of a candidate cluster into *covering cluster* is governed by its *cluster overlap* and *preference factor*. We define cluster overlap of a candidate cluster based on the number of frequent output parameter patterns that the services in a candidate cluster is supporting. Further, a cluster's preference factor suggests its inclusion into *covering cluster*, based on the number of non-overlapping web services that it contributes to final clustering. Figure 4.1 depicts the two-level pre-processing of registry discussed above.

When a user queries for a service with a required output parameter pattern, initially, a cluster from the *covering cluster*, that best matches the queried pattern is searched for. This matched cluster contains many services whose parameter pattern matches with queried pattern. And these services are further classified into three groups: exact, partial and super, based on the type of match the output parameter pattern of a service has with queried output parameter pattern.

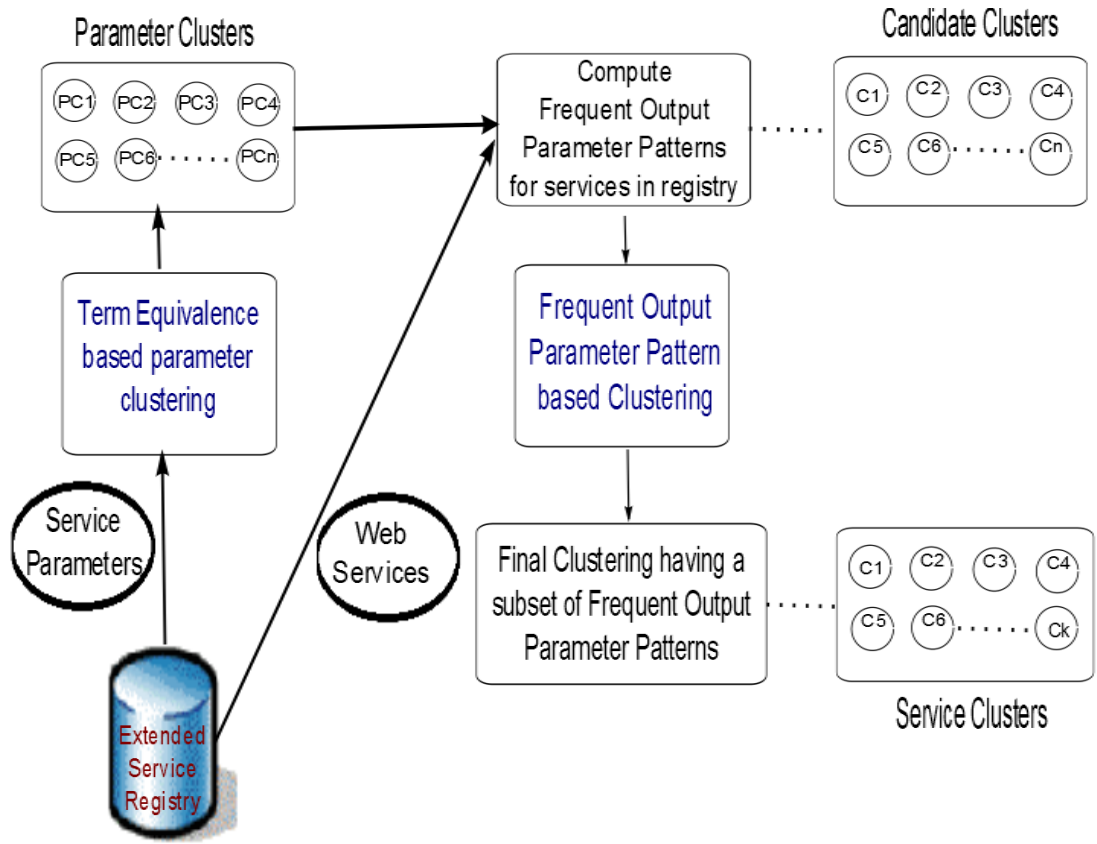


Figure 4.1: Pre-processing registry for service search

We first describe our approach for parameter clustering based on similarity of parameter names in section 4.2. In section 4.3 that follows we propose an approach to cluster web services on frequent output parameter patterns. The experimental results of both the clustering approaches is discussed in section 4.4. Conclusion is given in section

## 4.2 Semantics for service search

A parameter name, (eg: FlightInfo, CheckHotelCost), is typically a sequence of concatenated words, (eg: Flight, Check, Hotel, Cost), referred to as terms. For effectively matching input/output parameters of web services, it is essential to consider their underlying semantics. However, this is hard since parameter names are not standardized and are usually built using terms that are highly varied due to the use of synonyms, hypernyms, and different naming conventions. Hence, to widen the scope of input/output parameter matching, it is useful to cluster service input/output parameters based on their similarity.

We make use of semantic similarity and co-occurrence of terms in parameter names, computed using WordNet as the underlying ontology, to cluster service parameters into semantic groups. We propose a semantic similarity measure for computing similarity of parameters and an approach for clustering service parameters based on the classical DB-SCAN algorithm Ester *et al.* (1996). The parameter clustering serves as a pre-processing step for input/output parameter based web service search. Experimental evidence shows that the clusters generated by our algorithm has a better Precision and Recall values when compared with the clusters generated by K-means algorithm MacQueen *et al.* (1967).

### 4.2.1 Similarity metrics

The similarity/distance measure reflects the degree of closeness/separation between a symbolic description of two objects into a single numeric value and hence should correspond to the characteristics that distinguish the clusters embedded in a dataset. The similarity metrics like edit distance etc. mostly measure a kind of structural similar-

ity with scant regard to semantic similarity. For this application semantic based search is wishful for providing a number of alternative web services to a user on need. A cluster of web services is formed on the basis of similarity of their input/output parameters. Then for a given set of query input/output parameters, corresponding set of alternative services can be obtained from a cluster that is selected by matching of query input/output parameters to input/output parameters of a cluster. Thus, in general measuring similarity between two parameters has an importance for preprocessing of service registry to find clusters of services. Clustering of parameters based on similarity is a two-step method. First, a parameter is tokenized to terms and second, the parameter similarity is computed.

#### **4.2.1.1 Tokenizing web service input/output parameters**

A parameter name, (eg: FlightInfo, CheckResortCost), is typically a sequence of concatenated words like Flight, Check, Resort, etc., referred to as terms. We hence make use of semantic similarity of the terms in service registry to form clusters of parameter names. Further these clusters are used for searching web services for a given query input/output parameters.

Due to the nature of parameter names normally included in a WSDL, we first apply a tokenization process to produce set of terms to be actually matched before applying the parameter name similarity function. For eg., parameter names included in a web service could be like HotelName or FlightCost which are difficult to find in a general purpose ontology. On the contrary, we can find the terms composing these names: e.g., hotel, cost, name, flight, in the ontology. Hence, we tokenize a parameter name to obtain the terms to be used for similarity calculation. A parameter name  $P_i$  is tokenized to a term vector  $T_{P_i}$  according to a set of rules inspired by common naming conventions, taking care of case change, presence of underscore and hyphenation, etc.

#### 4.2.1.2 Parameter similarity function ( $pSim$ )

The parameter similarity function,  $pSim$ , calculates the semantic similarity between two parameters and returns a value in  $[0...1]$ . The higher the value of  $pSim$ , the higher the similarity between the two parameters. In particular  $pSim(P_1, P_2) = 1$  iff  $P_1$  is semantically similar to  $P_2$ , whereas,  $pSim(P_1, P_2) = 0$ , iff  $P_1$  and  $P_2$  are totally semantically unrelated. The  $pSim$  function relies on a maximization function called  $maxMeas$  that computes the maximum measure between the two term vectors we are comparing.

Par No	parameter Names	Placement for <i>Hotel</i>
1	Get <b>Hotel</b> Cost	second
2	Fetch <b>Room</b> Price	second (Synonymn)
3	Check <b>Resort</b> Cost	second (Synonymn)
4	Request <b>Room</b> Rate	second (Synonymn)
5	Hotel <b>Cost</b> Enquiry	first
6	<b>Resort</b> Price	first (Synonymn)
7	RequestFor <b>Hotel</b> Rate	third
8	CostOf <b>Hotel</b>	third

Table 4.1: Different placements of *Hotel* in parameters

#### 4.2.1.3 Maximization function ( $maxMeas$ )

On inspecting many web services in dataset Al-Masri and Mahmoud (2008), we observed that most of the parameter names are built of domain specific terms. Table 4.1 lists a few example parameter names that can be used to get the cost of a hotel or room. It can be seen that though the terms are similar, their placement in the parameters differ from one to another. For ex: looking for the term *Hotel* in all the parameter names listed in Table 4.1, we observe that it is the first term in 5, second term in 1, third term in 7 and 8, parameters 2 and 4 have synonym *Room* as second term and parameters 3 and 6 have Synonymn *Resort* as second and sixth term respectively. Given a set of terms for parameters, they can be placed anywhere based on the naming conventions used by the provider. Hence, given the term vectors  $T_{P_i}$  and  $T_{P_j}$ , we compare each term in a parameter  $T_{P_i}$  with all terms in  $T_{P_j}$ .

The maximization function is formulated as a maximum weighted bipartite matching problem. Given a graph  $G = (V, E)$ , a matching  $M \subseteq E$ , is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex. A maximum weighted bipartite matching is defined as a matching where the sum of the values of the edges in the matching have a maximal value.

Let  $A$  and  $B$  be set of vertices representing the term vectors that need to be matched, with an associated edge weight function given by measuring function  $mf : (A, B) \rightarrow [0...1]$ . The maximization function  $maxMeas : (mf, A, B) \rightarrow [0...1]$  returns the maximum weighted bipartite matching, i.e., a matching so that the average of the edge weights is maximum. Expressing the maximization function using linear programming model, we have :

$$\begin{aligned}
 maxMeas(mf, A, B) &= \frac{\max \left[ \sum_{i \in I, j \in J} mf(A_i, B_j) \cdot x_{ij} \right]}{\max \{ \|A\|, \|B\| \}} \\
 \text{subject to,} \\
 \sum_{j \in J} x_{ij} &= 1, \quad \forall i \in I, \quad \sum_{i \in I} x_{ij} = 1, \quad \forall j \in J \\
 0 \leq mf(A_i, B_j) &\leq 1, \quad i \in I, \quad j \in J \\
 I &= [1... \|A\|], \quad J = [1... \|B\|]
 \end{aligned} \tag{4.1}$$

$\|A\| < \|B\|$  implies that the number of terms in  $A$  is lesser than the number of terms in  $B$ ; so, for each term in  $A$ , we may find a similar term in  $B$ . On the contrary,  $\|A\| > \|B\|$  always evaluates to a  $maxMeas < 1$ . Since our approach aims to compute the similarity between  $A$  and  $B$ , we divide the result of the maximization by the maximum cardinality of  $A$  and  $B$ . Hence, the function  $maxMeas$  is symmetric, i.e.,  $maxMeas(mf, A, B) = maxMeas(mf, B, A)$ . Apart from  $\|A\|$  and  $\|B\|$ , the  $maxMeas$  function is affected by  $mf : (A, B)$  values. Figure 4.2 illustrates the computation in  $maxMeas$  function, where the bold lines constitute the maximum weighted bipartite matching.

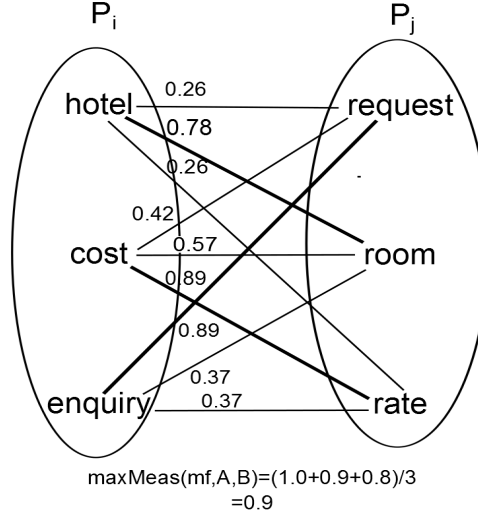


Figure 4.2: Illustration of  $\maxMeas$  function

Given two parameters  $P_A$  and  $P_B$  and their term vectors  $T_A$  and  $T_B$ , their semantic similarity is evaluated using the  $\maxMeas$  function introduced above as:

$$pSim(P_A, P_B) = \maxMeas(simMeas, T_A, T_B) \quad (4.2)$$

The weights of the edges are given by  $simMeas : (term_{Ai}, term_{Bj}) \rightarrow [0..1]$ .

#### WordNet for similarity computation :

Several approaches are available for computing semantic similarity among terms, by mapping terms (concepts) to an ontology and by examining their relationships in that ontology. Some of the popular semantic similarity methods are implemented and evaluated using WordNet as the underlying reference ontology Pedersen *et al.* (2004). WordNetMiller (1995) is a widely used on-line lexical reference system developed at Princeton University. WordNet attempts to model the lexical knowledge of a native speaker of english. WordNet can also be seen as an ontology for natural language terms. It contains over 150,000 terms (nouns, verbs, adjectives and adverbs) which are grouped into sets of synonyms called synsets. The synsets are also organized into senses (i.e., corresponding to different meanings of the same term or concept). The synsets (or con-



cepts) are related to other synsets higher or lower in the hierarchy by different types of relationships. The most common relationships are the Hyponym/Hypernym (i.e., Is-A relationships), and the Meronym/Holonym (i.e., Part-Of relationships).

Several methods for determining semantic similarity between terms have been proposed in the literature and most of them have been tested on WordNet. Similarity measures apply only for nouns and verbs in WordNet. Of the many methods, those based on information content of the least common subsumer (LCS) of given two concepts A and B are more apt for our application. Information content is a measure of the specificity of a concept, and the LCS of concepts A and B is the most specific concept that is an ancestor of both A and B, where the concept tree is defined by the is-a relationship. These measures include res (Resnik (1995)), lin (Lin (1998)), and jcn (Jiang and Conrath (1997)). The lin and jcn measures augment the information content of the LCS with the sum of the information content of concepts A and B themselves. The lin measure scales the information content of the LCS by this sum, while jcn takes the difference of this sum and the information content of the LCS.

We use jcn (Jiang and Conrath) measure for computing the semantic similarity between two terms. This measure uses a combination of edge counts in the WordNet 'is-a' hierarchy and the information content values of the WordNet concepts as described by Jiang and Conrath in (Jiang and Conrath (1997)). Their measure, however, computes values that indicate the semantic distance between words (as opposed to their semantic relatedness). The value is inverted to obtain a measure of semantic relatedness. The semantic relatedness value is computed as below :

$$\text{Semantic Relatedness}(A, B) = \frac{1}{JCN_d}$$

where  $JCN_d$ , the jcn distance is given by

$$JCN_d = IC(\text{synset1}) + IC(\text{synset2}) - 2 * IC(lcs). \quad (4.3)$$

where  $IC$  is the Information Content.

Note that the following properties hold for the *parameter similarity function* ( $pSim$ ):

1.  $0 \leq pSim(P_A, P_B) \leq 1$ .  $pSim(P_A, P_B) = 1$  iff the two parameters are totally similar and fully replacable and  $pSim(P_A, P_B) = 0$  iff the two parameters are totally unrelated.
2.  $pSim$  is symmetric , i.e.,  $pSim(P_A, P_B) = pSim(P_B, P_A)$ .

## 4.2.2 Term equivalence based parameter clustering

Our clustering is based on the following heuristic : " *parameters tend to express the same concept if the terms in their names are semantically similar*" and is validated by experimental results. The clustering algorithm, *similarity based clustering of service parameters*(SCSP), is based on the classical DBSCAN Algorithm *citedbscan*). *DBSCAN is a density based*

### 4.2.2.1 Definitions

The following definitions are used in our clustering algorithms :

1.  $\epsilon$  **Semantic neighborhood of a parameter** : We define  $\epsilon$  *semantic neighborhood* of a parameter  $P_i$  in a set of parameters  $P$ , as follows:

$$SN_\epsilon(P_i) = \{P_j \in P | pSim(P_i, P_j) \geq \epsilon\} \quad (4.4)$$

where  $\epsilon$  is the minimum parameter similarity value defined to form clusters.

2. **Core parameter(CP):** A parameter is a *core parameter*, iff,

$$|SN_{\epsilon}(CP)| \geq MinSP \quad (4.5)$$

where  $MinSP$  is the minimum number of similar parameters required to make a cluster.

3. **Noise:** Let  $C_1, \dots, C_k$  be the clusters of parameters in  $P$  wrt.  $\epsilon$  and  $MinSP$ . We define *noise* as the set parameters in  $P$  not belonging to any cluster  $C_i$ , i.e.,

$$Noise = \{p \in P | \forall i : p \notin C_i\} \quad (4.6)$$

### 4.2.3 Parameter clustering algorithm

To find a cluster, algorithm 2 starts with an arbitrary parameter  $P_i$  in  $P$  and retrieve all parameters that are in  $\epsilon$  semantic neighborhood of  $P_i$ . If  $P_i$  is a *core parameter*, as defined in eqn 4.5, then this set of parameters yields a cluster with respect to  $\epsilon$  and  $MinSP$ . If  $P_i$  is not a core parameter, then the algorithm continues by searching for another core parameter in  $P$ . This procedure is continued till all the clusters and *Noise* in  $P$  are generated. Algorithm 2 and procedure Coreparameter provides the psuedocode for the clustering algorithm and the procedure for selecting a core parameter.

```

Input:  $pSim$  for  $P, \epsilon, MinSP$ 
Output: Parameter clusters  $PC$ 
1  $PC = \emptyset$ 
2  $Noise = \emptyset$ 
3 foreach  $P_i$  in  $P$  do
4    $nextP = P_i$ 
   // Let parameter cluster created be  $pcc$ 
5    $pcc = \text{Coreparameter}(nextP, \epsilon, MinSP)$ 
6   if  $pcc \neq \emptyset$  then
7      $PC = PC \cup pcc$ 
8   else
9      $Noise = Noise \cup nextP$ 

```

**Algorithm 2:** Similarity based clustering of service parameters

```

Input:  $nextP, pSim$  for  $P, \epsilon, MinSP$ 
Output: Parameter cluster  $pcc$ 
1  $count := 0$ 
   // Let parameter cluster created be  $pcc$ 
2  $pcc = \emptyset$ 
3 foreach  $P_i$  in  $P$  do
4   if  $pSim(P_i, nextP) \geq \epsilon$  then
5      $count := count + 1$ 
6      $pcc = pcc \cup P_i$ 
7 if  $count \geq MinSP$  then
8   return  $pcc$ 
9 else
10   $pcc = \emptyset$ 
11  return  $pcc$ 

```

**Procedure** Coreparameter

#### 4.2.3.1 Cluster validation

Ideally, clusters generated by SCSP Algorithm should have the following features:

1. Rare parameters should be left unclustered.

2. The cohesion of a cluster(the similarity between parameters inside a cluster) should be strong; the correlation between clusters(the similarity between parameters in different clusters) should be weak.

Traditionally, cohesion is defined as the sum of squares of Euclidean distances from each point to the center of the cluster it belongs to; correlation is defined as the sum of squares of distances between cluster centers(Hand *et al.* (2001)). Clearly, these measures cannot be applied to our context because of the similarity measure utilised for clustering. We hence quantify the cohesion and correlation of clusters based on parameter similarity function defined for clustering.

Given a cluster  $C_I$ , we define the cohesion of  $C_I$  as the percentage of closely associated parameter pairs over all parameter pairs,i.e.,

$$Coh(C_I) = \frac{\|(P_i, P_j) \mid P_i, P_j \in C_I, i \neq j, pSim(P_i, P_j) > \epsilon\|}{\|C_I\| \cdot \|C_I - 1\|} \quad (4.7)$$

Given two cluster  $C_I$  and  $C_J$ , we define the correlation between  $C_I$  and  $C_J$  as the similarity measure between the core parameters of the two clusters,i.e.,

$$Cor(C_I, C_J) = pSim(CP(C_I), CP(C_J)) \quad (4.8)$$

An ideal clustering has a high average cohesion value while maintaining a low average correlation value. Our goal is to obtain a parameter clustering with such properties.

### 4.3 Output parameter pattern based service clustering

To improve the performance of parameter based search in our extended service registry, we propose an approach to cluster services based on their output parameter pattern similarity. This approach utilizes the parameter clusters obtained in the previous step. Each

output parameter of a web service is represented by the *cluster id* to which it belongs to, and hence the output parameter pattern is now reduced to a pattern of parameter cluster ids. It is on these patterns that we apply our clustering process. We use the term *output parameter pattern* to represent a *output parameter set*. A clustering approach for web services based on *frequent output parameter patterns* looks promising since it provides a natural way of reducing the candidate services when we are looking for a web service with desired *output parameter pattern*.

In data mining, association rule mining is a popular and well researched method intended to extract interesting correlations, frequent patterns, associations among sets of items in large databases or other data repositories. Frequent itemsets form the basis of association rule mining. Apriori(Agrawal *et al.* (1994)) is a classic algorithm for frequent itemset mining and association rule learning over transactional databases.

Apriori algorithm is utilized here to obtain the *frequent output parameter patterns* among the many *output parameter patterns* that the web services in the registry may have. The key idea is to consider the *frequent output parameter patterns*, along with web services supporting them, as candidate clusters and further, choose a subset of candidate clusters that covers all the web services registered in our proposed extended service registry.

### 4.3.1 Definitions

Let  $R = \langle P, W \rangle$  be a service registry, where  $P$  is a set of parameters,  $P = \{p_1, p_2, \dots, p_n\}$  and  $W$  is a set of web services in the registry,  $W = \{ws_1, ws_2, \dots, ws_n\}$ . Each web service,  $ws_i$  in  $W$ , has typically two sets of parameters as set of inputs  $ws_i^I$  and set of outputs  $ws_i^O$ , from  $P$ .

- **Output parameter pattern:** We use the term *output parameter pattern* to repre-

sent a set of output parameters, i.e,  $O_P \subseteq P$ .

- **Cover:** For any service output parameter pattern,  $O_P \subseteq P$ , let  $C(O_P)$  denote the *cover* of  $O_P$ , defined as the set of all services supporting  $O_P$ , i.e,  

$$C(O_P) = \{ws_i \in W | O_P \subseteq ws_i^O\}.$$
- **MinSupp:** Let  $MinSupp$  be a real number, such that  $0 \leq MinSupp \leq 1$ , representing the minimum support used in the mining process to generate the set of all *frequent output parameter patterns*. The value of  $MinSupp$  governs the minimum number of services that is to be present in a candidate cluster.
- **Frequent output parameter pattern:** Let  $S = \{S_1, S_2, \dots, S_k\}$  be the set of all *frequent output parameter patterns* in  $W$  with respect to  $MinSupp$ , obtained from mining services in registry.
- **Candidate cluster :** The cover of each  $S_j \in S$ , i.e,  $C(S_j)$  can be regarded as a *candidate cluster*.
- **Covering cluster:** We define *covering cluster*,  $CC$ , as a subset of  $S$  that minimally covers the entire registry  $W$ , i.e,

$$CC = \{S_j | S_j \in S\} \text{ such that } \bigcup C(S_j) = W$$

A set of services that cover a frequent output parameter pattern is a *candidate cluster*. To choose an appropriate subset from the set of all *frequent output parameter patterns* we propose to use the mutual overlap between services supporting the different *frequent output parameter patterns*.

