ALGORITHMS FOR SEMI-AUTOMATIC WEB SERVICE COMPOSITION:

DATA MEDIATION AND SERVICE SUGGESTION

by

RUI WANG

(Under the Direction of Eileen T. Kraemer and John A. Miller)

ABSTRACT

This dissertation presents a semi-automatic Web service composition approach, which works by ranking all the candidate Web service operations and suggesting service operations to a human designer during the process of Web service composition. The ranking scores are determined by computing sub-scores related to inputs/outputs, data mediation, functionality and precondition/effects. A formal graph model, namely IODAG, is defined to formalize an input/output schema of a Web service operation. Three data mediation algorithms are developed to handle the data heterogeneities arising during Web service composition. The data mediation algorithms analyze the schema of the input/output of service operations and consider the structure of the schema. Typed representations for the data mediation algorithms are presented, which formalize the data mediation problem as a subtype-checking problem. An evaluation is performed to study the effectiveness of different data mediation and service suggestion algorithms as well as the effectiveness of semantic annotations used to assist human designers composing Web services.

INDEX WORDS: Web service composition, data mediation, service suggestion, semantic annotations, SAWSDL

ALGORITHMS FOR SEMI-AUTOMATIC WEB SERVICE COMPOSITION: DATA MEDIATION AND SERVICE SUGGESTION

by

RUI WANG

B.S., Zhengzhou University of Light Industry, China, 1998

M.S., Xi'an Jiaotong University, China, 2005

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2011

ALGORITHMS FOR SEMI-AUTOMATIC WEB SERVICE COMPOSITION: DATA MEDIATION AND SERVICE SUGGESTION

by

RUI WANG

Major Professor:	Eileen T. Kraemer John A. Miller
Committee:	Hamid R. Arabnia Jessica C. Kissinger

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2011

DEDICATION

To my husband Trentin Bishop and my daughter Athena Bishop for their encouragement, support and love.

ACKNOWLEDGEMENTS

I feel so lucky to have Dr. Kraemer as my major professor. She is such a nice professor that she helps me with both school and personal life. When I first came to UGA, as a new international student, new country, new school, new classmates, different language, different custom, different life style and so many things just made me feel frustrated. She is always there helping me, encouraging me and never turns me down. I really appreciate and would like to give her many thanks for all her help to me.

I would also like to thank my co-major Dr. Miller for all his advice for my research. He is a very knowledgeable professor. He showed me the amazing technologies in the semantic Web and Web services. Meeting, talking and discussing with him always stimulates many new thoughts for my research.

I would like to give my appreciation to my other two committee members, Dr. Arabnia and Dr. Kissinger, for their time and help for my PhD study and research. Thanks to all my team members, classmates and friends for their help. Here is a list of some of them: Shefali, Srikalyan, Douglas, Zhiming, Jun, Haibo, Liren, Chaitanya, Haseeb, Maryam, Alok, Yonglong, Shiva, Cristina, Mark, Kelly, Sumedha, Haiming, Jeremy, Wenyuan, Jianxia, Haipan, Meng, Dong, Yong, Tianhao and etc.

Last but not the least, I want to thank my husband, daughter, mother, father, brother, mother-in-law, father-in-law and all my relatives for all their support and kindness to me. I am happy to have such a sweet family and so much love from all my family members and relatives.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS v

LIST OF TABLES vii

LIST OF FIGURES ix

CHAPTER

1 INTRODUCTION 1

2 MOTIVATING SCENARIO 16

3 SERVICE SUGGESTIONS 21

4 DATA MEDIATION 33

5 FORMAL SERVICE SPECIFICATION 61

6 SIMILARITY MEASURES 67

7 ARCHITECTURE AND IMPLEMENTATION 79

8 EVALUATOIN 85

9 RELATED WORK 102

10 CONCLUSIONS AND FUTURE WORK 111

BIBLOGRAPHY 118

APPENDICES 131

A SAWSDL ANNOTATIONS FOR WSDL 2.0 131

B RANKINGS OF CANDIDATE WEB SERVICES 132

C STATISTICAL DATA FROM EVALUATIONS 137

LIST OF TABLES

Page

Table 1.1: Allowable SAWSDL annotations for WSDL 1.1 8

Table 2.1: Candidate Web services 17

Table 8.1: Different cases of annotations 92

Table A.1: Allowable SAWSDL annotations for WSDL 2.0 131

Table B.1: Rankings from the first evaluator 132

Table B.2: Rankings from the second evaluator 133

Table B.3: Rankings from the third evaluator 135

Table C.1: Degree of overlap of request 1 for evaluation I 137

Table C.2: Degree of overlap of request 2 for evaluation I 137

Table C.3: Degree of overlap of request 3 for evaluation I 137

Table C.4: Degree of overlap of request 4 for evaluation I 137

Table C.5: Degree of overlap of request 5 for evaluation I 138

Table C.6: Degree of overlap of request 6 for evaluation I 138

Table C.7: Average degree of overlap of all six requests for evaluation I 138

Table C.8: Degree of overlap of request 1 for evaluation II 138

Table C.9: Degree of overlap of request 2 for evaluation II 139

Table C.10: Degree of overlap of request 3 for evaluation II 139

Table C.11: Degree of overlap of request 4 for evaluation II 140

Table C.12: Degree of overlap of request 5 for evaluation II 140
Table C.13: Degree of overlap of request 6 for evaluation II 141
Table C.14: Average degree of overlap of all six requests for evaluation II 141
Table C.15: Degree of overlap of request 1 for evaluation III 142
Table C.16: Degree of overlap of request 2 for evaluation III 142
Table C.17: Degree of overlap of request 3 for evaluation III 142
Table C.18: Degree of overlap of request 4 for evaluation III 142
Table C.19: Degree of overlap of request 5 for evaluation III 142
Table C.20: Average degree of overlap of all six requests for evaluation III 143
Table C.21: Time (sec) comparison of three data mediation algorithms 143
Table C.22: Time (sec) comparison of different annotation cases 143
Table C.23: Time (sec) comparison of forward and bi-directional suggestion algorithms 143

LIST OF FIGURES

Page

Figure 2.1: A scenario: a WSC process that performs multiple alignments on related protein sequences of a given protein sequence 18

Figure 3.1: Forward suggestion 24

Figure 3.2: Backward suggestion 29

Figure 3.3: Bi-directional suggestion 31

Figure 4.1: An input message that will be fed by the output message 40

Figure 4.2: XSD build-in data type hierarchy 43

Figure 4.3: Graph G is sub-graph homeomorphic to graph H 47

Figure 4.4: Algorithm for approximated labeled sub-tree homeomorphism 50

Figure 4.5: An example for path-based data mediation algorithm 60

Figure 5.1: Precondition and effects for operation "*getIds*" 64

Figure 5.2: Algorithm to compute current state for a process 65

Figure 5.3: States in a BPEL process 66

Figure 6.1: Different types of subsumption relationships: exact match is a special case of plug-in match and plug-in match is a special case of subsumed-by match 73

Figure 6.2: The decay graphs of concept coverage similarity of four cases 74

Figure 6.3: Examples of four cases in formula 6-4 75

Figure 7.1: System architecture 80

Figure 8.1: Degree of overlap of three data mediation algorithms for request 1 (selected operation: "getIds") 88
Figure 8.2: Degree of overlap of three data mediation algorithms for request 2 (selected operation: "array2string") 88
Figure 8.3: Degree of overlap of three data mediation algorithms for request 3 (selected operation: "fetchBatch") 89
Figure 8.4: Degree of overlap of three data mediation algorithms for request