Our approach assumes that each web service in the registry supports at least one *frequent output parameter pattern*. We need to determine a *covering cluster* with a minimum overlap of the clusters. To measure the overlap between the different *candidate clusters* we propose to use the mutual overlap between the services supporting

different *Candidate clusters*. Further, a preference factor is used to suggest the inclusion of a *candidate cluster* into *covering cluster*.

Our approach uses a greedy algorithm which initially selects a *candidate cluster* that has the least cluster overlap, from the set of other *candidate clusters*, and includes them in  $CC$  if its contribution to  $CC$  is greater than the preference factor. The process is repeated until  $CC$  covers the entire registry  $W$ . So for a given registry  $R$  involving  $P$  parameters,  $CC$  covers all possible queries on *output parameter patterns*.

Let  $NFS_i$  denote the number of *frequent output parameter patterns* supported by a web service  $ws_i$ , i.e.,

$$NFS_i = \{|S_j \in S | S_j \subseteq ws_i^O\}, \forall ws_i \in W.$$

**Cluster overlap :** We define *cluster overlap* of a *candidate cluster*  $C_j$ , representing a *frequent output parameter pattern*  $S_j$ , denoted by  $CO(C_j)$ , as the average  $NFS_i - 1$  value of all web services in the cluster, i.e.,

$$CO(C_j) = \frac{\sum_{ws_i \in C_j} (NFS_i - 1)}{|C_j|} \quad (4.9)$$

Smaller the values of  $NFS_i$  are, smaller will be the value of *cluster overlap* of a cluster  $C_j$ . Ideally, if  $NFS_i = 1$  for all services in the cluster  $C_j$  then  $C_j$  has an overlap of 0 with the other candidate clusters.

To minimize the effect of monotonicity property of *frequent output parameter pattern* sets (each subset of a *frequent output parameter pattern* set is also frequent), we have defined the *cluster overlap* of clusters based on the *frequent output parameter pattern* that the clusters represent. Also, we include only those clusters into *covering cluster* that results in a sufficient change in cover of  $CC$ , thereby ignoring clusters that have services already included in the cover.



**Preference factor** : Let  $|CC|$  represent the number of services, excluding repetitions, in the clusters currently present in *covering cluster*,  $CC$ . After calculating *cluster overlap* for all the cluster candidates, we choose a cluster candidate having the least overlap value,  $C_j$ , and add it to  $CC$  if inclusion of  $C_j$  into  $CC$  will increment the size of *covering cluster* by a factor greater than or equal to *preference factor*(pf) i.e, we include the cluster  $C_j$  into  $CC$  if it satisfies the below relation,

$$\frac{(|CC \cup C_j| - |CC|)}{(|W|)} \geq pf, \quad 0 \leq pf \leq 1 \quad (4.10)$$

The process is repeated for all *frequent output parameter patterns*, until the *covering cluster*,  $CC$ , covers all the web services in the registry, i.e,  $|CC| = |W|$ .

### 4.3.2 Frequent output parameter pattern based clustering algorithm

As defined earlier *covering cluster*,  $CC$ , is a subset of the set of all *frequent output parameter patterns*,  $S$ , that minimally covers the entire service registry  $W$ . Due to the inherent complexity of the *frequent output parameter pattern* based clustering approach: the number of all subsets of  $S$  is  $O(2^{|S|})$  an exhaustive search is prohibitive, hence we present a greedy approach for discovering *covering cluster*.

Algorithm *FOPC*, *frequent output parameter pattern based clustering*(*FOPC*), determines a *covering cluster* that covers the entire service registry  $W$  with minimum *cluster overlap*. Algorithm 3 presents a pseudo-code for *frequent output parameter pattern based clustering*. Table 4.2 illustrates the frequent patterns obtained on a sample set of web services containing 12 web services. The highlighted rows indicates the working of Algorithm *frequent output parameter pattern based clustering* on the sample set with  $MinSupp=0.16$  and  $pf = 0.1$ .

```

Input:  $W, \text{MinSupp}, f$ 
Output: Covering cluster,  $CC$ 
 $CC = \emptyset$ 
// Find all frequent output parameter patterns
 $S = \text{DetermineS}(W, \text{MinSupp})$ 
forall the  $S_i \in S_k$  do
    Calculate  $CO(S_i)$  using eqn 4.9
forall the  $S_i \in S$  do
    // Let  $CCl$  represent Chosen Cluster
    // Let  $\text{LeastCO}()$  be a function that returns a
    cluster with least cluster overlap
     $CCl = \text{LeastCO}(S)$ 
    if  $CCl$  satisfies eqn 4.10 then
         $CC = CC \cup CCl$ 
         $S = S - S_i$ 
    if ( $|CC| = |W|$ ) then
        break
    if ( $|CC| \neq |W|$ ) then
        while ( $|CC| \neq |W|$ ) do
             $CCl = \text{LeastCO}(S)$ 
             $CC = CC \cup CCl$ 
             $S = S - S_i$ 
return  $[CC]$ 

```

**Algorithm 3:** FOPC

### 4.3.3 Accelerating output parameter based service search

When a user queries for a service with a required output parameter pattern, initially, a cluster from the *covering cluster*, that best matches the queried pattern is searched for. This matched cluster contains many services whose parameter pattern matches with queried pattern. And these services are further classified into three groups: exact, partial and super, based on the type of match the output parameter pattern of a service has with queried output parameter pattern. One may obtain many services in each match type, from which best matching services need to be selected. Hence we propose a service selection approach as a next step in our next chapter.

The clusters obtained from Algorithm *FOPC* are stored in a ClusterTable to ac-

celerate parameter based service search. Algorithm 4 presents pseudo-code for output parameter based service search utilizing the ClusterTable.  $Q^O$  represent output parameters specified in user query.

service output parameter Pattern	Cluster Candidates	Cluster Overlap
<b>Period</b>	<b>ws6,ws8</b>	<b>0.5</b>
HotelName	ws1,ws10,ws11	9.33
FlightInfo	ws2,ws10,ws11	9.33
CarType	ws3,ws10,ws11	9.0
<b>TourInfo</b>	<b>ws4,ws8,ws12</b>	<b>0.66</b>
<b>PackageID</b>	<b>ws9,ws12</b>	<b>0.5</b>
<b>TaxiCost</b>	<b>ws5,ws11</b>	<b>8.0</b>
<b>TourCost</b>	<b>ws7,ws10,ws11</b>	<b>8.66</b>
<b>HotelCost</b>	<b>ws1,ws11</b>	<b>9.0</b>
<b>FlightCost</b>	<b>ws2,ws11</b>	<b>9.0</b>
HotelName,HotelCost	ws1,ws11	9.0
FlightInfo,FlightCost	ws2,ws11	9.0
<b>CarType,TaxiCost</b>	<b>ws3,ws11</b>	<b>8.5</b>
HotelName,FlightInfo	ws10,ws11	13
HotelName,CarType	ws10,ws11	13
HotelName,TourCost	ws10,ws11	13
HotelName,FlightInfo,CarType	ws10,ws11	13
HotelName,FlightInfo,TourCost	ws10,ws11	13
FLightInfo,CarType,TourCost	ws10,ws11	13
HotelName,FlightInfo,CarType,TourCost	ws10,ws11	13

Table 4.2: Illustration of Algorithm *FOPC* for Sample web services

## 4.4 Experiments

### 4.4.1 Experimental Setup

To evaluate the performance of algorithm *SCSP*, discussed in section 4.2.3, and algorithm *FOPC*, explained in section 4.3, we conducted experiments on QWS dataset (Al-Masri and Mahmoud (2008)). We ran our experiments on a 1.3GHz Intel machine with 4 GB RAM running Microsoft Windows 7. Our algorithms were implemented with

```

Input:  $Q^O$ , ClusterTable : table
Output: MWSTable : table
foreach  $ParName$  in  $Q^O$  do
    | Select PID from ParTable where PName =  $ParName$ 
    | INSERT PID into the QueryT table
foreach  $Cl$  in ClusterTable do
    | cntWS = Number of output parameters in  $Cl.ClusterPars$  matching those
    | in  $Q^O$ 
    | if  $cntWS > 0$  then
    |     | foreach  $ws$  in  $Cl.WSList$  do
    |         | if  $ws^O = Q^O$  then
    |             | INSERT  $ws$  as Exact Match in MWSTable
    |         | else if  $ws^O \subset Q^O$  then
    |             | INSERT  $ws$  as Partial Match in MWSTable
    |         | else
    |             | INSERT  $ws$  as Super Match in MWSTable

```

**Algorithm 4:** Service search using clusters

JDK 1.6, Eclipse 3.6.0 and Oracle 10g. Each query was run 5 times and the results obtained were averaged, to make the experimental results more sound and reliable.

#### 4.4.2 Parameter clustering results

In this section, we present an experimental evaluation of our parameter clustering approach by measuring:

1. Quality of parameter clustering.
2. Performance of parameter clustering.

The values for the parameters  $\epsilon$  and  $MinSP$  for *SCSP* Algorithm is set as 0.75 and 3 respectively. For implementing the parameter similarity measure we used WS4J (WordNet Similarity for Java)Package(hideki (2013)) that provides a pure Java API for several published semantic relatedness/similarity and can be used on Princeton's English WordNet 3.0. The experiments were initially conducted for 200 parameters in the registry and then repeated for all the 500 input/output parameters of 1000 web services in the registry.

#### 4.4.2.1 Evaluating quality of parameter clustering

To evaluate the quality of *SCSP* clustering algorithm, we introduce two metrics cluster cohesion and correlation as defined in section 4.2.3.1. We compare the performance of our approach with clusters generated by K-means(MacQueen *et al.* (1967)) clustering approach. K-means is a widely adopted clustering algorithm that is simple and fast. Its drawback is that the number of clusters has to be predefined manually before clustering. On applying *SCSP* and K-means algorithm on 200 service input/output parameters, 5 clusters were obtained, for which cluster cohesion was calculated. Repeating the same process for 500 service input/output parameters yielded 10 clusters. Figure 4.3 compares the cluster cohesion values obtained for the clusters generated by *SCSP* algorithm with those generated by K-means clustering. The average cluster cohesion for *SCSP* algorithm is 0.623 whereas for K-means algorithm is 0.295 for 500 service input/output parameters, as seen in figure 4.3b. It can be inferred from the results that our approach generates clusters with higher cluster cohesion values, thus generating clusters of better quality.

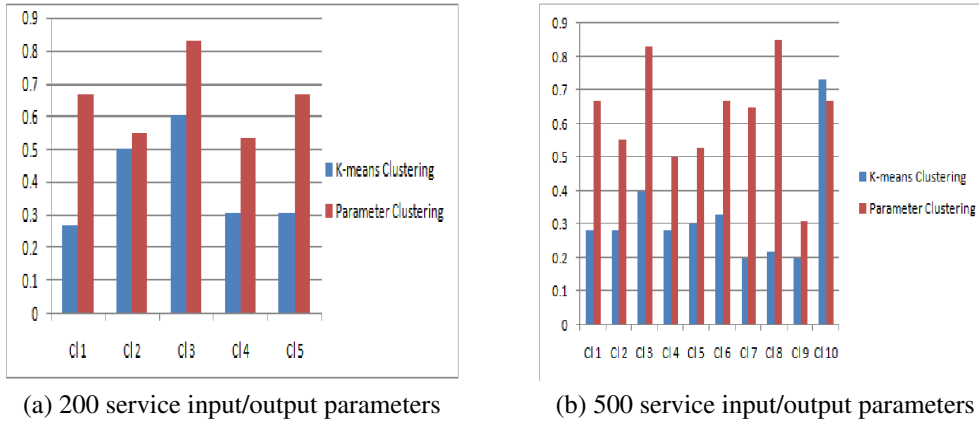


Figure 4.3: Cluster cohesion of generated clusters

#### 4.4.2.2 Performance of parameter clustering

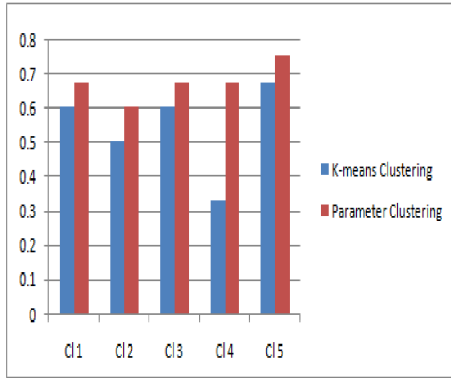
We use two standard metrics: *precision* and *recall*, widely adopted in information retrieval domain, to measure the overall performance of *SCSP* clustering algorithm. The

performance of our approach is compared with clusters generated by the standard K-means(MacQueen *et al.* (1967)) algorithm. Let  $SP(C_i)$  be the number of parameters that are correctly placed in the cluster  $C_i$ ,  $MP(C_i)$  be the number of parameters that are misplaced in the cluster  $C_i$  and  $MSP(C_i)$  be the number of parameters that should have been placed in  $C_i$  but has been placed in another cluster. Then *Precision* and *Recall* of a cluster  $C_i$  is defined as below :

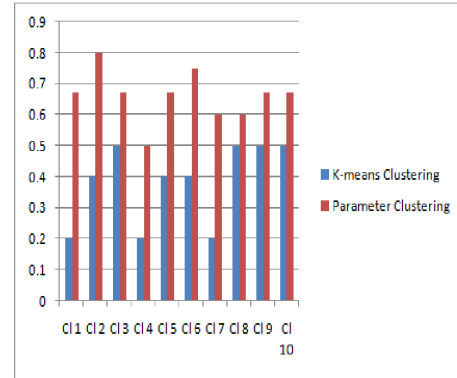
$$Precision(C_i) = \frac{SP(C_i)}{SP(C_i) + MP(C_i)} \quad (4.11)$$

$$Recall(C_i) = \frac{SP(C_i)}{SP(C_i) + MSP(C_i)} \quad (4.12)$$

Figures 4.4 and 4.5 compares the values of Precision and Recall of clusters generated by *SCSP* algorithm versus that of K-means clustering, for 200 and 500 service input/output parameters respectively. The average Precision and Recall values obtained is summarized in Table 4.3 It can be inferred from the results obtained that our approach outperforms the traditional K-means approach both in Precision and Recall.



(a) 200 service input/output parameters



(b) 500 service input/output parameters

Figure 4.4: Precision of Generated Clusters

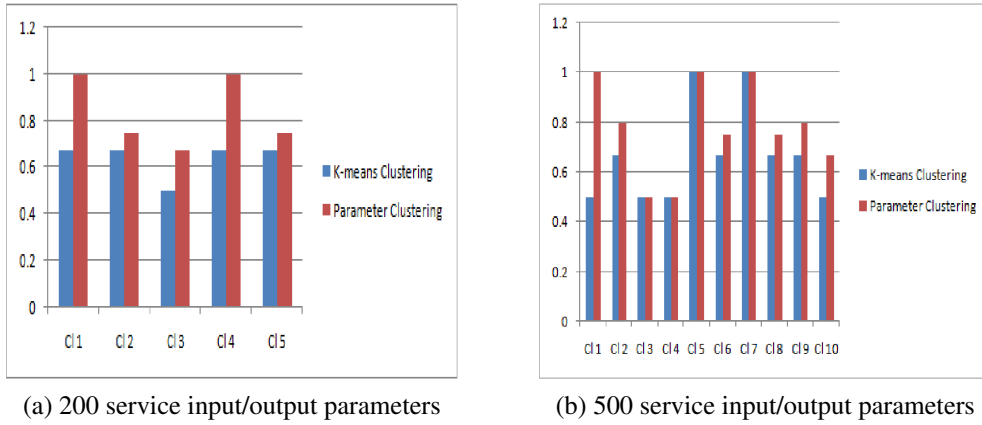


Figure 4.5: Recall of Generated Clusters

Number of input/output parameters	Performance Measure	Algorithm SCSP	Algorithm K-means
200	Precision	0.672	0.54
200	Recall	0.834	0.636
500	Precision	0.66	0.38
500	Recall	0.777	0.668

Table 4.3: Average Precision and Recall of Clusters

#### 4.4.3 Web services clustering results

The effectiveness of frequent output parameter pattern based clustering method is shown by conducting two sets of experiments:

1. Performance analysis of our clustering approach.
2. Performance improvement obtained using clustering for output parameter based service search.

##### 4.4.3.1 Performance Analysis

In this section, we analyze the performance of Algorithm *FOPC* in terms of number of clusters in the final *covering cluster, CC*, and the average cluster overlap in *CC*. The number of web services were varied from 100 to 500 and a constant value of 0.01 was used for *MinSupp*.

#### 4.4.3.2 Impact of Preference factor on Number of Clusters

We have compared the number of clusters obtained in the final  $CC$  with respect to the value of *preference factor*. The results for Algorithm  $FSOPC$  for 100 web services and 500 web services are shown in figure 4.6.

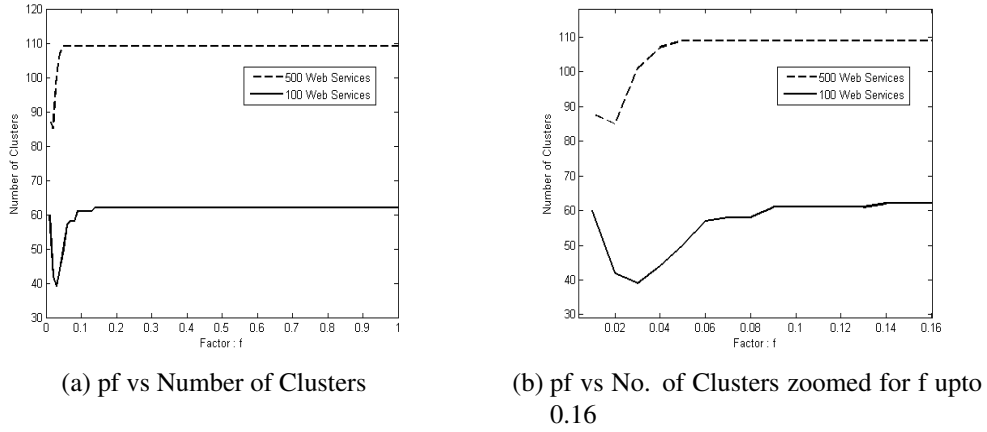


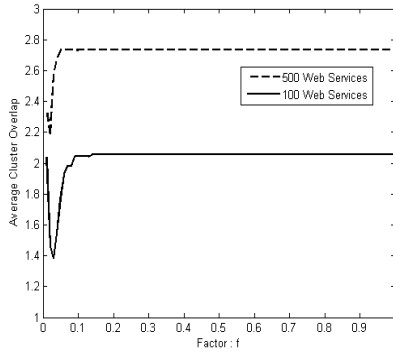
Figure 4.6: Preference Factor vs Number of Clusters in CC

#### 4.4.3.3 Impact of Preference Factor on Cluster Overlap

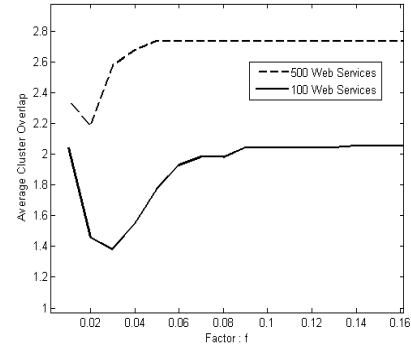
Next, we compared the average *cluster overlap* in final  $CC$  with respect to the value of *preference factor*. The results are shown in figure 4.7. It is seen that average cluster overlap in both the cases (100 web services and 500 web services) decreases as the *preference factor* increases upto a value of around 0.03 and increases thereon, as visualized in 4.7b. Also the value of average cluster overlap remains constant after the value of *preference factor* crosses 0.05 for 500 web services and 0.09 for 100 web services, as seen in 4.7a.

From the results obtained, we can infer that the *preference factor* is indeed required to have a control on the number of clusters obtained by algorithm  $FOPC$ , but as the value of *preference factor* crosses a cut-off point it will have the same effect on number of clusters created. Also the value of *preference factor* affects the *cluster overlap* among the clusters generated by the algorithm. The cut-off point for *preference factor* depends





(a) pf vs Average Cluster Overlap



(b) pf vs Avg Cluster Overlap zoomed for f upto 0.16

Figure 4.7: Preference Factor vs Average Cluster Overlap for clusters in CC

on the dataset and the number of web services. From the results obtained and the analysis made above we come to a conclusion that a value between 0.01 and 0.03 is ideal for the *preference factor* and using algorithm *FOPC* with this ideal value we will obtain minimum number of clusters in *CC* with minimum average *cluster overlap*.

#### 4.4.3.4 Performance improvement obtained in output parameter based service search

To improve the performance of parameter based service search, we have proposed an approach to cluster services in the registry on their output parameter patterns as discussed in section 4.3. In the first set of experiments we published 500 web services in our *extended service registry*. We then evaluated the performance of output parameter based search using different output parameter patterns as user queries, in our extended service registry prior to pre-processing.

In the next set of experiments, to evaluate the performance of output parameter based service search using clusters, we first implemented algorithm 3 to obtain clusters of the registered services and then implemented algorithm 4 on these clusters. We then queried our *extended service registry* (covering clusters included) with the same set of

output parameter patterns used for the first set of experiments. The experiment were repeated by publishing 1000 web services in service registry. A comparison of time taken for output parameter pattern based search prior to clustering and after clustering is depicted in figure 4.8. From the graphs obtained we can infer that on integrating our clustering approach into *extended service registry*, there is a substantial improvement in the performance of parameter based service search.

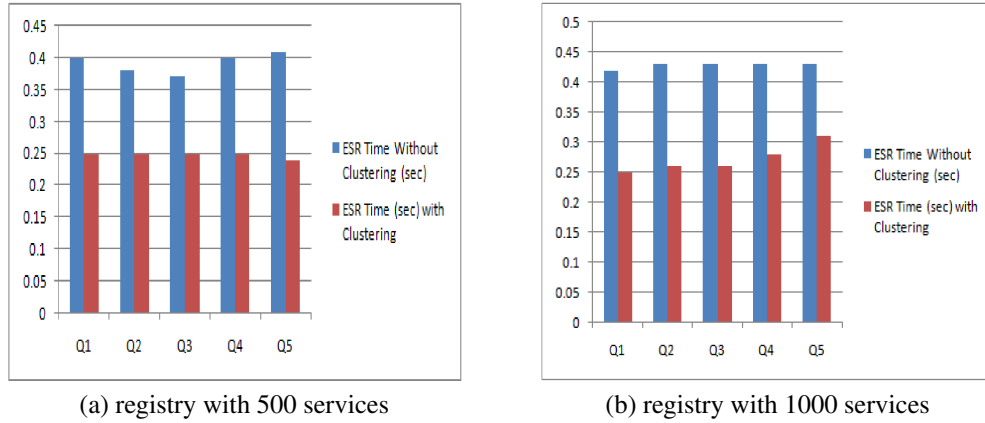


Figure 4.8: Output Parameter based Search Using Clusters

## 4.5 Conclusion

Input/output parameter based web service matching is an important issue, especially for non-semantic web service search and composition. Traditional UDDI based service search lacks the ability to recognize input/output parameters described in WSDL files. For this application semantic based search is wishful for providing a number of alternative web services to a user on need. A cluster of web services is formed on the basis of similarity of their input/output parameters. Then for a given set of query input/output parameters, corresponding set of alternative services can be picked from a cluster that is selected by matching of query input/output parameters to input/output parameters of a cluster.

Thus, in general measuring similarity between two parameters has an importance for preprocessing of service registry to find clusters of services. In order to widen the scope of search for a web service with a given set of input/output parameters, it is useful to cluster services having similar input/output parameters. Hence, we propose an approach for clustering service input/output parameters on their similarity, using WordNet as the underlying ontology.

Algorithm *SCSP* in section 4.2.3 generates parameter clusters of input/output parameters of the web services in a registry. The cluster selection in the algorithm is governed by the values of  $\epsilon$  and *minSP*, as discussed in section 4.2. We have simulated the algorithms on QWS Dataset (Al-Masri and Mahmoud (2008)). Experimental evidence shows that the clusters generated by our algorithm has a better Precision and Recall values when compared with the clusters generated by K-means algorithm (MacQueen *et al.* (1967)).

We utilize these parameter clusters for clustering services with similar input/output parameters to enable efficient parameter based web service search in service registries. Further, we propose an approach to accelerate service search of non-semantic web services by clustering services on their output parameter similarity. Our approach for clustering web services based on *frequent output parameter patterns* looks promising since it provides a natural way of reducing the candidate services when we are looking for a web service with desired *output parameter pattern*.

Algorithm *FOPC* in section 4.3.2 generates a *covering cluster* that covers all the web services in the registry. The cluster selection in the algorithm is governed by *preference factor*, as discussed in section 4.3.1. We have simulated the algorithms on QWS-WSDL dataset (Al-Masri and Mahmoud (2008)). The experimental results demonstrate the performance of our clustering approach on varying user queries.

Algorithm4 for service search discussed in section4.3.3 lists a set of web services that matches the queried output parameters. From the many services that are returned we need to choose the best match for a given user query. Hence, we propose a service selection approach that ranks the set of matching services based on their QoS values in the next chapter. Further a service composition method is proposed that can be used when the service search fails to find a web service matching the user requirements.

# CHAPTER 5

## Parameter based service selection and composition

### Abstract

One critical challenge in web service search and composition is the selection of web services, to be executed or to be composed, from the pool of matching services. Most of the current service selection proposals (Cai and Xu (2014); El Hadad *et al.* (2010); Li *et al.* (2014b); Mobedpour and Ding (2013); Yager *et al.* (2011)) apply a weighted sum model (WSM) as an evaluation method for selection of services with the same functionality. We propose a **bi-level service selection approach** that selects the most appropriate web services from the pool of matching services that considers both the functional and non-functional requirements for service selection. The functional requirements are provided by a user as a set of input parameters provided for and output parameters desired from the web service. The user also provides a set of desired QoS values and the order of their preference for selection.

By service composition, we mean making of a new service (that does not exist on its own) from existing services. Most of the existing algorithms for service composition consider services based on exact matches of input/output parameters for composition. However, for service composition the construction of such a chain of services fails at a point when the output parameters of a preceding service ( $ws_P^O$ ) does not match exactly with the input parameters of a succeeding service ( $ws_S^I$ ). We propose<sup>1</sup> to alleviate this problem by extending the classical definition of service composition to give rise to three types of service composition: *exact composition*, *super composition* and *collaborative composition*. Adopting three types of service composition for a desired service output parameter set, the possibility of having different kinds of compositions is demonstrated in form of a *composition search tree*. Further, we propose<sup>2</sup> the utility of *composition search tree* for finding compositions of interest like *leanest composition* and the *shortest depth compositions*.

---

<sup>1</sup>Lakshmi.H.N, Hrushiksha Mohanty;RDBMS for service repository and composition, Proceedings of 4th International Conference on Advanced Computing, ICoAC 2012, published in IEEE Xplore.

<sup>2</sup>Lakshmi.H.N, Hrushiksha Mohanty,Utility of composition search tree for searching Optimal service compositions, Published in Elsevier Science, proceedings of Eighth International Conference on Data Mining and Warehousing ISBN: 9789351072515, 2014 Pp 36- 42.

## 5.1 Introduction

One of the critical challenges in the area of service search and composition is to define a **service selection approach** that selects the most appropriate web services from a pool of services discovered. Most of the current approaches(Cai and Xu (2014); El Hadad *et al.* (2010); Li *et al.* (2014b); Mobedpour and Ding (2013); Yager *et al.* (2011)), as discussed in detail in chapter 2, select services based on their QoS values from a set of web services that are functionally similar. These approaches usually apply a weighted sum model (WSM) as an evaluation method, represented as -

$$Score(WS) = \sum (q'_i * w_i) \quad (5.1)$$

where  $q'_i$  is a normalized QoS attribute value and  $w_i$  is the weight given to the QoS attribute. Such methods require users to express their preference over different (and sometimes conflicting) quality attributes as numeric weights. User's inability to model their preferences mathematically leads to hard selection that may often lead to failure in selecting best matching services.

On the contrary, in order to model both the functional and non-functional requirements of users, **we propose a bi-level service selection approach**. The functional requirements are provided by the user as a set of input parameters provided for and output parameters desired from the web service. The user also provides a set of desired QoS values and the order of their preference for selection. In first level services matching the functional requirements are shortlisted, which are further filtered in second level based on given QoS requirements, thus providing a list of web services that best matches a given user query. Experiments were conducted using QWS dataset(Al-Masri and Mahmoud (2008)) to compare the second level(QoS based selection) of our approach with that of Chen's(Mobedpour and Ding (2013)) approach. Various sets of queries were fed for both the approaches and the results were analyzed on the quality of services selected and the execution time taken by both approaches. From the results obtained we can in-

fer that our approach performs better and returns quality web services as compared with Chen's approach.

A web service,  $ws$ , has typically two sets of parameters - set of inputs  $ws^I$  and set of outputs  $ws^O$ . When a user is looking for a web service for a given input and desired output parameters and there is no single web service in the service registry satisfying the request, service composition becomes necessary. By service composition, we mean making of a new service(that does not exist on its own) from existing services registered in UDDI. Conventionally two services  $ws_i$  and  $ws_j$  are said to be composable iff  $ws_i^O = ws_j^I$ , i.e,  $ws_j$  receives all the required inputs from outputs  $ws_i$  has. A service composition is formed by constructing a chain of such composable services.

Most of the existing algorithms for service composition(Kwon *et al.* (2007); Lee *et al.* (2011); Zeng *et al.* (2010)) consider services based on Exact matches of input/output parameters for composition. However, for service composition the construction of such a chain of services fails at a point when the output parameters of a preceding service ( $ws_P^O$ ) does not match exactly with the input parameters of a succeeding service( $ws_S^I$ ). In chapter 3 we proposed three types of service match : *exact match*, *super match* and *partial match*, extended from the conventional definition of a service match, to alleviate this problem.

Utilizing these types of matches, in this chapter **we widen the scope of composition by defining possibly three types of service composition: *exact, collaborative and super composition***. For a given user query, when service selection fails to find a matching service, we propose to find the possible compositions that satisfy the given user query. **The process of service composition is visualized as generation of a *composition search tree*** that arranges services in levels showing the way service compositions can be made to meet the user requirements, i.e, such a tree can be viewed as a result to consumer query. Further, **we propose the utility of *composition search tree* for finding**

**best service compositions like *leanest composition* and *shortest depth composition*.**

We first describe our bi-level model for service selection in section 5.2. In section 5.3 that follows, we explain the service composition process and propose an algorithm for construction of *composition search tree*. Further utility of *composition search tree* for finding best compositions like *shortest depth composition* and *leanest composition* is explained in detail in section 5.4. The experimental results of our service selection approach is discussed in section 5.5. Conclusion is given in section 5.6.

## **5.2 I/O parameter and QoS based service selection**

There are two kinds of requirements that are crucial to web service selection and composition: functional and non-functional requirements. Functional requirements focus on functionality of the selected service, whereas the non-functional requirements are concerned with the quality of service (QoS).

We propose a bi-level model that considers both the functional and non-functional requirements for service selection. The functional requirements are provided by the user as a set of input parameters provided for and output parameters desired from the web service. The user also provides a set of desired QoS values and the order of their preference for selection. In the proposed bi-level model, the two objective functions: functional match and non-functional match, are arranged in two levels according to their order of importance. The first level short lists a set of web services that optimizes the functional requirements from which services that best matches the QoS requirements are selected in the second level.

In the first level, we propose to compute input and output parameter deviation of a matched web service with respect to query input and output parameters using weighted



sum model. This computation is done for all matching web services and is utilized for ranking them on functional match. Web services with lesser deviation values are shortlisted and considered in the second level, where further selection is done based on QoS values of these services. In the second level we consider 4 QoS attributes: response time, reliability, availability and price to rank web services. These attributes are modeled as constraints to be satisfied. For selection of services for composition we propose a  $\epsilon$ -constraint method for ranking services. Figure 5.1 depicts the bi-level service selection approach discussed above. Each of these levels is explained in detail in the following subsections.

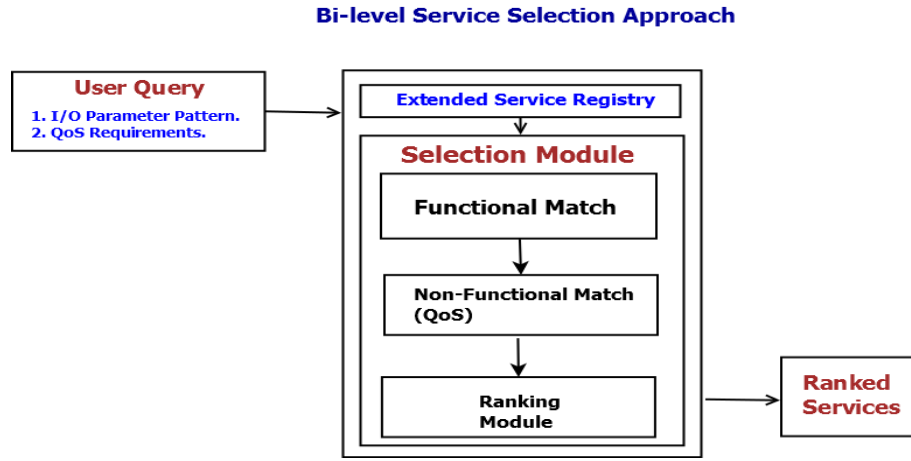


Figure 5.1: Bi-level service selection approach

### 5.2.1 Functional Match

The first objective of our bi-level model is to select web services that best match the given functional requirements of the user. The functional requirements are specified as a set of input parameters (the user is providing) and output parameters (that the user expects). We introduce a *deviation measure* for both input and output parameters that measures the deviation in the parameter set of the matched web service with respect to those provided by the user. The input and output parameter deviations are combined using weighted sum method and the values obtained is used to rank the available matching

web services for further shortlisting. Higher the value of the deviation measure, lesser will be the rank of the matched web service.

### 5.2.1.1 Output parameter deviation measure

Here we describe the method to compute the *output parameter deviation measure*. There are 3 types of output parameter matches : exact, super and partial, as explained earlier in chapter 3. The following notations are used for defining output parameter deviation measure:

- Let  $WS_D$  denote web service with desired requirements.
- Let  $WS_M$  denote the matched web service.
- Let  $WS_D^O$  denote desired output parameter set.
- Let  $WS_M^O$  denote output parameter set of matched web service.

The number of non-matched output parameters(NMOP) is given by,

$$NMOP = |WS_D^O| - |WS_M^O| \quad (5.2)$$

Using eqn.5.2 we determine the type of output parameter match, as follows :

1.  $NMOP = 0$  for exact and super match.
2.  $NMOP > 0$  for partial match.

*Measuring deviation from  $WS_D^O$  :*

The output parameter set of the matched web service,  $WS_M^O$ , is compared with the desired set,  $WS_D^O$ , and by using eqn.5.2 the type of match is determined. Then the *output parameter deviation measure*,  $DM_P^O$ , for the matched web service is calculated, depending on the type of match, as follows :

1. Exact match : For a web service that matches exactly,  $DM_P^O = 0$ , since there is no deviation and hence no ordering is required.
2. Super match : For a web service that is a super match of the desired web service, deviation is measured in terms of number of parameters that are redundant in the output parameter set of the matched service  $WS_P^O$ , and is given by -

$$DM_P^O = |WS_M^O - WS_D^O| \quad (5.3)$$

3. Partial match : For a web service that is a partial match of the desired web service, deviation is measured in terms of number of parameters not provided by output parameter set of the matched service,  $WS_M^O$ , and is given by -

$$DM_P^O = |WS_D^O| - |WS_M^O| \quad (5.4)$$

### 5.2.1.2 Input parameter deviation measure

Here we describe the method used to compute the *input parameter deviation measure*. There are 3 types of input parameter matches : exact, super and partial, as explained earlier in chapter 3. The following notations are used for defining input parameter deviation measure:

- Let  $Q^I$  denote query input parameter set provided by the user.
- Let  $WS_M^I$  denote input parameter set required by the matched web service.

The number of non-matched input parameters (NMIP) is given by -

$$NMIP = |WS_M^I| - |Q^I| \quad (5.5)$$

Using eqn.5.5 we determine the type of input parameter match, as follows :

1.  $NMIP = 0$  for exact and super match.

2.  $NMIP > 0$  for partial match.

*Measuring deviation from  $Q^I$  :*

The input parameter set of the matched web service,  $WS_M^I$ , is compared with the query input parameter set,  $Q^I$ , and by using eqn.5.5 the type of match is determined. Then the deviation measure,  $DM_P^I$ , for the matched web service is calculated, depending on the value of  $NMIP$ , as follows :

1. Full match : The value of  $NMIP = 0$  for a full match, which implies that the input parameters required by the matched web service is satisfied by  $Q^I$ . This is possible when  $Q^I \supseteq WS_M^I$ . Hence

$$DM_P^I = NMIP = 0. \quad (5.6)$$

2. Partial match : The value of  $NMIP < 0$  for a partial match , which implies that the input parameters required by the matched web service is not completely provided by  $Q^I$ . This case is encountered when  $Q^I \subset WS_M^I$ . Hence

$$DM_P^I = NMIP > 0. \quad (5.7)$$

### 5.2.1.3 Combining input/output deviation measures

We use *weighted sum method for combining output and input parameter deviation measures*, as follows :

- Let  $x_1 = DM_P^O$ , the output parameter deviation measure.
- Let  $x_2 = DM_P^I$ , the input parameter deviation measure.

Then, the total deviation of the matched web service,  $WS_M$ , is given by -

$$\begin{aligned}
 DM_P^{IO} &= w_1 * x_1 + w_2 * x_2, \\
 \text{where, } w_1 &= \frac{1}{|WS_M^O|} \text{ for exact and super output match,} \\
 w_1 &= \frac{1}{|WS_D^O|} \text{ for partial output match} \\
 \text{and } w_2 &= \frac{1}{|WS_M^I|} \text{ for both full and partial match}
 \end{aligned} \tag{5.8}$$

The value for  $DM_P^{IO}$  is computed for all the matching web services and is utilized for ranking them on their functional match. Since the value represents the amount of deviation in input and output parameter set of the matched web service with respect to queried input and output parameter set, its obvious that lesser the value higher will be the matching and hence higher the rank. Web services with lesser deviation values are shortlisted and considered in the second level, where further selection is done based on the QoS values of these services.

### 5.2.2 QoS based service selection

The second objective of our bi-level model is to select web services that match best the given non-functional requirements of the user. After shortlisting the matched web services considering their functional match, they are now further ranked considering their QoS values. The user is expected to provide a set of desired QoS values which will be considered in this level. The objective of this level is to list a set of web services that best match with the desired QoS values. We consider 4 QoS attributes : response time, reliability, availability and price in our model for ranking web services.

We assume that the service provider provides the values of web services QoS attributes and also update their value often. These values are stored in QoSTable in our extended service registry as explained previously in chapter 3. QoS attributes are either

*positive*, for which higher values indicates better quality, Eg: availability, reliability, etc or *negative*, for which lower values indicate better quality, Eg: price, response time, etc. We present a QoS model taking into consideration these aspects, for service selection in the next subsection.

### 5.2.2.1 QoS model used

Table 5.1 shows the QoS model used in our approach. Each of the QoS attribute is explained briefly.

Dimension	Attribute	Definition
Performance	Response Time	Execution Time(WS) + Waiting Time(WS)
Dependability	Reliability	1-Failure Rate(WS)
	Availability	$\text{Uptime(WS)} / (\text{Uptime(WS)} + \text{Downtime(WS)})$
Cost	Price	Execution Fees(WS)/Request

Table 5.1: QOS Model of web service

1. *Response time* ( $q_{RT}$ ) : Evaluating a service's response time to a request typically comprises of measurement of the execution time and waiting time of the web service. It is measured as the time between sending a service request and receiving a response.
2. *Reliability* ( $q_R$ ) : Reliability refers to the service provider's ability to successfully deliver requested service functionality. This ability can be quantified by the probability of success in a service execution, but it is usually evaluated through the service failure rate. This rate is calculated as the ratio of execution time and mean time between failures (MTBF).
3. *Availability* ( $q_A$ ) : Availability of a web service is the degree to which a service is operational and accessible when it is required for use. This value is defined by the proportion of the service's uptime to downtime, as represented by the mean time between failures (MTBF) and mean time to recovery (MTTR), respectively.

4. *Price* ( $q_C$ ) : It is the amount of money the requester has to pay for using the service.

### 5.2.2.2 Selection method

Most of the current proposals have applied a weighted sum model (WSM) as a uniform evaluation method for selection of services with the same functionality. This is represented as -

$$Score(WS) = \sum (q'_i * w_i) \quad (5.9)$$

where  $q'_i$  is a normalized QoS attribute value and  $w_i$  is the weight given to the QoS attribute. Such methods require users to express their preference over different (and sometimes conflicting) quality attributes as numeric weights. The objective function assigns a scalar value to each service based on the QoS attribute values and the weights given by the user. The service that has the highest value for the objective function will be selected and returned to the user.

Such optimization techniques are unable to model user preferences precisely. For example, let us assume that the service selection is based on two quality attributes  $q_1$  and  $q_2$  with 0.6 and 0.4 as the associated weights for the objective function. Suppose there are two web services  $w_i$  and  $w_j$  with QoS values as  $\{3, 8\}$  and  $\{5, 5\}$  respectively. The weighted sum model gives a Score of 5 for both  $w_i$  and  $w_j$ . However, from the weights specified by the user, it is quite clear that  $q_1$  needs to be given a greater preference than  $q_2$  and hence  $w_2$  would be the obvious choice.

The shortlisted web services from the first level can be categorized as those that have a deviation measure of 0 (an exact match) and those having a deviation measure  $> 0$  (a partial or super match). When there are no exact matching services available, then service composition becomes inevitable and services need to be selected in each each step of composition process. Hence, in order to model both the qualitative and quantitative

preference of users, we propose a  $\epsilon$ -constraint model Marler and Arora (2004a). The four QoS attributes that we consider, as explained in section 5.2.2.1 are modeled as four objectives. Out of these we choose to minimize the cost and the remaining three objectives: response time, reliability and availability, are constrained to be greater/lesser than or equal to given user values. Formally,

$$\begin{aligned}
& \min q_C(W S_i) \\
& q_{RT}(W S_i) \leq \epsilon_{RT} \\
& q_R(W S_i) \geq \epsilon_R \\
& q_A(W S_i) \geq \epsilon_A
\end{aligned} \tag{5.10}$$

where values for  $\epsilon_{RT}, \epsilon_R$  and  $\epsilon_A$  are the desired QoS values for response time, reliability and availability respectively and are provided by the user. The model selects a web service that has minimum cost with the desired ( or better ) response time, reliability and availability values.

When there are services matching exactly with queried input/output parameters readily available in the registry, then we propose to rank them based on the QoS requirements by modeling the 4 QoS attributes as constraints. This *constraint method* is formally described below,

$$\begin{aligned}
& q_C(W S_i) \leq \epsilon_C \\
& q_{RT}(W S_i) \leq \epsilon_{RT} \\
& q_R(W S_i) \geq \epsilon_R \\
& q_A(W S_i) \geq \epsilon_A
\end{aligned} \tag{5.11}$$

where  $\epsilon_C$  is desired cost of web service and is provided by the user. Web services satisfying these constraints are ranked based on the values of QoS attributes and the user can then select a service from this ranked list of services. However, when a web service that does not match all the QoS requirements of the user is available in the registry, the



selection system returns a list of web services that best approximates the user requirements.

The effectiveness of our QoS based service selection approach is shown by comparing the selected services of our system with the system proposed by Chen and Delnavaz (Mobedpour and Ding (2013)). Experiments were conducted using the QWS dataset (Al-Masri and Mahmoud (2008)) as explained in detail in section 5.5. From the results obtained, we can infer that our approach outperforms Chen's method both in terms of execution time and the quality of services matched.

Service selection approach explained in this section finds the best service matches of all three types. If an exact match is not found for the given query, then it implies that there is no single web service in the extended service registry that satisfies the user query and hence service composition becomes necessary. In the next section we describe our approach for service composition using the various service match types.

### **5.3 Query processing: construction of composition search tree (CST)**

By service composition, we mean making of a new service (that does not exist on its own) from existing services registered in UDDI. Conventionally two services  $ws_i$  and  $ws_j$  are said to be composable iff  $ws_i^O = ws_j^I$ , i.e.,  $ws_j$  receives all the required inputs from outputs  $ws_i$  has. A service composition is formed by constructing a chain of such composable services. However, the making of a service composition chain may fail at a point when the output parameters of a preceding service ( $ws_P^O$ ) does not match exactly with the input parameters of a succeeding service ( $ws_S^I$ ). Three types of service matchings are proposed as explained earlier in chapter 3 to alleviate this problem. Based on these matching types, we propose three types of service compositions, as explained in

detail in section 5.3.1.

Here we describe the process of generating service compositions satisfying a given query. We start by creating a dummy web service  $dws$  that takes the output parameters specified in the query,  $Q^O$ . We match services in the registry with  $dws$  to identify compositions satisfying  $Q^O$ . The compositions are further classified as *collaborative composition*, *exact composition* and *super composition* based on type of composition of services participating in the composition. These compositions would require additional input parameters than those provided in  $Q^I$ . Compositions satisfying these additional input parameters are retrieved from the extended service registry and the process is repeated recursively until there are no additional input parameters required. We propose to construct a *composition search tree* to visualize the composition process and the compositions satisfying a given query.

### 5.3.1 Service composition types

Given a registry  $R = \langle P, W \rangle$ , any desired set of output parameters,  $D^O \subset P$ , can be satisfied by possibly many compositions as discussed in the motivating example in chapter 1. To generate such compositions we start by matching the output of services in the registry,  $ws_i^O$ , with  $D^O$  and classify the services on their matching type as exact, super and partial as explained in chapter 3. We can readily define two types of compositions - exact composition and super composition, from the exact and super matching services as follows -

1. **Exact composition (EC):** Exact composition is a composition obtained by using a web service that is exactly matching with  $D^O$ , i.e.,  $ws_i^O = D^O$ , where  $ws_i \in W$ . Such a composition would require additional input parameters ( $RI_{EC}^I$ ) than those specified in  $Q^I$  given by,

$$RI_{EC}^I = ws_i^I - Q^I$$

where  $ws_i^I$  is input parameters of web service  $ws_i$ . There can be many services in  $W$  that are exactly matching with  $D^O$ , the best matching one among them is selected using the service selection method discussed in section 5.2 in each level to be solved further.

2. **Super composition (SC):** Super composition is a composition obtained by using a web service that has a super match with  $D^O$ , i.e. ,  $ws_i^O \supset D^O$ , where  $ws_i \in W$ . The additional input parameters required by such a composition ( $RI_{SC}^I$ ) is given by,

$$RI_{SC}^I = ws_i^I - Q^I$$

where  $ws_i^I$  is input parameters of web service  $ws_i$ . Among the many services in  $W$  that have a super match with  $D^O$  the best match is selected using the approach discussed in section 5.2 at each level to be solved further.

Most of the existing algorithms for service composition construct chains of services based on exact Matches of input/output parameters to satisfy a given query. However, this approach fails when the available services satisfy only a part of the input/output parameters in the given query. This shortcoming motivated us to define a new type of composition - collaborative composition, that is obtained by using a set of partial matching services. We define collaborative composition as -

**Collaborative composition (CC):** Collaborative composition is a composition obtained by using a set of partial matching services ,  $WS$ , that can collaboratively satisfy the desired set of output parameters  $D^O$ , i.e ,there exists a set of services  $WS_{CC}$  , such that

$$WS_{CC} \subset W, WS_{CC} = \{ws_1, ws_2, \dots, ws_n\}$$

$$\text{where } ws_i^O \subset D^O, \forall ws_i \in WS_{CC}$$

$$\text{and } \{ws_1^O \cup ws_2^O \cup \dots \cup ws_n^O\} \supseteq D^O$$

There can be many such service sets that satisfy  $D^O$ . We choose the best set among them using the selection approach discussed previously in section 5.2. The additional input parameters required ( $RI_{CC}^I$ ) to execute the services in  $WS_{CC}$  is given by

$$RI_{CC}^I = WS_{CC}^I - Q^I - WS_F^O$$

where  $WS_{CC}^I$  is collective input parameters required by the set  $WS_{CC}$ , i.e ,

$$WS_{CC}^I = \{ws_1^I \cup ws_2^I \cup \dots \cup ws_n^I\}$$

and  $WS_F$  is a set of services such that

$$WS_F \subseteq WS_{CC} , WS_F = \{ws_1, ws_2, ..\} \text{ and } \forall ws_j \in WS_F, ws_j^O \subseteq Q^I.$$

### 5.3.2 Process of CST Construction

In order to visualize the composition process and to find possible compositions that satisfy a given user query we propose to construct a *composition search tree*. Every node of the *composition search tree* stores the web service(or set of web services) that satisfies desired output parameters of its parent node ( $WS$ ), number of web services used for composition( $NWS$ ) and set of additional input parameters required by the web service(or set of web services) ( $D^O$ ). Each node in the *composition search tree* has utmost 3 *child nodes*, a *left child node* representing *exact composition*, a *middle child node* representing *super composition* and a *right child node* representing *collaborative composition*.

The structure of a node in *composition search tree* is given by Backus Naur Form(BNF) in Fig 5.2. The abbreviations used in BNF are described in Table 5.2.

The *composition search tree* has 4 types of nodes as described below -

1. **Root node** : A CST node from where the composition process begins, having the following special properties -

Abbreviation	Description
WS	web Service/Set of web services participating in composition.
NWS	Number of web services used in composition
$D^O$	Desired set of output parameters.

Table 5.2: Abbreviations used in BNF

<pre> &lt;CST Node&gt; ::= &lt;CST Data&gt; &lt;CST Pointer&gt; &lt;Node Type&gt;  &lt;CST Data&gt; ::= &lt;WS&gt; &lt;NWS&gt; &lt;D<sup>O</sup>&gt; &lt;Composition Type&gt; &lt;WS&gt; ::= {WS ID}* &lt;NWS&gt; ::= &lt;Integer&gt; &lt;D<sup>O</sup>&gt; ::= {Parameter Symbol}* &lt;CompositionType&gt; ::= &lt;Exact&gt;   &lt;Super&gt;   &lt;Collaborative&gt;   &lt;NIL&gt;  &lt;CST Pointers&gt; ::= &lt;Parent Node&gt; [&lt;Left Child&gt;] [&lt;Middle Child&gt;] [&lt;Right Child&gt;] &lt;Parent Node&gt; ::= &lt;Pointer to CST Node&gt; (* Present except for Root Node *) &lt;Left Child&gt; ::= &lt;Pointer to CST Node Exact&gt; &lt;Middle Child&gt; ::= &lt;Pointer to CST Node Super&gt; &lt;Right Child&gt; ::= &lt;Pointer to CST Node Collaborative&gt;  &lt;Node Type&gt; ::= &lt;Root&gt;   &lt;Internal&gt;   &lt;UnSolvable&gt;   &lt;Solution&gt; </pre>
--

Figure 5.2: BNF of a CST Node

- $\langle WS \rangle = \emptyset$
- $\langle NWS \rangle = 0$
- $\langle D^O \rangle = Q^O$
- $\langle CompositionType \rangle = NIL$
- $\langle ParentNode \rangle = NULL$

2. **Internal node** : A CST node that represents a composition (exact, super or collaborative) satisfying  $D^O$  of its parent node. Every internal node of the *composition search tree* has utmost 3 *child nodes*, a *left child node* representing *exact composition*(EC), a *middle child node* representing *super composition*(SC) and a *right*

*child node* representing *collaborative composition*(CC). Although there may be many compositions in each type : exact, super and collaborative, that satisfy  $D^O$  of the *parent node*, we propose to select one in each type that best matches the requirement using the service selection method described earlier in section 5.2, for every internal node and hence limit the number of children to three. Note that the *root node* is also an internal node with special properties as explained before.

3. **Unsolvable node** : It is a leaf node that cannot be solved further, since  $D^O$  of such a node does not have matching services in the service registry. These type of nodes have the following special properties -

- $\langle CompositionType \rangle = \langle Exact \rangle | \langle Super \rangle | \langle Collaborative \rangle$
- $\langle D^O \rangle = \{parameterSymbol\}^*$
- $\langle LeftChild \rangle = NULL$
- $\langle MiddleChild \rangle = NULL$
- $\langle RightChild \rangle = NULL$

4. **Solution node** : A leaf node that need not be solved further since  $D^O$  of such a node is  $\emptyset$ . These type of nodes represent compositions solving the given user query and have the following special properties -

- $\langle CompositionType \rangle = \langle Exact \rangle | \langle Super \rangle | \langle Collaborative \rangle$
- $\langle D^O \rangle = \emptyset$
- $\langle LeftChild \rangle = NULL$
- $\langle MiddleChild \rangle = NULL$
- $\langle RightChild \rangle = NULL$ .

The following terminologies are used for *composition search tree* construction:

- *Live node*: A node that is unsolved. These are the nodes for which compositions are not yet found and hence needs to be expanded.

- *Current node*: A node that is currently being solved.
- *Solution node*: A node that represents the solution.
- *Unsolvable node*: A node that cannot be solved further.
- *LivenodesQ* : A Queue to store the live nodes.
- *Solutions*: An array that stores all the *Solution Nodes*.

The process for tree construction is given below :

1. Create a *root node* that has desired output parameters equal to the output parameters specified in the query, i.e  $D^O = Q^O$  , initialize the number of web services used in composition,  $NWS$ , to 0 and set of web services participating in composition as empty set,  $WS = \emptyset$ .
2. Insert the *root node* to *LivenodesQ*.
3. Delete a *live node* from *LivenodesQ* and set it as the *current node*.
4. From the *covering clusters* (obtained by clustering services on frequent output parameter pattern as discussed earlier in chapter 4) find services that match with  $D^O$  of the *current node*.
5. Classify these services according to their match type: exact, partial and super ( as discussed in chapter 3).
6. Find different compositions that satisfy  $D^O$  from these services based on their type of composability as -
  - (a) If there exists a exact matching service  $ws$  for  $D^O$  in service repository, then create a *left child node* for the *current node*, store  $ws$  and update  $NWS$  as  $NWS = NWS + 1$ . If there are many exact matching services, then the best matching service is chosen using the service selection approach discussed

in section 5.2.2.2. Calculate the additional input parameters required, to execute  $ws$ , as  $R_{EC}^I = ws^I - Q^I$ , where  $ws^I$  is input parameters of web service  $ws$ .  $R_{EC}^I$  is the new set of desired output parameters that need to be satisfied, i.e.,  $D^O = R_{EC}^I$ . If  $D^O \neq \emptyset$  then *Insert* the *left child node* to *LivenodesQ*, otherwise mark the *left child node* as *solution node* and *Insert* a copy of the node to *Solutions*. Make the *left child node* point to its *parent node*.

- (b) If there exists a super matching service  $ws$  for  $D^O$  in service repository, then create a *middle child node* for *current node*, store  $ws$  and update  $NWS$  as  $NWS = NWS + 1$ . If there are many super matching services, then the best matching service is chosen using the service selection approach discussed in section 5.2.2.2. Calculate the additional input parameters required, to execute  $ws$ , as  $R_{RC}^I = ws^I - Q^I$ , where  $ws^I$  is input parameters of web service  $ws$ .  $R_{RC}^I$  is the new set of desired output parameters that need to be satisfied, i.e.,  $D^O = R_{RC}^I$ . If  $D^O \neq \emptyset$  then *Insert* the *middle child node* to *LivenodesQ*, otherwise mark the *middle child node* as *solution node* and *Insert* a copy of the node to *Solutions*. Make the *middle child node* point to its *parent node*.
- (c) Among services that have partial match with  $D^O$ , search for a set of services that can collaboratively satisfy  $D^O$ . If such a set,  $WS$ , is available, create a *right child node* for the *current node*, store  $WS$  and update  $NWS$  as  $NWS = NWS + |WS|$ . Calculate the additional input parameters required, if any, to execute the services in  $WS$ , as  $R_{CC}^I = WS^I - Q^I - WS_F^O$  where  $WS^I$  is collective input parameters required by the set  $WS$ , i.e.,  $WS^I = \{ws_1^I \cup ws_2^I \cup \dots \cup ws_n^I\}$  and  $WS_F$  is a set of services such that  $WS_F \subseteq WS$ ,  $WS_F = \{ws_1, ws_2, \dots\} | \forall ws_j \in WS_F, ws_j^O \subseteq Q^I$ .  $R_{CC}^I$  is the new set of desired output parameters that need to be satisfied, i.e.,  $D^O = R_{CC}^I$ . If  $D^O \neq \emptyset$  then insert the *right child node* to *LivenodesQ*, otherwise mark the *right child node* as *solution node* and *Insert* a copy of the node to *Solutions*. Make the *right child node* point to its *parent node*.



(d) If  $D^O$  cannot be satisfied by any of the above 3 cases then mark *current node* as *unsolvable node*.

7. Delete a *live node* from *LivenodesQ* and set it as the *current node*.

8. From the *service clusters* find services that match with  $D^O$  of the *current node*.

9. Repeat steps 5 to 8 until the *LivenodesQ* becomes empty.

### 5.3.3 CST Construction Algorithm

The process of constructing the *composition search tree* is given in Algorithm 5. The various notations used in the algorithm is described in Table5.3.

Notation	Description
$Q^O$	Set of output parameters desired in the searched web service as given in the user query.
$Q^I$	Set of input parameters that the user is capable of providing as given in the user query.
WS	web service/Set of web services satisfying the $D^O$ of the <i>Parent node</i> .
NWS	Number of web services participating in the service composition.
$D^O$	Desired set of output parameters.
$R_{EC}^I$	Additional input parameters required to execute the Exact matching web service.
$R_{SC}^I$	Additional input parameters required to execute the Super matching web service.
$R_{CC}^I$	Additional input parameters required to execute the web services participating in Collaborative composition.

Table 5.3: Notations used in CST Algorithm

Figure 5.3 depicts *composition search tree* generated when the web services in table5.4 are used to construct the tree for a query with  $Q^I = \{Date, City\}$  and  $Q^O = \{HotelName, FlightInfo, CarType, TourCost\}$ .

**Input:**  $WSInOutTable$ , *Covering clusters*,  $Q^O$ ,  $Q^I$   
**Output:** *composition search tree*

- 1 Create a *RootNode* with  $D^O = Q^O$
- 2  $NWS = 0$ ,  $WS = \emptyset$
- 3 Insert the *RootNode* to *LivenodesQ*
- 4 *CurrentNode* = *LivenodesQ.Delete()*
- 5 Retrieve matching services from *service clusters* for *CurrentNode* // search for Matching services
- 6 Classify the compositions
- 7 **repeat**
- 8   **if** *ExactComposition* **then**  
     // Left child for EC  
     Create *LeftChild* for *CurrentNode*  
      $WS = ws$   
      $NWS = NWS + 1$ ,  $D^O = RI_{EC}^I$   
     **if**  $D^O \neq \emptyset$  **then**  
         *LivenodesQ.Insert(LeftChild)*
- 14   **else**  
     Mark the *LeftChild* as *SolutionNode*  
     Insert a copy of *LeftChild* to *Solutions*  
     Make the *LeftChild* point to its *ParentNode*
- 18   **if** *SuperComposition* **then**  
     // Middle child for SC  
     Create *MiddleChild* for *CurrentNode*  
      $WS = ws$   
      $NWS = NWS + 1$ ,  $D^O = RI_{SC}^I$   
     **if**  $D^O \neq \emptyset$  **then**  
         *LivenodesQ.Insert(MiddleChild)*
- 24   **else**  
     Mark the *MiddleChild* as *SolutionNode*  
     Insert a copy of *MiddleChild* to *Solutions*  
     Make the *MiddleChild* point to its *ParentNode*
- 28   **if** *CollaborativeComposition* **then**  
     // Right child for CC  
     Create *RightChild* for *CurrentNode*  
      $WS = WS_{CC}$   
      $NWS = NWS + |WS_{CC}|$ ,  $D^O = RI_{CC}^I$   
     **if**  $D^O \neq \emptyset$  **then**  
         *LivenodesQ.Insert(RightChild)*
- 34   **else**  
     Mark the *RightChild* as *SolutionNode*  
     Insert a copy of *RightChild* to *Solutions*  
     Make the *RightChild* point to its *ParentNode*
- 38   **if** *No Compositions* **then**  
     Mark *CurrentNode* as *UnsolvableNode*  
     *CurrentNode* = *LivenodesQ.Delete()*
- 41   Extract compositions from *Covering clusters* // Solve for Additional input parameters
- 42 **until** *LivenodesQ* is not empty

**Algorithm 5:** Composition Search Tree Construction

Service No	Service Name	input parameters	output parameters
ws1	HotelBooking	Period, City	HotelName, HotelCost
ws2	AirlineReservation	Date, City	FlightInfo, FlightCost
ws3	TaxiInfo	Date, City	CarType, TaxiCost
ws4	DisplayTourInfo	HotelName, FlightInfo, CarType	TourInfo
ws5	TaxiReservation	CarType, Date, City	TaxiCost
ws6	TourPeriod	Date, City	Period
ws7	TourCost	TourInfo	TourCost
ws8	AgentPackage	PackageID	Period, TourInfo
ws9	TourPackages	Date, City	PackageID
ws10	TourReservation	Period, TourInfo	HotelName, FlightInfo, CarType, TourCost
ws11	PackageDetails	PackageID	HotelName, HotelCost, FlightInfo, FlightCost, CarType, TaxiCost, TourCost

Table 5.4: Example web services

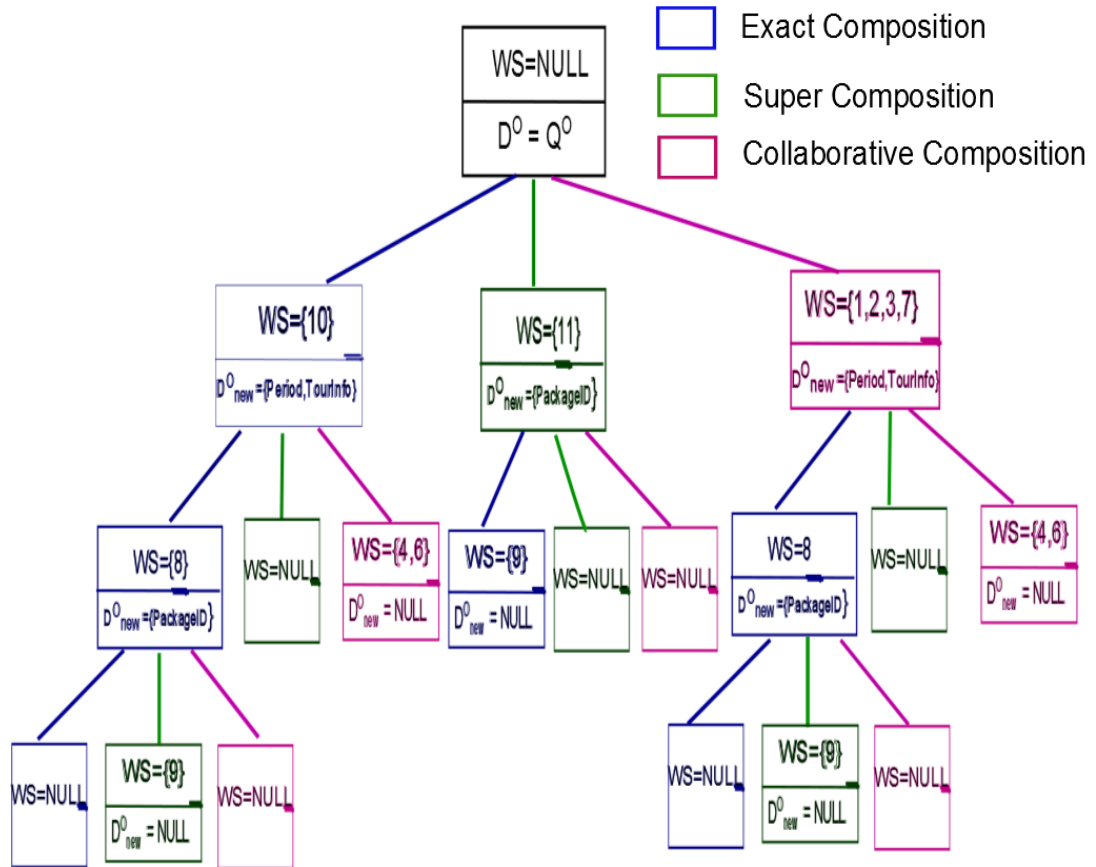


Figure 5.3: Example CST

### 5.3.4 Finding a composition

The *composition search tree* finds all compositions satisfying the given query as explained in the previous section. The array *Solutions* stores all *solution nodes* in the *composition search tree*. Compositions satisfying the given query can be obtained by tracing back from a *solution node* till the *root node* in the *composition search tree*. The pointers stored in each node in the *composition search tree* is utilized in this process. The set of web services required for service composition satisfying given query for a *solution node* can be obtained using procedure *WebServicesInComposition*.

Algorithm 6 lists all the solutions for a given user query obtained from *composition search tree* generated for the query. Each solution is a set of web services that needs to be composed in an order, to obtain the desired output parameters utilizing input parameters provided by the user. These services also satisfy the QoS requirements of user query. The algorithm takes *Solutions* array as input and for each *solution node* in the array it calls procedure *WebServicesInComposition* which lists web services participating in service composition.

```
Input: SolutionNode
Output: SWS for service composition
// SWS is a set of required services for composition
1 SWS =  $\emptyset$ .
2 CurrentNode = SolutionNode
3 while CurrentNode  $\neq$  RootNode do
    // WS is the set of services in the CurrentNode.
4   SWS = SWS  $\cup$  WS
5   CurrentNode = CurrentNode.ParentNode
6 return [SWS]
```

#### **Procedure** *WebServicesInComposition*

From the CST depicted in figure 5.3, we have 5 solutions for the given query, listed below in table 5.5.

<b>Input:</b>	<i>Solutions</i>
<b>Output:</b>	<i>SWS</i> for all solutions in CST
	// <i>SWS</i> is a set of required services for composition
1	<i>SWS</i> = $\emptyset$ .
2	<b>foreach</b> <i>SolutionNode</i> in <i>Solutions</i> <b>do</b>
	// Call procedure <i>WebServicesInComposition</i> for each <i>SolutionNode</i>
3	<i>SWS</i> = <i>WebServicesInComposition</i> ( <i>SolutionNode</i> )
4	Print <i>SWS</i>

**Algorithm 6:** All Solutions in CST

Sl No	Web services in composition	Composition types involved
1	ws9,ws8,ws10	Exact, Super
2	ws4,ws6,ws10	Exact, Collaborative
3	ws9,ws11	Super
4	ws9,ws8,ws1,ws2,ws3,ws7	Collaborative, Super
5	ws4,ws6,ws1,ws2,ws3,ws7	Collaborative

Table 5.5: Solutions in CST

## 5.4 Finding best composition

As discussed earlier the *composition search tree* not only finds all possible compositions satisfying a given query but can also be utilized for querying for optimal service compositions. Here, we define two criteria for choosing best service composition. The criteria are:

1. **Shortest depth composition** - A service composition satisfying a given query that has minimum depth in the *composition search tree* is called *shortest depth composition*. The *shortest depth composition* is a best composition in that it has least depth in *composition search tree*. The process for searching a *shortest depth composition* in *composition search tree* is given in Algorithm 7. Breadth first search is used in the algorithm and the first *solution node* encountered in the

search is returned as the *shortest depth composition*. Solutions 2,3 and 5 in table5.5 are shortest depth compositions satisfying the given user query.

<p><b>Input:</b> <i>composition search tree</i></p> <p><b>Output:</b> <i>SolutionNode</i> for Shortest depth composition</p> <p>// Initialization Steps</p> <p>1 <i>CurrentNode</i> = <i>RootNode</i> of <i>CST</i></p> <p>2 <i>Level</i> = 0</p> <p>3 <i>CArr</i> and <i>NLCArr</i> are arrays of <i>ChildNodes</i></p> <p>4 Insert all <i>ChildNodes</i> of <i>CurrentNode</i> to <i>CArr</i></p> <p>5 <b>while</b> <i>CArr</i> is not empty <b>do</b></p> <p>6     <i>Level</i> = <i>Level</i> + 1</p> <p>7     <b>foreach</b> <i>ChildNode</i> in <i>CArr</i> <b>do</b></p> <p>8         <i>CN</i>=<i>ChildNode</i></p> <p>9         <b>if</b> <i>CN</i> is an <i>SolutionNode</i> <b>then</b></p> <p>10             <b>return</b> [<i>CN</i>]</p> <p>11         <b>if</b> <i>CN</i> is an <i>UnsolvableNode</i> <b>then</b></p> <p>12             Go To 6</p> <p>13         <b>else</b> <i>CN</i> is an <i>UnResolvedNode</i></p> <p>14             Insert all <i>ChildNodes</i> of <i>CN</i> to <i>NLCArr</i></p> <p>15     <i>CArr</i>=<i>NLCArr</i></p> <p>16     <i>NLCArr</i>=0</p> <p>17 <b>return</b> [<i>NULL</i>] // Composition Not Found</p>
--

**Algorithm 7:** Searching shortest depth composition

2. **Leanest composition** - A service composition that requires minimum number of web services to satisfy a given query is called *leanest composition*. The *leanest composition* is an optimal composition in that it uses least number of services possible for service composition. The process for searching a *leanest composition* in *composition search tree* is given in Algorithm 8. Breadth first search is used in the algorithm and the *solution node* with least number of web services participating in composition is returned as the *leanest composition*. Solution 3 in table5.5 is a leanest composition satisfying the given user query.

```

Input: composition search tree
Output: SolutionNode for Leanest composition
1 CurrentNode = RootNode of CST , Level = 0
2 CArr and NLCArr are arrays of ChildNodes
   // LCN is leanest composition SolutionNode
3 LCN = NULL
4 minNWS =  $\infty$  // Current minimum number of services
5 Insert all ChildNodes of CurrentNode to CArr
6 while CArr is not empty do
7   Level = Level + 1
8   foreach ChildNode in CArr do
9     CN = ChildNode
10    if CN is an SolutionNode then
11      // NWS is number of web services used
12      if Level = NWS then
13        if NWS < minNWS then
14          // CN is the shortest composition
15          LCN = CN
16          return [LCN]
17        else
18          if LCN  $\neq$  NULL then
19            // LCN already present
20            return [LCN]
21          else
22            if NWS < minNWS then
23              // A shorter composition
24              LCN = CN
25    if CN is an UnsolvableNode then
26      Go To 7
27    else // CN is an UnResolvedNode
28      Insert all ChildNodes of CN to NLCArr
29    CArr = NLCArr
30    NLCArr = 0
31 if LCN  $\neq$  NULL then
32   return [LCN]
33 else
34   return [NULL] // Composition Not Found

```

**Algorithm 8:** Searching leanest composition

### 5.4.1 Observations

The following observations can be made from the algorithms for *leanest composition* and *shortest depth composition* -

- **Observation 1:** A *solution node* representing *shortest depth composition* also represents a *leanest composition* if it appears at a *level i* that is equal to *NWS*.

**Rationale:** This observation can be deduced from line number 10 in algorithm 7 and line number 13 in algorithm 8. Line number 10 in algorithm 7 always returns the first *solution node* as obtained in the breadth first search of the *composition search tree*. If this *solution node* has the property that it appears at a *level i* that is equal to *NWS*, then this node will be returned as *solution node* from line number 13 in algorithm 8, since this node will have the least *NWS* among all *solution nodes*. Solution 3 in table 5.5 is an example of a shortest depth composition that is also a leanest composition.

- **Observation 2:** A *solution node* represents a *leanest composition* iff there are no other *solution nodes* in the *composition search tree* that has a lesser *NWS* than this *solution node*.

**Rationale:** This observation can be deduced from line numbers 13 and 20 in algorithm 8. These statements search for the *solution node* with the least *NWS* and hence the algorithm always returns a *solution node* that has the least *NWS*.

## 5.5 Experimental results

The effectiveness of proposed QoS based service selection approach is shown by comparing the selected services of our system with the system proposed by Chen and Delnavaz (Mobedpour and Ding (2013)). We compare the two approaches with respect to the following :



1. Number of exact/super service matches obtained w.r.t. user-specified QoS ranges.
2. Number of partial service matches obtained w.r.t. user-specified QoS ranges.
3. Performance in terms of average running time of both the algorithms.

#### **5.5.0.1 Experimental setup**

We conducted experiments on QWS Data set (Al-Masri and Mahmoud (2008)), which includes WSDLs and QoS information of 2507 web services. We ran our experiments on a 1.3GHz Intel machine with 4 GB memory running Microsoft Windows 7. Our algorithms were implemented using Oracle 10g and JDK 1.6. Each query was run 5 times and the results obtained were averaged, to make the experimental results more sound and reliable.

#### **5.5.0.2 Quality of service matches**

In this section, we analyze the quality of services selected in our algorithm versus those selected in Chen's (Mobedpour and Ding (2013)) approach. Chen Ding (Mobedpour and Ding (2013)) propose a selection model capable of handling both exact and fuzzy requirements. The model returns two categories of matching web services: super-exact and partial matches, which are ranked based on relaxation orders and then preference orders of the QoS attributes provided by the user, using MIP as the base algorithm. Symbolic dynamic clustering algorithm (SCLUST) is used to cluster services into 3 groups: good, medium, and poor, based on the values of QoS attributes of the web services.

For comparison of our method with Chen's (Mobedpour and Ding (2013)) approach we consider only the second level of our proposed service selection method, QoS based service selection, as discussed in section 5.2.2 since Chen's approach does a keyword based service search. We implemented keyword based service search in our *extended service registry* and then applied the proposed QoS based service selection approach.

Since this is a selection approach for searching atomic services, we model the QoS attributes as constraints as given in eqn.5.11.

To analyze correctness of both the methods, we count the number of web services that have an exact/super match and partial match with respect to the QoS ranges specified by the user. We check the number of matching web services available in the registry for a given user query manually and compare this with the results of both the methods. Our experimental results shows the web services obtained for 4 different keywords - Google, Commerce, Business and Flight. The QoS requirements fed were as follows :

- Cost below 100.
- Reliability between 50 and 100%.
- Response time below 200ms.
- Availability between 50 and 100%.

From the results obtained, as depicted in figure5.4, we can infer that our constraint based approach retrieves matching services better than Chen's method, in terms of number of matching services retrieved v/s the number of matching services available in the registry. The accuracy in our method is due to the ORDBMS schema used for storing services and their QoS details, whereas the clustering method adopted in Chen's approach might miss some matching services as seen in results.

### **5.5.0.3 Performance comparison**

Next, we compare the average execution time taken by our proposed selection method with that of Chen's(Mobedpour and Ding (2013)) method. The time taken for the set of queries explained in section.5.5.0.2 were noted for both the approaches and tabulated as shown in fig.5.5. From the results obtained, we can infer that our approach outperforms Chen's method both in terms of execution time and the quality of services matched.

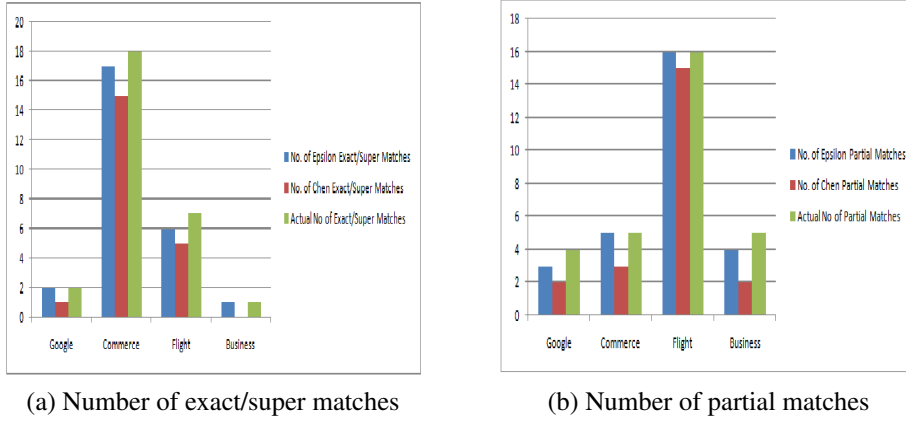


Figure 5.4: Service matches of Constraint method v/s Chen's method v/s available services in registry

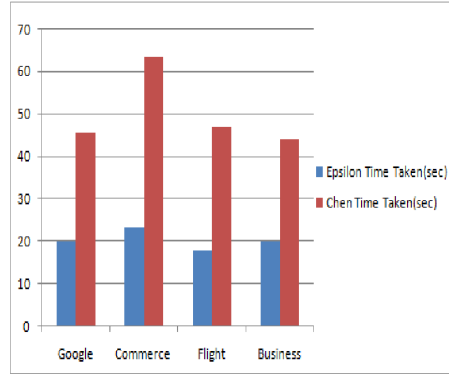


Figure 5.5: Performance comparison of Constraint method v/s Chen's method

## 5.6 Conclusion

One of the critical challenges in web service search and composition is the selection of web services, to be executed or to be composed, from the pool of matching services. Here, we propose a **service selection approach**, as explained in section 5.2 that selects the most appropriate web services from the pool of matching services based on a **bi-level model that considers both the functional and non-functional requirements for service selection**. Experiments were conducted and the results of our approach were compared with that of Chen's (Mobedpour and Ding (2013)) approach. The experimental results show that our approach outperforms Chen's approach as discussed in section ??.

In chapter 3 we have proposed three types of service matching extended from the conventional definition of a service match. Utilizing these types of matches, in this chapter we widen the scope of composition by defining possibly three types of service composition: *exact*, *collaborative* and *super composition*. Based on composability of all three types and sequencing them differently, *composition search tree* explores possible compositions for a given user requirement. Further, we propose the utility of *composition search tree* for finding best service compositions like *leanest composition* and *shortest depth composition*. The process of constructing *composition search tree* and finding best compositions along with their respective algorithms are explained briefly in section 5.3.

We have implemented a tool that takes a user query with functional requirements (represented by input and output parameters) and non-functional requirements (given as a set of desired QoS attribute values). In the tool we have implemented all the proposed approaches : *extending service registry* (described in chapter 3) , *parameter clustering and web service clustering* (described in chapter 4), *service selection* (described in this chapter). On receiving a user query for a service, the tool utilizes all the above approaches and lists matching services if readily available in the extended registry, failing which it lists various service compositions that result in the desired web service. *Composition search tree* is constructed for finding the possible compositions satisfying the given user requirements. The various compositions thus obtained are listed as sets of web services participating in the service composition, along with its type of service composition. The user may thus choose among the various compositions obtained or further query for best compositions satisfying the query. We briefly explain the tool in the next chapter.

# CHAPTER 6

## Tool Implementation

### 6.1 Introduction

The objective of this thesis is to support input/output based web service search and composition. Towards this objective we have proposed strategies for :

1. Extending service registry(ESR).
2. Service search.
3. Service composition.

This chapter implements these strategies so that a user can avail tool support for service search as well as composition. The proposed *ESR Tool* has the following modules that implements the tool functionalities :

1. Pre-processing module - an ORDBMS schema is created for storing web services in the *ESR*.
2. Service search module - parameter clustering, web service clustering and bi-level approach for service search are implemented in this module.
3. Service composition module - contains implementation for *composition search tree(CST)* construction and listing of possible compositions from the CST generated.

The user interface of the tool is implemented in 4 JSP pages, each web page for each stage of service search and composition. The 4 stages of the tool are as follows :

1. User query input.
2. Confirmation of the requirements entered by the user in the previous step.
3. Searching for web services in the *ESR* that satisfy the user requirements.
4. Finding service compositions satisfying the given user query.

In this chapter we first describe the system architecture along with a brief discussion on the various modules of our tool in section 6.2. In section 6.3 we demonstrate the usage of our tool with two example user queries. The screen-shots obtained for the examples are also discussed in this section. Conclusion is given in section 6.4.

## 6.2 System Architecture

In this section, we shall describe the core components of the our *ESR System*, an architectural overview is depicted in Figure 6.1. Service providers register web services into the UDDI registry server. The WSDL description of web services in UDDI is pre-processed to extract information of the web services and stored across various tables in our proposed ORDBMS schema as explained earlier in chapter 3. When a user query is given, the service search module searches for services matching the requirements of user query from *ESR* using the approaches discussed in Chapter 4. These best matching services among those listed are selected using the service selection approach discussed in chapter 5. When there are no readily available services satisfying the user query in the *ESR*, then the possible service compositions satisfying the user query are found in the service composition module, where the approaches discussed in chapter 5 are implemented. Each of these modules are explained further in subsections that follow.

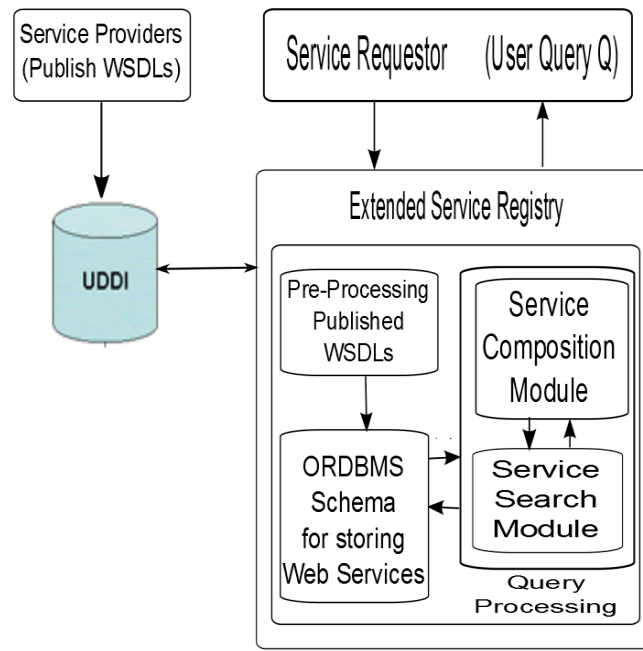


Figure 6.1: ESR System Architecture

### 6.2.1 Pre-processing Module

In the pre-processing module we extract information of web services (published in UDDI) like service name, service port, service provider, service input and output parameters from the WSDLs of the respective web services and store them across various tables in our *ESR* as explained previously in chapter 3. Service QoS attribute values as published by the provider is stored in *QoSTable* in *ESR*.

Figure 6.2 shows the class diagram for this module. There are three classes defined in this module: *WSDLToDB*, *MyDataSetToDB* and *QOSDataToDB*. The class *WSDLToDB* contains methods for parsing a WSDL document and extracting the service details such as service name, service provider, service URL, input and output parameters, etc. These methods are utilized in class *MyDataSetToDB* where information is extracted for all WSDL files of services registered in ESR and are stored in respective tables of ESR.

The major steps involved in this class is depicted as a sequence diagram in fig-

ure.6.3. Sequence diagram in figure.6.4 depicts the steps involved in uploading QoS information of services into ESR.

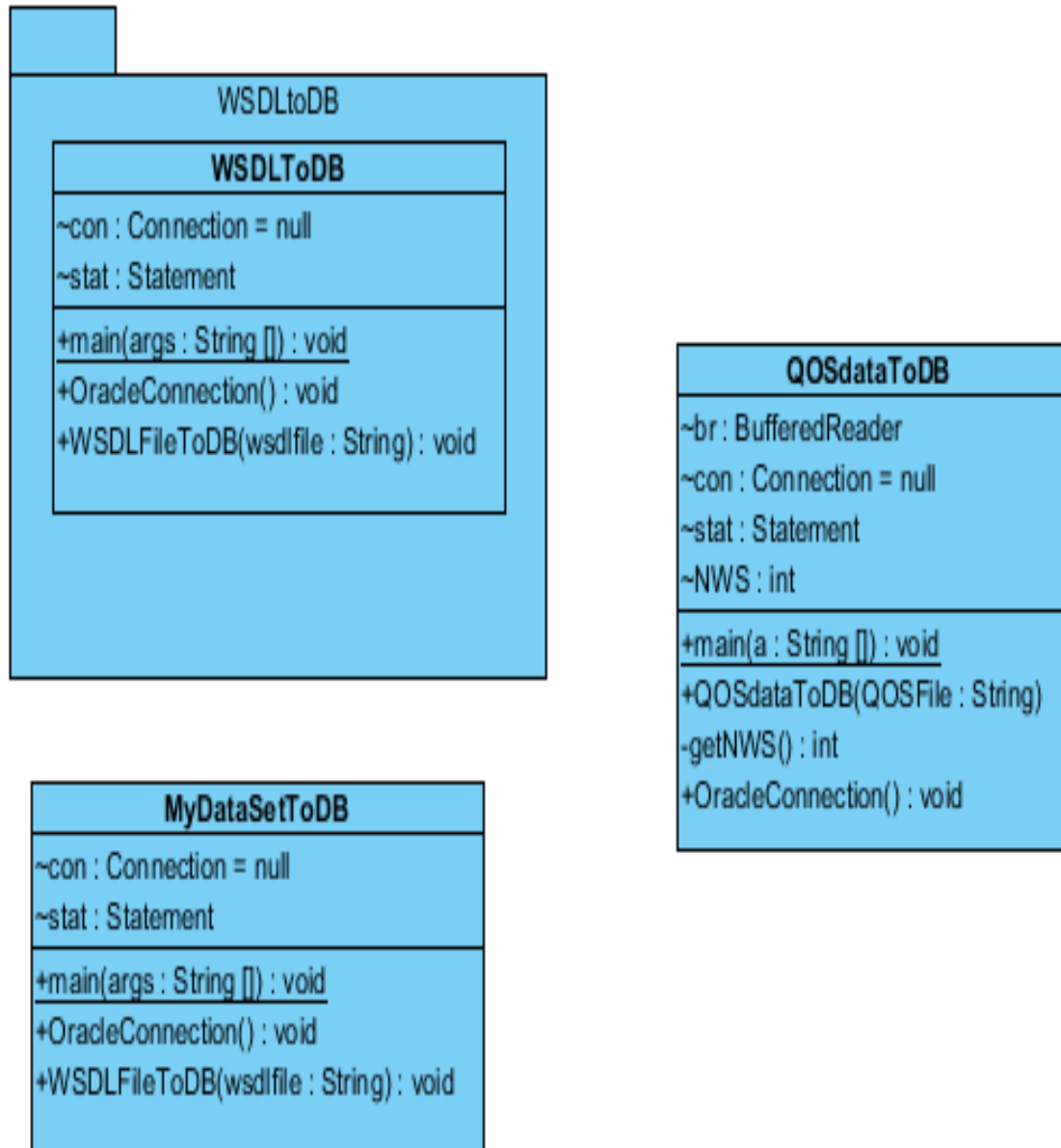


Figure 6.2: Class Diagram for Pre-processing Module



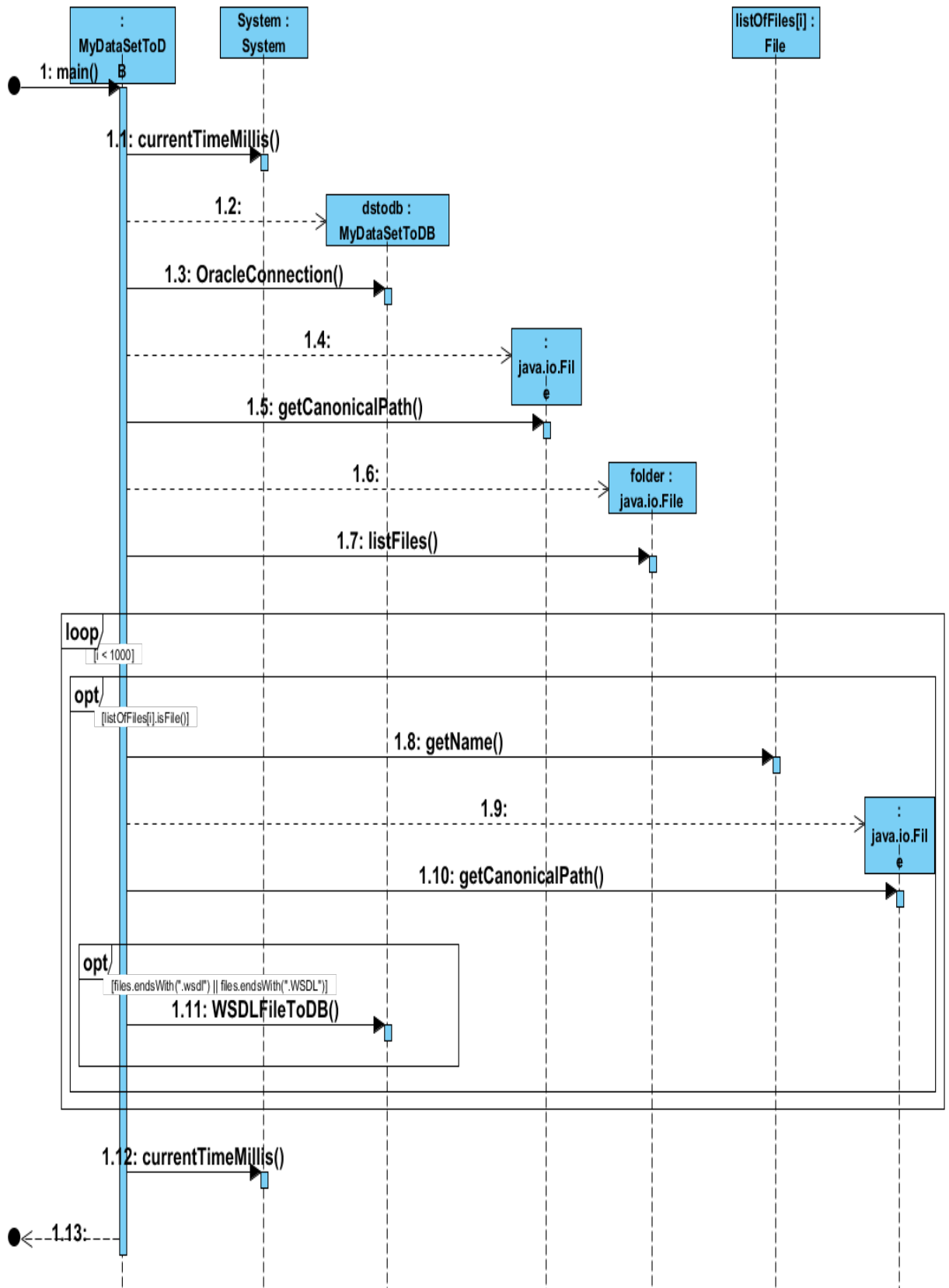


Figure 6.3: Sequence Diagram for MyDataSetToDB

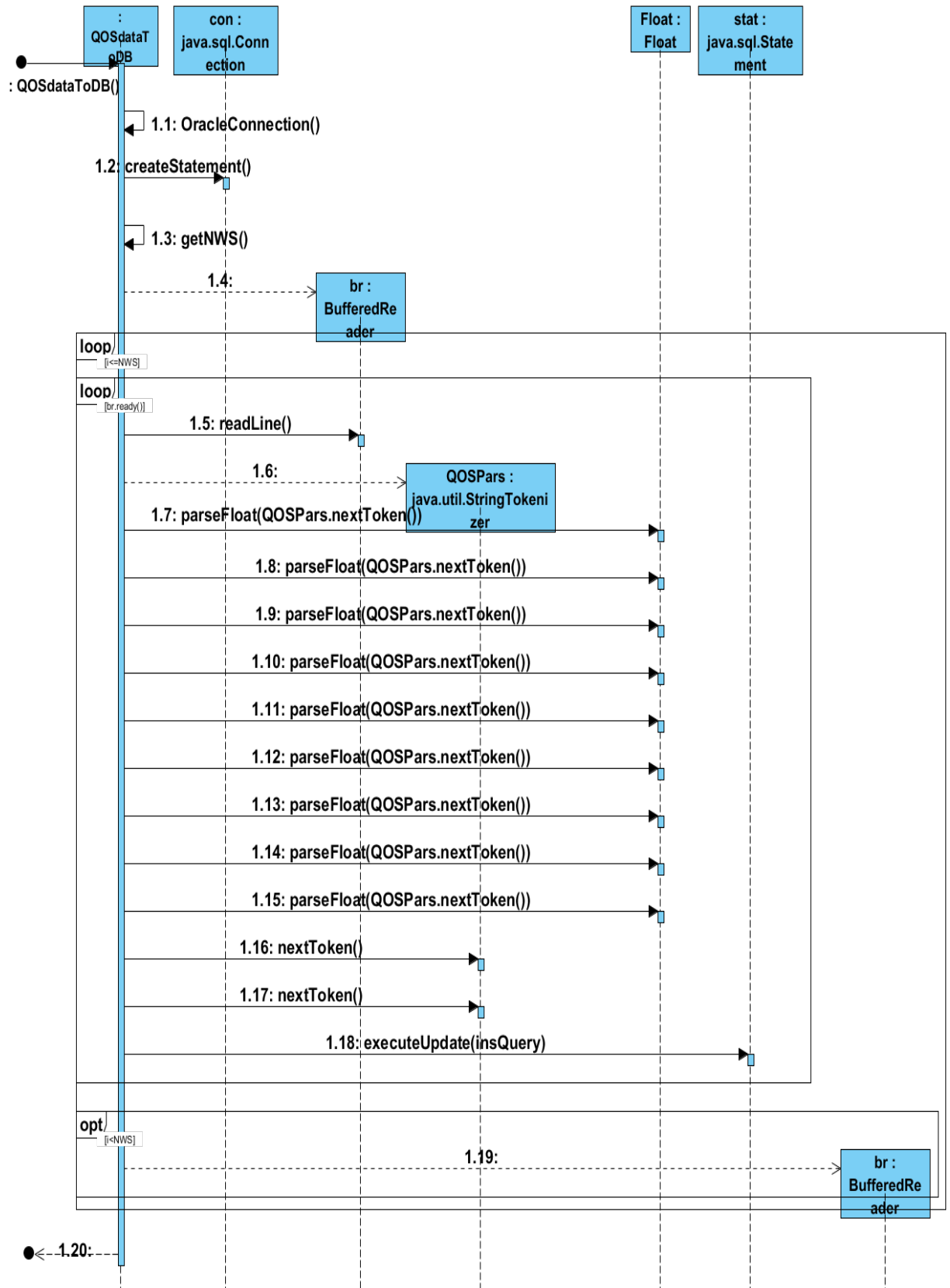


Figure 6.4: Sequence Diagram for QoSdataToDB

## 6.2.2 Service Search Module

The module first clusters service input and output parameters stored in an *ESR* based on their term equivalence as described previously in chapter 4. These clusters are further utilized to cluster web services in the registry based on their *frequent output parameter patterns* as explained in Chapter 4. This pre-processing is done to improve the performance of output parameter service search, since this search has a major role in parameter based service search and also in construction of *composition search tree* for finding service compositions. This module is implemented in three packages: *ParClustering*, *WSDLToClusters* and *SearchInCl*, each of which is explained in details in the following subsections.

### 6.2.2.1 ParClustering Package

Figure.6.5 depicts the class diagram of *ParClustering* package having 4 major classes: *ParClustersToDB*, *ClusterCohesion*, *ResizeMatrix* and *ResizeArray*. Class *ParClustersToDB* utilizes *ResizeMatrix* and *ResizeArray* and contains methods to calculate similarity values of parameter pairs using WordNet, storing these values in a matrix and to cluster parameters based on the computed similarity values. The *parameter clusters* so generated are further stored in *ESR*. Class *ClusterCohesion* is used to calculate the cluster cohesion values of the generated parameter clusters, to analyze the quality of clusters generated. The major steps involved in parameter clustering process is depicted as a sequence diagram in figure.6.6.

### 6.2.2.2 WSDLToClusters Package

Figure.6.7 depicts the class diagram of *WSDLToClusters* package having 4 major classes: *SCDToClFinalNoCO*, *AprioriAllFrSetsInOneFile*, *DBToDataFile* and *ClusterOverlapForAllSets*. Class *DBToDataFile* retrieves output parameters for all services in *ESR* and stores it in a file which is further utilized in *AprioriAllFrSetsInOneFile* to compute all

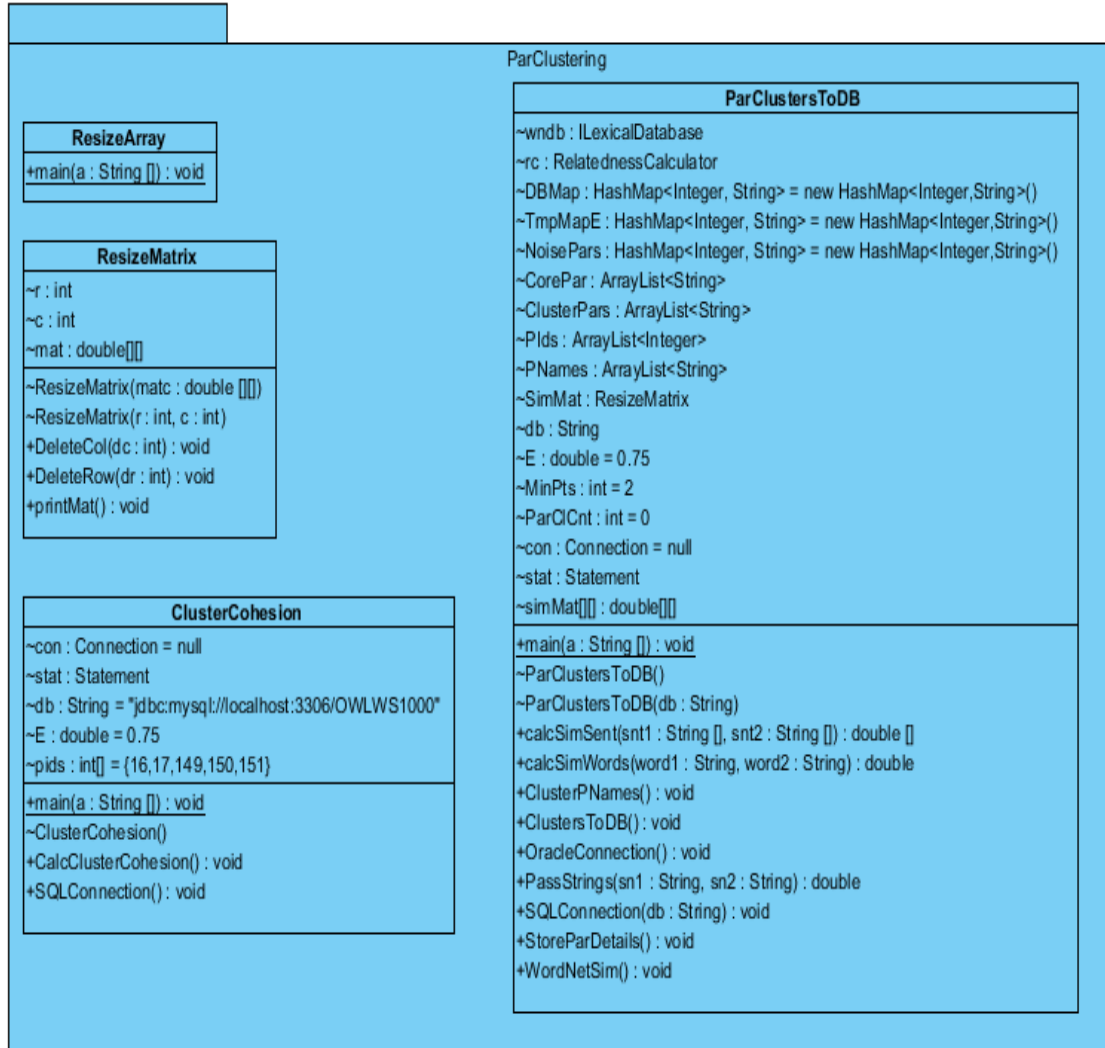


Figure 6.5: Class Diagram for parameter Clustering Package

frequent output parameter patterns and stores it in a file. This file is used in *SCDTo-ClFinalNoCO* which contain methods to cluster services on *frequent output parameter patterns*. This class utilizes *ClusterOverlapForAllSets* to compute cluster overlap values for all candidate clusters. The *covering clusters* so generated are further stored in *ESR*. The major steps involved in service clustering process is depicted as a sequence diagram in figure.6.8.

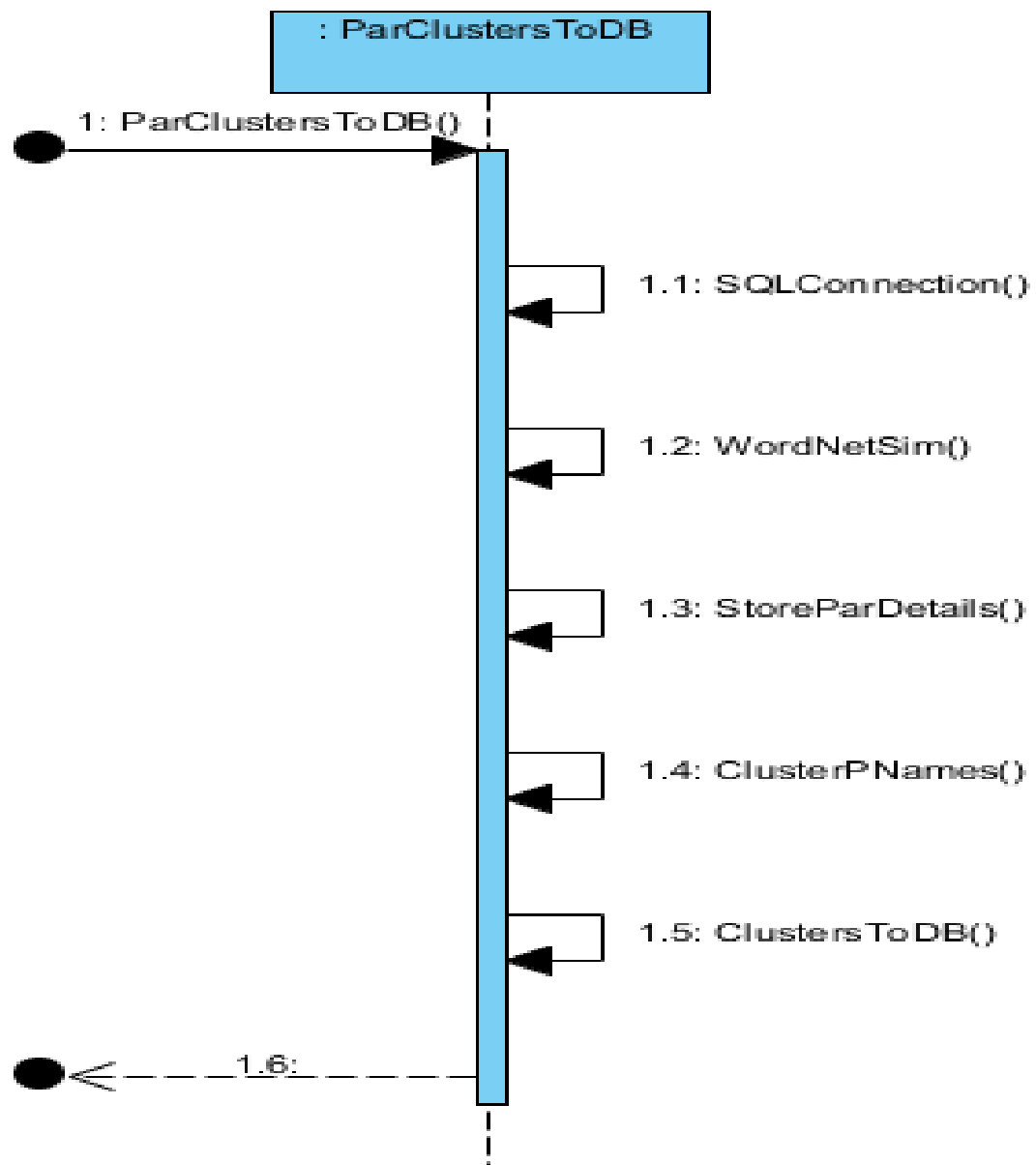


Figure 6.6: Sequence Diagram for ParClustersToDB

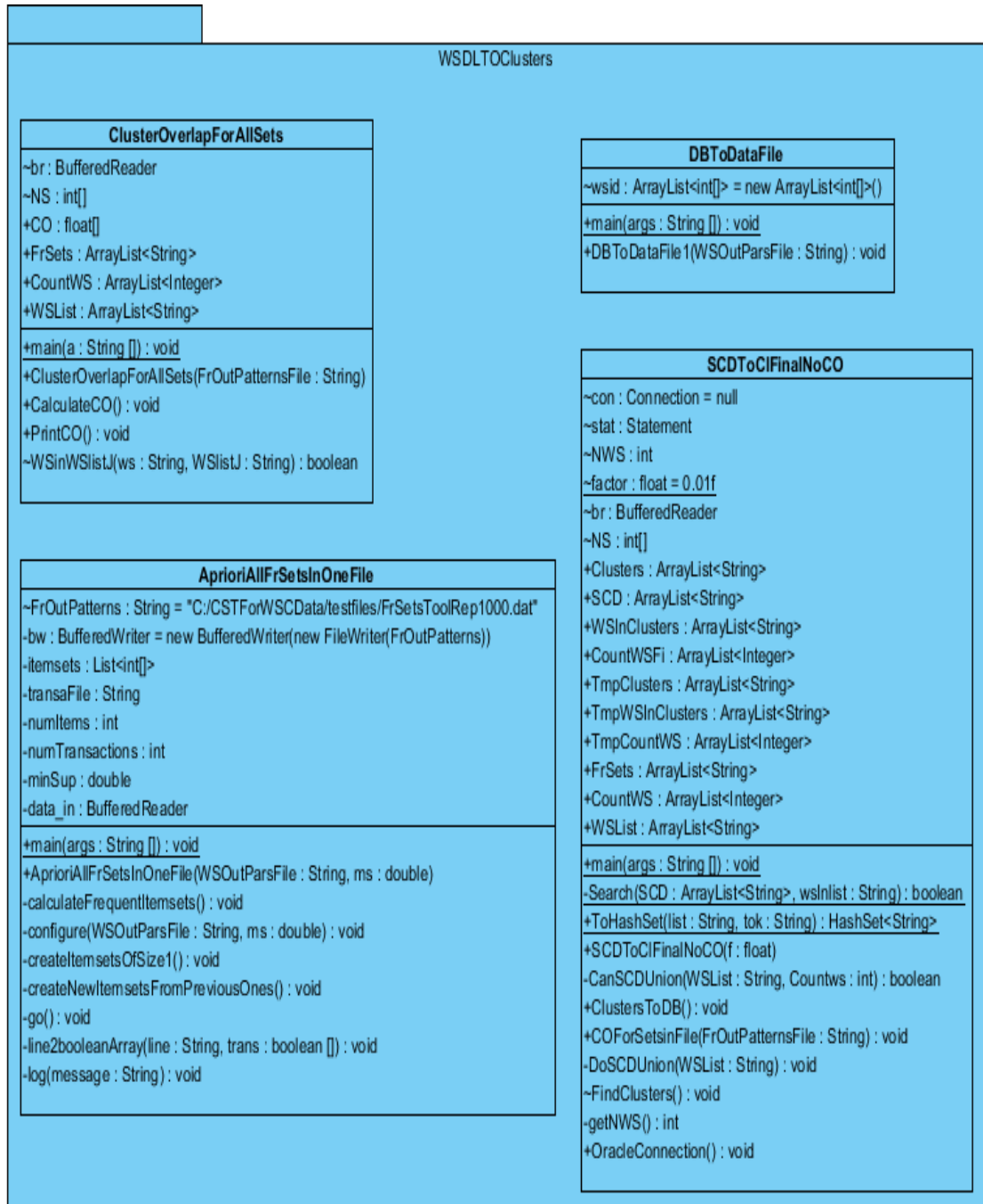


Figure 6.7: Class Diagram for Web Services Clustering Package

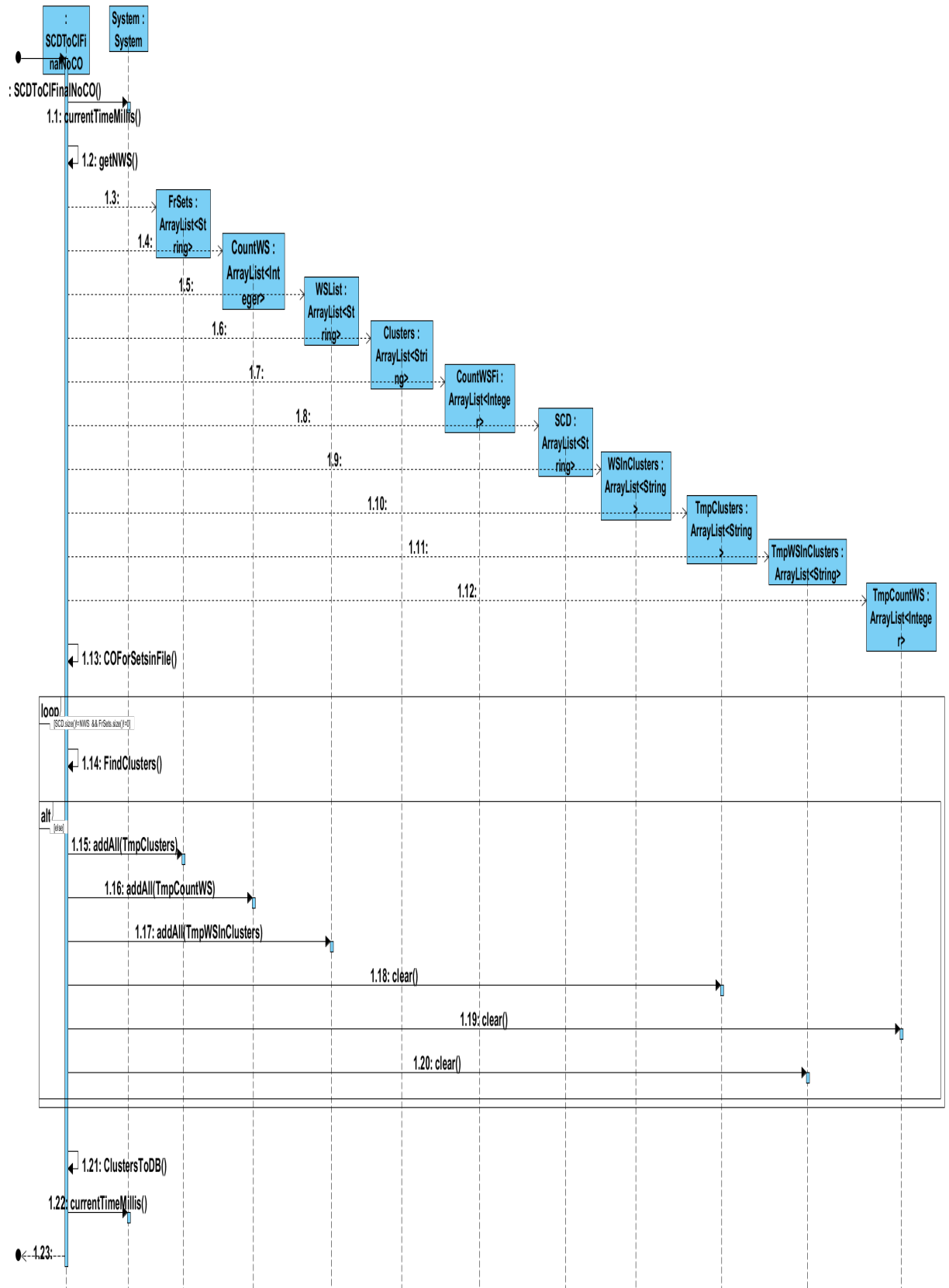


Figure 6.8: Sequence Diagram for SCDTToCIFinalNoCO

### 6.2.2.3 *SearchInCl* Package

Figure.6.9 depicts the class diagram of *SearchInCl* package containing 4 major classes: *SearchNameQoSInToolRep*, *SearchIOParmsInToolRep*, *SearchOutParsClToolRep* and *SearchIOQoSMatchInToolRep*. Class *SearchNameQoSInToolRep* implements keyword based service search and has a method for service selection based on QoS attributes. This class is used to compare our QoS based service selection approach with Chen's approach as explained earlier in chapter 5. Class *SearchOutParsClToolRep* has methods that searches for services matching user queried output parameters from *covering clusters* obtained from service clustering.

Class **SearchIOQoSMatchInToolRep** implements the service search wherein initially a set of web services matching the desired output parameters as provided in user query is searched from the web service clusters using class *SearchOutParsClToolRep*. Further, services that match the input parameters and the QoS attribute values provided in the user query are selected from this set using the service selection method proposed in chapter 5. The set of matching services thus obtained are classified according to the types of input and output parameter match as follows:

1. Exact Match : These web services have exact output match and full input match with the user query. These services can readily satisfy all the requirements in the query and hence a composition is not needed in such cases.
2. Exact Output Partial Input Match : These web services have a exact output match and partial input match with the user query. These services satisfy the output parameters required in the query but need some extra input parameters for executing them and hence composition is needed for satisfying the user query completely.
3. Super match : These web services have a super output match and a full input match with the user query. These services can readily satisfy all the requirements in the query and hence a composition is not needed in such cases.



4. Super output partial input match : These web services have a super output match and a partial input match with the user query. These services satisfy the output parameters required in the query but need some extra input parameters for executing them and hence composition is needed for satisfying the user query completely.
5. Partial output full input match : These web services have a partial output match and a full input match with the user query. These services partially satisfy the output parameters required in the query, but the input parameters provided by the user are sufficient for executing them. Hence a collaborative composition of such services is needed for satisfying the user query completely.
6. Partial Match : These web services have a partial output match and a partial input match with the user query. These services partially satisfy the output parameters required in the query and also need some extra input parameters for executing them. Hence a collaborative composition of such services is needed for satisfying the user query completely.

The major steps involved in service search process is depicted as a sequence diagram in figure.6.10. On obtaining web services having exact or super matches with the given user requirements, the user can infer that there are web services readily available in the *Extended service registry* satisfying the queried requirements and can use them directly. When no such web services are available service composition becomes inevitable for satisfying the user requirements.

### 6.2.3 Service Composition Module

When the user finds that there are no exact and super full matches, he may be interested finding compositions that satisfy the given requirements. This module implements the algorithm proposed for constructing *composition search tree* described previously in Chapter 5, for finding various service compositions satisfying the given user require-

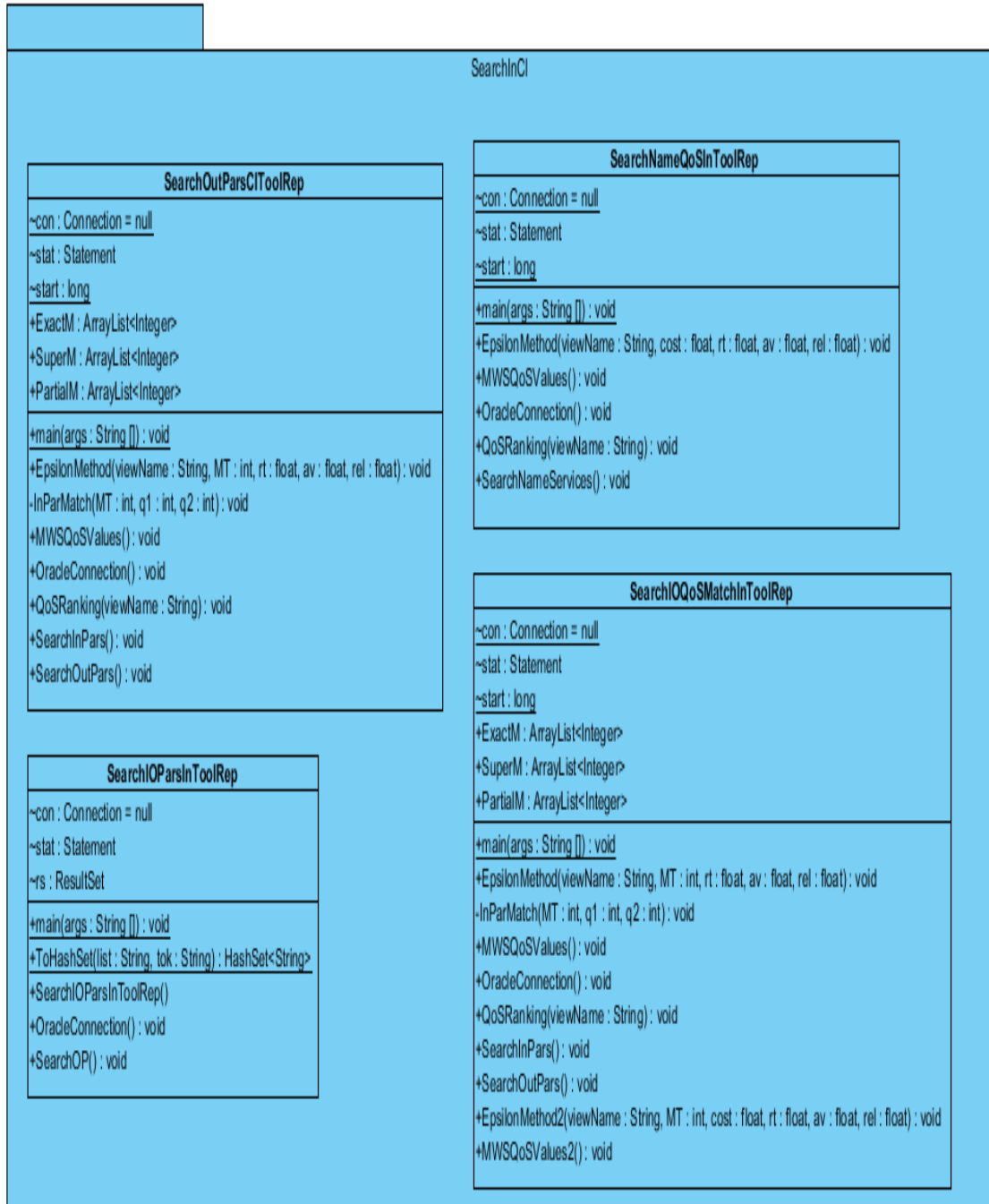


Figure 6.9: Class Diagram for Service Search Package

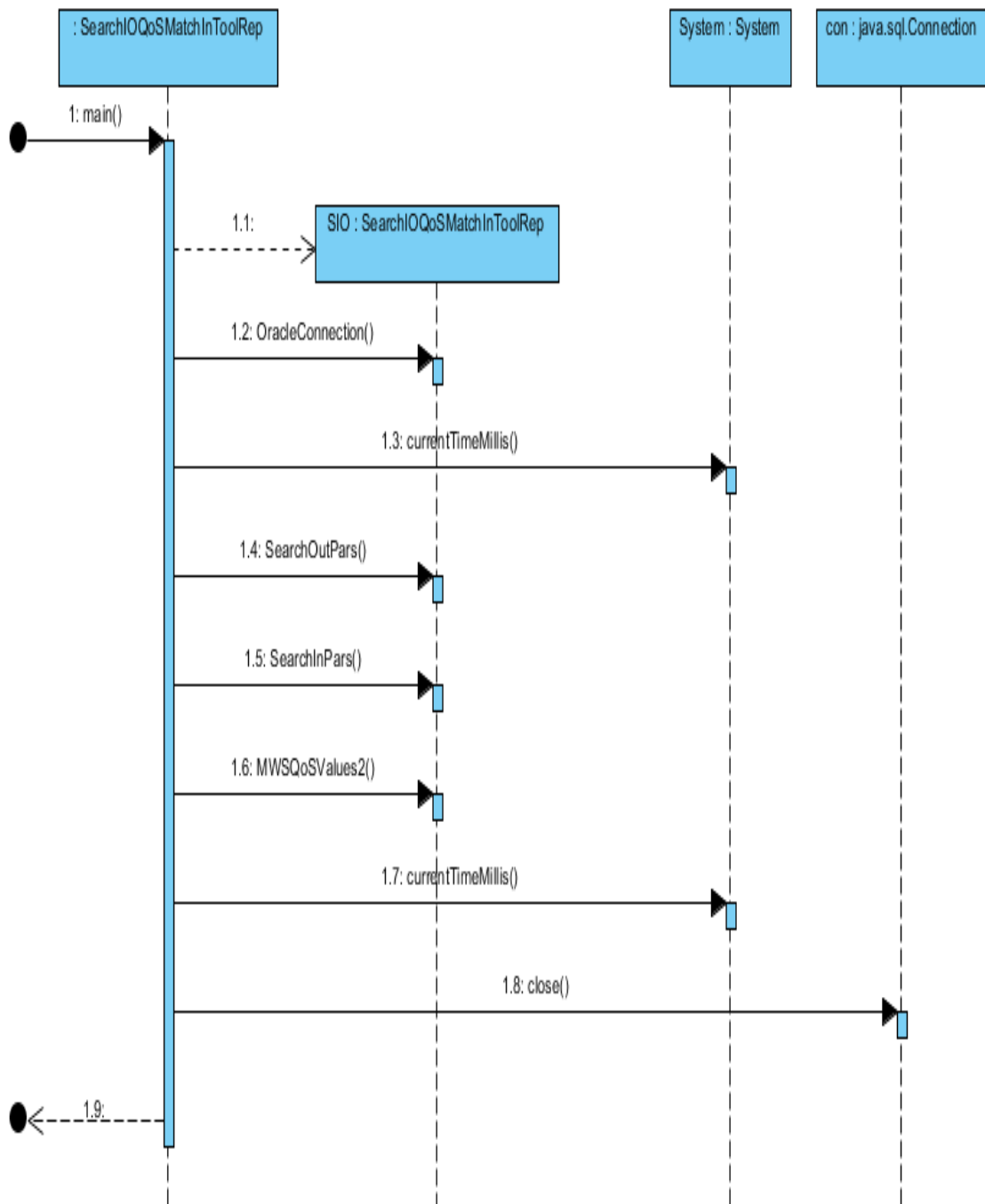


Figure 6.10: Sequence Diagram for SearchIOQoSMatch

ments. The tree construction utilizes the web service clusters obtained in service search module and also selection of services for each node in the *composition search tree* is done using the service selection approach implemented in service search module. Further solutions obtained in the *composition search tree* are listed as sets of web services participating in each composition.

Figure.6.11 shows the class diagram of this module containing 4 major classes: *CSTreeConstructionToolRep*, *TreeNode*, *SearchIOQoSForComposition* and *CSTreeConstruction*. Class *TreeNode* defines the data structure for a tree node and has constructors for initializing tree nodes. Class *CSTreeConstructionToolRep* defines methods for CST construction and utilizes class *SearchIOQoSForComposition* for service selection in each tree node, when there are more than one matching services available in the ESR. Class *SearchIOQoSForComposition* implements the proposed  $\epsilon$  method for service selection. The major steps in CST construction is depicted as sequence diagrams in figures 6.12 and 6.13.

#### 6.2.4 User Interface

The user interface for our tool is implemented in 4 JSP pages as described below:

1. **ToolMainPage.jsp**: The first and the main web page of the tool, used to input a user query where the user needs to provide details of the input parameters, desired output parameters and QoS attribute values expected from the web service being searched for.
2. **InParsList.jsp**: In this web page the various values provided by the user are displayed, and a confirmation is taken for searching services.
3. **MatchserviceList.jsp**: This web page displays a list of services matching the given user Query, along with their details. The matched services are classified



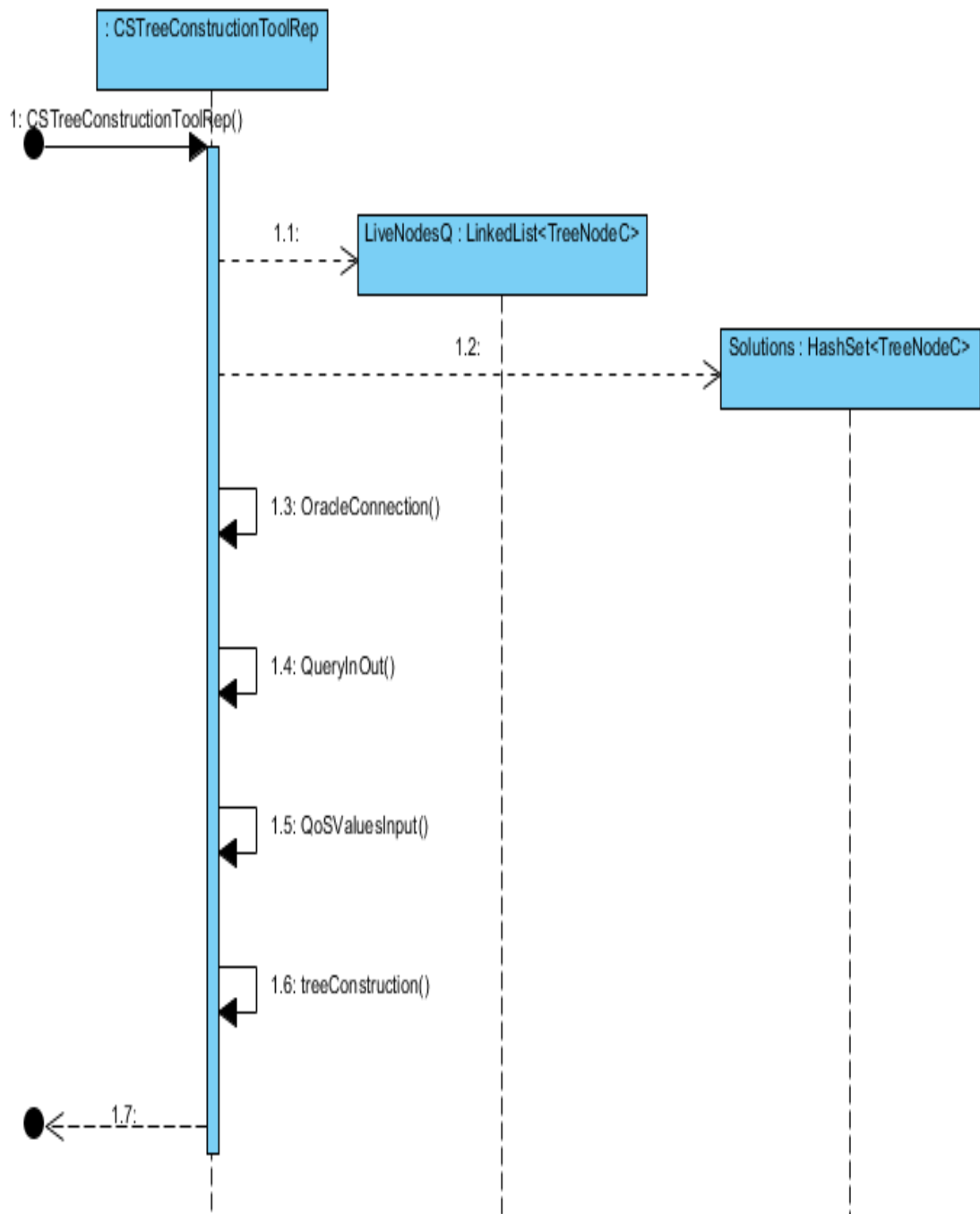


Figure 6.12: Sequence Diagram for CTreeConstruction

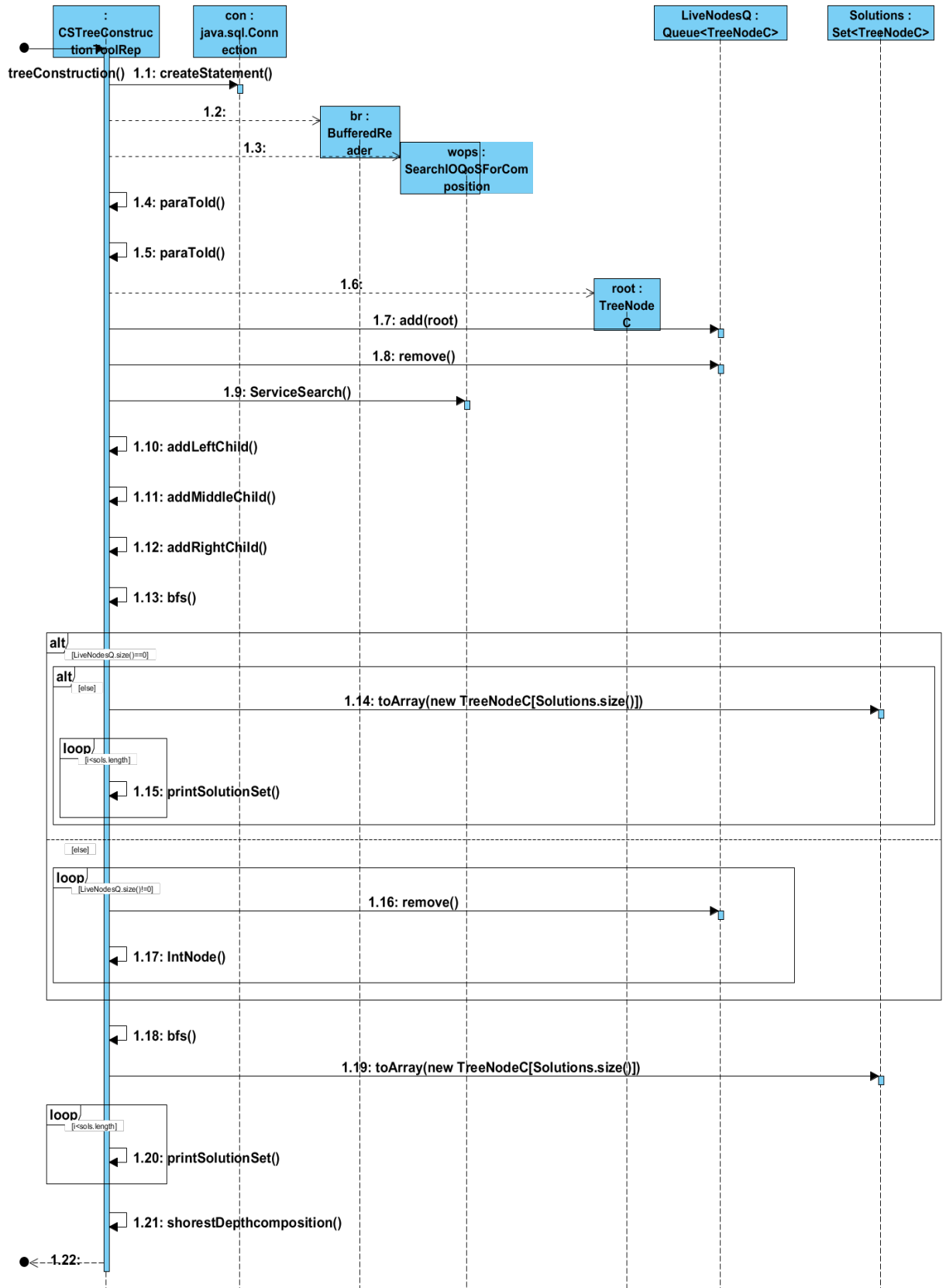


Figure 6.13: Sequence Diagram for TreeConstruction method

according to the types of input and output parameter match as follows:

- (a) Exact Match.
- (b) Exact Output Partial Input Match.
- (c) Super Match.
- (d) Super Output Partial Input Match.
- (e) Partial Output Full Input Match.
- (f) Partial Match.

4. **Composition.jsp**: This page displays the compositions that satisfy the user query.

The execution of our tool is demonstrated using 2 examples in the next section.

#### **6.2.4.1 Software Requirements for Tool Implementation**

We conducted experiments on QWS Data set Al-Masri and Mahmoud (2008), which includes WSDLs and QoS information of 2507 web services. We have run our experiments on a 1.3GHz Intel machine with 4 GB memory running Microsoft Windows 7. Our tool has been implemented using the following softwares and jar files:

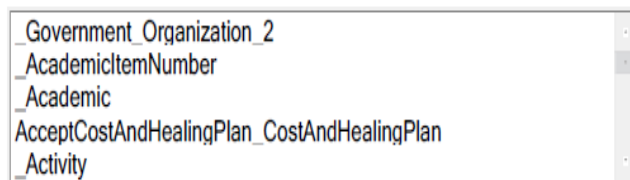
- Oracle 10g for storing ORDBMS schema.
- JDK 1.6 for implementing the proposed approaches.
- Java Server Pages (JSP) for implementing user interface.
- WSDL4J WSDL4J - Jar file for pre-processing WSDL files.
- WS4J hideki (2013) - WordNet Similarity for Java Package for finding parameter similarity.



## 6.3 Tool Usage

In this section we describe the working of our tool with 2 running examples. The various steps for executing the examples are clearly explained with actual screen-shots that are obtained from our tool. The tool has 4 JSP pages as discussed before, and screen-shots of these pages are taken for both the examples. The Main Page of our tool is shown in fig.6.14.

Select the input parameters you will provide from the drop down list



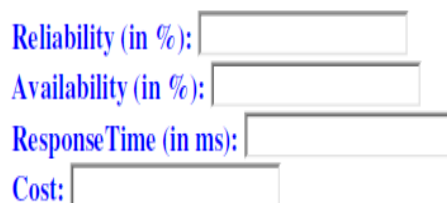
A screenshot of a web application's dropdown menu. The menu is open, showing a list of input parameters. The parameters are: "\_Government\_Organization\_2", "\_AcademicItemNumber", "\_Academic", "AcceptCostAndHealingPlan\_CostAndHealingPlan", and "\_Activity". The dropdown has a scroll bar on the right side.

Select the Output parameters you desire from the drop down list



A screenshot of a web application's dropdown menu. The menu is open, showing a list of output parameters. The parameters are: "\_UnilateralGiving", "\_Scholarship", "\_Duration", "\_Book", and "\_Author". The dropdown has a scroll bar on the right side.

Enter Desired QoS Values from Web Service



A screenshot of a web application's input fields for QoS values. There are four input fields, each with a label to its left: "Reliability (in %):", "Availability (in %):", "ResponseTime (in ms):", and "Cost:". The input fields are empty.

SUBMIT

Figure 6.14: Main Page

### 6.3.1 Screen-shots of Example 1

In our first example we provide the following values as input in the main web page:

1. input parameters : `_AcademicDegree`
2. output parameters : `_Lending`
3. QoS Attributes :
  - Reliability (in %) : 50
  - Availability (in %) : 50
  - Response Time (in ms) : 200
  - Cost : 100

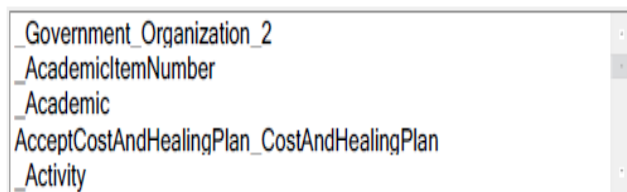
Fig.6.15 shows the **Main page** for user query. On clicking the **Submit** button, the **User IO and QoS Values** page displays all the requirements of the user as entered in the **Main page** , as shown in fig.6.16.

Once the user clicks on the **search services** button, a search for web services matching the user requirements is made in the Extended service registry. The matched services are classified according to their input and output parameter matches, as discussed previously. All the matching services are listed in the **Matched web services page** as shown in fig.6.17. In this example, there exists web services with Exact Full and Super Full matches for the user requirements and hence composition is not necessary.

The user can request for finding compositions satisfying the user query by clicking on the **Compose** button in this page. The compositions satisfying the query are found by constructing a *composition search tree* as explained previously in chapter 5. These compositions are displayed in the **Compositions Page** as shown in fig.6.18 where the list of web services required for each composition is tabulated separately. The compositions shown for this example are the web services that have Exact and Super matches

with the user query. Further compositions are not listed since web services satisfying the query requirements are readily available in the extended service registry.

Select the input parameters you will provide from the drop down list



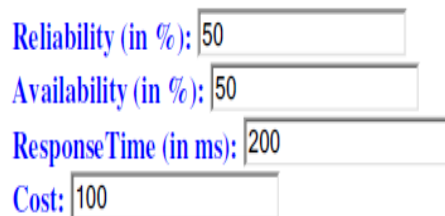
\_Government\_Organization\_2  
\_AcademicItemNumber  
\_Academic  
AcceptCostAndHealingPlan\_CostAndHealingPlan  
\_Activity

Select the Output parameters you desire from the drop down list



\_UnilateralGiving  
\_Scholarship  
\_Duration  
\_Book  
\_Author

Enter Desired QoS Values from Web Service



Reliability (in %): 50  
Availability (in %): 50  
ResponseTime (in ms): 200  
Cost: 100

SUBMIT

Figure 6.15: Main Page of Example 1

You have selected the following Input Parameters

- \_AcademicDegree

You have selected the following Output Parameters

- \_Lending

QoS Values entered

- Reliability (in %): 50
- Availability (in %): 50
- Response Time (in ms): 200
- Cost : 100

SEARCH SERVICES

Figure 6.16: Values Provided in User Query

#### MATCHED SERVICES

##### EXACT MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
9	Academic-degreeLendingService	1	4	78	102.62	91	67

##### EXACT OUTPUT PARTIAL INPUT MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
514	GovernmentLendingService	3	4	100	122.93	93	60
222	CitygovernmentLendingService	169	4	78	196	88	73
306	DegreeLendingService	203	4	89	126	83	83
303	DegreegovernmentLendingService	3,203	4	89	138.5	86	73

##### SUPER MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
12	Academic-degreeLendingdurationService	1	4,8	89	221.48	90	53

##### SUPER OUTPUT PARTIAL INPUT MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
101	AwardLendingdurationService	77	4,8	89	146.08	57	53
640	MissilegovernmentLendingrangeService	3,344	4,80	89	81	93	58

##### PARTIAL OUTPUT FULL INPUT MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
------------	--------------	------------	-------------	------	---------------	--------------	-------------

##### PARTIAL MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
------------	--------------	------------	-------------	------	---------------	--------------	-------------

COMPOSE

Figure 6.17: Web Services Matching User Query

### User Query

- Reliability : 50.0
- Availability : 50.0
- Response Time :200.0
- Cost :100.0
- Input Parameters Provided : \_AcademicDegree
- Desired Outpar Parameters : \_Lending

### COMPOSITIONS

#### Composition 1

Service Name	Service Id
Academic-degreeLendingService	9

#### Composition 2

Service Name	Service Id
Academic-degreeLendingdurationService	12

Figure 6.18: Compositions Satisfying User Query

### 6.3.2 Screen-shots of Example 2

As a second example we provide the following values as input in the main web page:

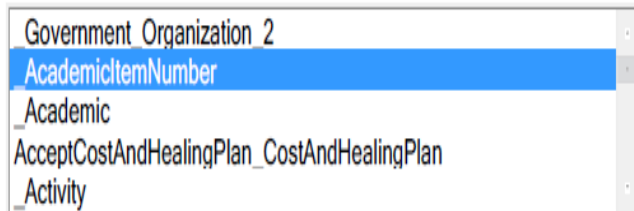
1. input parameters : `_AcademicItemNumber`
2. output parameters : `_Book` and `_MaxPrice`
3. QoS Attributes :
  - Reliability (in %) : 50
  - Availability (in %) : 50
  - Response Time (in ms) : 200
  - Cost : 100

Fig.6.19 shows the **Main page** for user query. On clicking the **Submit** button, the **User IO and QoS Values** page displays all the requirements of the user as entered in the **Main page** , as shown in fig.6.20.

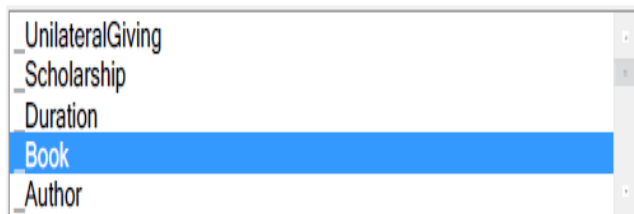
Once the user clicks on the **search services** button, a search for web services matching the user requirements is made in the Extended service registry. The matched services are classified according to their input and output parameter matches, as discussed previously. All the matching services are listed in the **Matched web services page** as shown in fig.6.22. In this example, there does not exist any web services with Exact and Super matches for the user requirements and hence composition is necessary.

The user can request for finding compositions satisfying the user query by clicking on the **Compose** button in this page. The compositions satisfying the query are found by constructing a *composition search Tree* as explained previously in chapter 5. These compositions are displayed in the **compositions page** as shown in fig.6.23 where the list of web services required for each composition is tabulated separately. It is seen that there is only one composition that satisfies the given user requirement.

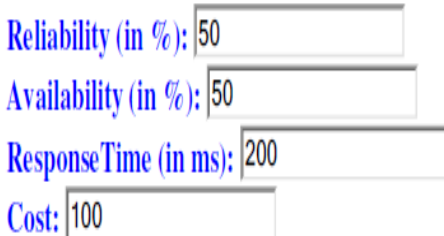
Select the input parameters you will provide from the drop down list



Select the Output parameters you desire from the drop down list



Enter Desired QoS Values from Web Service



SUBMIT

Figure 6.19: Main Page of Example 2



You have selected the following Input Parameters

- \_AcademicItemNumber

You have selected the following Output Parameters

- \_Book
- \_MaxPrice

QoS Values entered

- Reliability (in %): 50
- Availability (in %): 50
- Response Time (in ms): 200
- Cost : 100

SEARCH SERVICES

Figure 6.20: Values Provided in User Query

#### MATCHED SERVICES

##### EXACT MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
------------	--------------	------------	-------------	------	---------------	--------------	-------------

##### EXACT OUTPUT PARTIAL INPUT MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
52	AuthorBookmaxpriceService	11	10,57	100	227	88	73

##### SUPER MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
------------	--------------	------------	-------------	------	---------------	--------------	-------------

##### SUPER OUTPUT PARTIAL INPUT MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
------------	--------------	------------	-------------	------	---------------	--------------	-------------

##### PARTIAL OUTPUT FULL INPUT MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
134	BookService		10	100	180	95	73
16	Academic-item-numberBookService	9	10	78	67.5	86	73
15	Academic-item-numberBookauthorService	9	10,11	89	134.07	84	60

##### PARTIAL MATCHES

Service Id	Service Name	Input Pars	Output Pars	Cost	Response Time	Availability	Reliability
950	TitleActionfilmmaxpricequalityService	521	180,57,139	100	62.05	72	67
955	TitleComedyfilmmaxpricequalityService	521	179,57,139	100	94.67	98	73
870	ShoppingmallMaxpricedigital-videoService	508	57,512	78	103	85	73
64	AuthorPublicationmaxpriceService	11	12,57	89	115	72	83
54	AuthorBooktaxedpriceService	11	10,58	100	173	67	73
369	FoodMaxpricequantityService	248	57,87	78	152	96	73
691	NovelAuthormaxpriceService	66	11,57	78	169.4	90	73
849	SciencefictionbookAuthormaxpriceService	67	11,57	78	140	89	73
831	RomanticnovelAuthormaxpriceService	488	11,57	78	136.67	63	78
568	IsbnBookauthorService	381	10,11	78	166.6	51	73
881	Short-storyAuthormaxpriceService	516	11,57	89	114	83	73

Figure 6.21: Web Services Matching User Query

602	MeatMaxpricequantityService	400	57,87	78	86.86	100	73
368	FoodMaxpricequantityService	248	57,87	89	50.75	56	80
658	Mp3playerMaxpriceService	142	57	89	194	56	78
83	AutobicycleMaxpriceService	71,69	57	100	100	94	73
79	Auto2personbicycleMaxpriceService	70,69	57	89	123	96	83
782	PublicationBookService	12	10	89	173	89	73
773	Publication-numberBookService	468	10	89	168.5	88	60
772	Publication-numberBookService	468	10	89	197	89	60
735	PreparedfoodMaxpriceService	353	57	78	174.17	88	73
155	ButterMaxpriceService	119	57	89	104.5	85	73
932	TeaMaxpriceService	206	57	78	98.67	91	67
193	CdplayerMaxpriceService	141	57	100	149.67	86	73
199	Cheapcar2personbicycleMaxpriceService	70,52	57	78	116	89	67

COMPOSE

Figure 6.22: Web Services Matching User Query

## User Query

- Reliability : 50.0
- Availability : 50.0
- Response Time :200.0
- Cost :100.0
- Input Parameters Provided : \_AcademicItemNumber
- Desired Outpar Parameters : \_Book,\_MaxPrice

## COMPOSITIONS

### Composition 1

Service Name	Service Id
Academic-item-numberBookauthorService	15
AuthorBookmaxpriceService	52

Figure 6.23: Compositions Satisfying User Query

## 6.4 Conclusion

The motivation for proposing extended service registry was to enable parameter based web service search and composition in UDDI based service Registries. We have proposed approaches for extending the service registry, searching web services on their output parameters, approaches to improve output parameter based searches, service selection and service composition in the previous Chapters. This chapter gives a brief description of the Tool we have implemented to demonstrate the usefulness of our proposed extended service registry. The various approaches proposed in chapters 3, 4 and 5 are implemented in 3 modules : pre-processing module, service search module and service composition module as explained in section6.2. A brief description of the system architecture and the user interface is also given in section6.2.

Further, we have tested the correctness of our tool for various user queries in the user interface and checking the corresponding service matches and compositions listed in the interface. Two such examples along with the screen-shots obtained during execution are presented in section6.3. The tool helps a user in finding services that satisfies a user requirement by either finding a service or composing services appropriate to the requirement, from the registered services in the extended service registry.

## CHAPTER 7

### Conclusion and Future Work

Research on web service search and composition has become increasingly important in recent years due to the growing number of web services over Internet and the challenge of automating the process of composition. Since there is a large growth in number of available web services and possible interactions among them are huge, searching for desired set of services to satisfy a user query becomes very difficult. This in turn means service search and composition problem has become increasingly sophisticated and complicated for finding a solution.

Various approaches can be used for service search, such as, searching in UDDI, Web and service portals. Searching in service registries based on UDDI are still popular for its practicality in business world as searching there is less time consuming than searching on world wide web. The limitation of UDDI is that its search API is limited by the kind of information that is available and searchable in UDDI entries and hence do not provide any support for complex searches like input/output parameter based search and automatic composition of web services. The limitations of UDDI motivated us to explore the feasibility of input/output parameter based web service search and composition, as an extension to UDDI, to support varying requirements of a user. We propose to build an *extended service registry(ESR)* by extending existing UDDI framework with meta information of services a UDDI contain. ESR is capable of offering parameter based web service search and composition operations. We summarize the research problems encountered and the corresponding solutions proposed below:

- A web service,  $ws$ , has typically two sets of parameters - set of inputs  $ws^I$  and set of outputs  $ws^O$ . Conventionally two services  $ws_i$  and  $ws_j$  are said to be

composable iff  $ws_i^O = ws_j^I$ , i.e,  $ws_j$  receives all the required inputs from outputs  $ws_i$  has. A service composition is formed by constructing a chain of such composable services. However, the making of a service composition chain may fail at a point when the output parameters of a preceding service ( $ws_P^O$ ) does not match exactly with the input parameters of a succeeding service( $ws_S^I$ ). We propose an approach to alleviate this problem by making match criteria flexible. In addition to *exact* match we allow *partial* as well as *super* match for conditions  $ws_i^O \subset ws_j^I$  and  $ws_i^O \supset ws_j^I$  respectively. Web service details like service name, provider, service url, input and output parameters,etc are stored in an *extended service registry*(ESR) that supports input/output parameter based service search.

- A web service parameter name,(eg: FlightInfo, CheckHotelCost), is typically a sequence of concatenated words,(eg: Flight, Check, Hotel, Cost), referred to as terms. For effectively matching input/output parameters of web services, it is essential to consider their underlying semantics. However, this is hard since parameter names are not standardized and are usually built using terms that are highly varied due to the use of synonyms, hypernyms, and different naming conventions. Hence, to widen the scope of search and to further improve the performance of parameter based search in our ESR we propose a two-level pre-processing of service registry.

The scope of input/output parameter matching is widened in the first level, by clustering web service input/output parameters based on their term equivalence. We make use of semantic similarity and co-occurrence of terms in parameter names, computed using wordnet as the underlying ontology, to cluster service parameters into semantic groups. The parameter clustering serves as a pre-processing step for the next level where web services are clustered on their output parameter patterns, to accelerate parameter based service search in ESR. Our clustering approach makes use of the co-occurrence of output parameters to cluster web services. Set of output parameters that co-occur in more than a threshold number of

web services make a *frequent output parameter pattern*. A set of services having the same frequent output parameter pattern make a *candidate cluster*. A *covering cluster* is a chosen set(subset) of candidate clusters so that they contain all the web services of the registry and also have a minimum overlap.

- When a user queries ESR for a service with a required output parameter pattern, initially, a cluster from *covering cluster*, that best matches the queried pattern is searched for. This matched cluster contains many services whose parameter pattern matches with queried pattern, from which best matching services need to be selected. Hence, as a next step, a bi-level service selection approach is proposed, that considers both the functional and the non-functional requirements of users during service selection. The functional requirements are provided by a user as a set of input parameters provided for and output parameters desired from the web service. The user also provides a set of desired QoS values and the order of their preference for selection. In first level services matching the functional requirements are shortlisted, which are further filtered in second level based on given QoS requirements, thus providing a list of web services that best matches a given user query.
- When service selection fails to find a matching service for a given user requirement, service composition becomes inevitable. In order to meet such requirements we propose to find possible compositions that satisfy a given user query. Utilizing the three types of matches: exact, super and partial, we widen the scope of composition by defining possibly three types of service composition: *exact*, *super* composition and *collaborative* compositions. The process of service composition is visualized as generation of a *composition search tree* that arranges services in levels showing the way service compositions can be made to meet a user requirements, i.e, such a tree can be viewed as a result to processing of a consumer query. The paths of a tree depicts chaining of services due to matching of input output parameters. Further, the utility of *composition search tree* for finding compositions like *leanest composition* and *shortest depth composi-*



tion is proposed, that helps users pick up the best compositions among the many compositions found.

Summarizing, we have proposed an *extended service registry* that supports parameter based search and composition. This is useful when a user is looking for a web service for a given input and desired output parameters and also has specific QoS requirements to be met. Based on the developed concept, we have designed a tool to build extended service registry and automate both service search and composition.

### 7.0.1 Future Work

Our proposed *extended service registry* finds many compositions that satisfy a user query when there are no ready services in the registry satisfying the given requirements. These compositions are returned to user as set of services participating in the compositions, irrespective of the types of compositions (*exact*, *super* and *collaborative*) involved. We can further generate abstract business process (say in BPEL), for each composition satisfying given user query, from the set of services participating in a composition. User can choose an abstract business process from which an executable business process could be generated to obtain the required results for his query.

Due to the dynamic nature of web services, a participating service in a composition may be unavailable or fail when executing the corresponding business service. This scenario requires a fault tolerant framework to enable a failure free service composition execution. A CST for a user query provides several workflows showing the participating services and their sequences of executions in order to meet a requirement a user asked for in his query. Our CST can be easily adopted to achieve fault tolerance, since at each node of CST, we obtain many matching services. On failure of a service at any stage, a list of services that are equivalent to the failed service can be obtained from these set of matching services. A replacement policy that selects a service from this list to replace the failed service can be proposed, thereby making service execution resilient.

Our service clustering approach clusters services registered in ESR. The service environment is highly dynamic in that new services may be added, existing services may be modified or certain services may be called off, regularly. To enable ESR to adapt to such registry evolution, techniques to re-organize service clusters in ESR, with minimal effort, need to be proposed.

## REFERENCES

1. **Agrawal, R., R. Srikant, et al.** (1994). Fast algorithms for mining association rules. **1215**, 487–499.
2. **Al-Masri, E. and Q. H. Mahmoud** (2008). Investigating web services on the world wide web, 795–804. URL <http://doi.acm.org/10.1145/1367497.1367605>.
3. **Alrifai, M., T. Risse, and W. Nejdl** (2012). A hybrid approach for efficient web service composition with end-to-end qos constraints. *ACM Trans. Web*, **6**, 7:1–7:31.
4. **Arpinar, I. B., B. Aleman-Meza, R. Zhang, and A. Maduko** (2004). Ontology-driven web services composition platform, 146–152.
5. **Blake, M., W. Cheung, M. Jaeger, and A. Wombacher** (2006). Wsc-06: The web service challenge, 62–62.
6. **Braga, D., S. Ceri, F. Daniel, and D. Martinenghi** (2008). Optimization of multi-domain queries on the web. *Proc. VLDB Endow.*, **1**, 562–573.
7. **Cai, D. and S. Xu** (2014). Lexical multicriteria-based quality evaluation model for web service composition. **278**, 253–258.
8. **Carolgeyer** (2013). Uddi specifications. URL [http://uddi.org/pubs/uddi\\_v3.html](http://uddi.org/pubs/uddi_v3.html).
9. **Colgrave, J., K. Januszewski, L. Clément, and T. Rogers** (2004). Using wsdl in a uddi registry, version 2.0. 2. *Technical note, OASIS*.
10. **Curbera, F., M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana** (2002). Unraveling the web services web: an introduction to soap, wsdl, and uddi. *Internet Computing, IEEE*, **6**(2), 86–93.
11. **Dong, X., A. Halevy, J. Madhavan, E. Nemes, and J. Zhang** (2004). Similarity search for web services, 372–383.
12. **Dustdar, S. and W. Schreiner** (2005). A survey on web services composition. *Int. J. Web Grid Serv.*, 1–30.
13. **El Hadad, J., M. Manouvrier, and M. Rukoz** (2010). Tqos: Transactional and qos-aware selection algorithm for automatic web service composition. *Services Computing, IEEE Transactions on*, **3**, 73–85.

14. **Elgazzar, K., A. Hassan, and P. Martin** (2010). Clustering wsdl documents to bootstrap the discovery of web services.
15. **Ester, M., H.-P. Kriegel, J. Sander, and X. Xu** (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. **96**, 226–231.
16. **Gekas, J. and M. Fasli** (2005). Automatic web service composition based on graph network analysis metrics. **3761**, 1571–1587.
17. **Goodwin, J. C., D. J. Russomanno, and J. Qualls** (2007). Survey of semantic extensions to uddi: Implications for sensor services., 16–22.
18. **Hand, D. J., H. Mannila, and P. Smyth**, *Principles of data mining*. MIT press, 2001.
19. **Hashemian, S. and F. Mavaddat** (2006). A graph-based framework for composition of stateless web services, 75–86.
20. **hideki, s.** (2013). Ws4j. URL <https://code.google.com/p/ws4j/>.
21. **Huang, A. F., C.-W. Lan, and S. J. Yang** (2009). An optimal qos-based web service selection scheme. *Information Sciences*, **179**, 3309 – 3322.
22. **Jiang, J. J. and D. W. Conrath** (1997). Semantic similarity based on corpus statistics and lexical taxonomy.
23. **Juric, M. B., A. Sasa, B. Brumen, and I. Rozman** (2009). Wsdl and uddi extensions for version support in web services. *Journal of Systems and Software*, **82**, 1326–1343.
24. **Karim, R., C. Ding, and C.-H. Chi** (2011). An enhanced promethee model for qos-based web service selection, 536–543.
25. **Kuang, L., Y. Li, S. Deng, and Z. Wu** (2007). Inverted indexing for composition-oriented service discovery, 257–264.
26. **Kurt T Stam, A. O.** (2014). Apache juddi. URL <http://juddi.apache.org/docs/3.2/juddi-guide/html/>.
27. **Kwon, J., K. Park, D. Lee, and S. Lee** (2007). Psr : Pre-computing solutions in rdbms for fastweb services composition search, 808–815.
28. **Lakshmi, H. and H. Mohanty** (2012). Rdbms for service repository and composition, 1–8.
29. **Lakshmi, H. and H. Mohanty** (2015). Extended service registry to support i/o parameter-based service search, 145–155.
30. **Lee, D., J. Kwon, S. Lee, S. Park, and B. Hong** (2011). Scalable and efficient web services composition based on a relational database. *J. Syst. Softw.*, **84**(12). ISSN 0164-1212.

31. **Li, J., Y. Yan, and D. Lemire** (2014a). Full solution indexing using database for qos-aware web service composition, 99–106.
32. **Li, J., X.-L. Zheng, S.-T. Chen, W.-W. Song, and D. ren Chen** (2014b). An efficient and reliable approach for quality-of-service-aware service composition. *Information Sciences*, **269**(0), 238 – 254. ISSN 0020-0255. URL <http://www.sciencedirect.com/science/article/pii/S0020025513008657>.
33. **Lin, C.-F., R.-K. Sheu, Y.-S. Chang, and S.-M. Yuan** (2011). A relaxable service selection algorithm for qos-based web service composition. *Information and Software Technology*, **53**, 1370 – 1381.
34. **Lin, D.** (1998). An information-theoretic definition of similarity. **98**, 296–304.
35. **Liu, F., Y. Shi, J. Yu, T. Wang, and J. Wu** (2010). Measuring similarity of web services based on wsdl, 155–162.
36. **Liu, W. and W. Wong** (2009). Web service clustering using text mining techniques. *Int. J. Agent-Oriented Softw. Eng.*, **3**(1), 6–26. ISSN 1746-1375. URL <http://dx.doi.org/10.1504/IJAOSE.2009.022944>.
37. **Ma, J., Y. Zhang, and J. He** (2008). Efficiently finding web services using a clustering semantic approach, 1–8.
38. **MacQueen, James, et al.** (1967). Some methods for classification and analysis of multivariate observations, 281–297.
39. **Makris, C., Y. Panagis, E. Sakkopoulos, and A. Tsakalidis** (2006). Efficient and adaptive discovery techniques of web services handling large data sets. *J. Syst. Softw.*, **79**, 480–495.
40. **Marler, R. and J. Arora** (2004a). Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, **26**(6), 369–395. ISSN 1615-147X. URL <http://dx.doi.org/10.1007/s00158-003-0368-6>.
41. **Marler, R. T. and J. S. Arora** (2004b). Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, **26**(6), 369–395.
42. **Medjahed, B., A. Bouguettaya, and A. K. Elmagarmid** (). Composing web services on the semantic web. *The VLDB Journal*, 333–351.
43. **Mili, H., R. B. Tamrout, and A. Obaid** (2005). Jregistry: an extensible uddi registry. *Reports of NOTERE*, 115–128.
44. **Miller, G. A.** (1995). Wordnet: a lexical database for english. *Communications of the ACM*, **38**(11), 39–41.
45. **Mobedpour, D. and C. Ding** (2013). User-centered design of a qos-based web service selection system. *Service Oriented Computing and Applications*, **7**(2), 117–127. ISSN 1863-2386. URL <http://dx.doi.org/10.1007/s11761-011-0091-x>.

46. **Muschamp, P. ()**. An introduction to web services. *BT Technology Journal*, **22**(1), 9–18.
47. **Nayak, R. and B. Lee** (2007). Web service discovery with additional semantics and clustering, 555–558.
48. **Pedersen, T., S. Patwardhan, and J. Michelizzi** (2004). Wordnet:: Similarity: measuring the relatedness of concepts, 38–41.
49. **QoS** (2003). Qos for web services: Requirements and possible approaches. URL <http://www.w3c.or.kr/kr-office/TR/2003/ws-qos/>.
50. **Ran, S.** (2003). A model for web services discovery with qos. *SIGecom Exch.*, **4**, 1–10.
51. **Rao, J. and X. Su** (2005). A survey of automated web service composition methods, 43–54.
52. **Resnik, P.** (1995). Using information content to evaluate semantic similarity in a taxonomy, 448–453.
53. **Rodriguez-Mier, P., M. Mucientes, and M. Lama** (2011). Automatic web service composition with a heuristic-based search algorithm, 81–88.
54. **SOA** (2013). Soa. URL <http://www.opengroup.org/soa/source-book/soa/soa.htm>.
55. **Srivastava, U., K. Munagala, J. Widom, and R. Motwani** (2006). Query optimization over web services, 355–366.
56. **Syu, Y., S.-P. Ma, J.-Y. Kuo, and Y.-Y. FanJiang** (2012). A survey on automated service composition methods and related techniques, 290–297.
57. **Talantikite, H. N., D. Aissani, and N. Boudjlida** (2009). Semantic annotations for web services discovery and composition. *Computer Standards and Interfaces*, **31**, 1108 – 1117.
58. **WSDL** (2001). web services description language. URL <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>.
59. **WSDL4J** (). Open source project. URL <http://sourceforge.net/projects/wsdl4j/files/WSDL4J/>.
60. **Yager, R. R., G. Gumrah, and M. Z. Reformat** (2011). Using a web personal evaluation tool “PET” for lexicographic multi-criteria service selection. *Knowledge-Based Systems*, **24**(7), 929 – 942. ISSN 0950-7051. URL <http://www.sciencedirect.com/science/article/pii/S0950705111000311>.
61. **Zeng, C., W. Ou, Y. Zheng, and D. Han** (2010). Efficient web service composition and intelligent search based on relational database, 1–8.

62. **Zhou, C., L.-T. Chia, and B.-S. Lee** (2004). Qos-aware and federated enhancement for uddi. *International Journal of Web Services Research (IJWSR)*, **1**(2), 58–85.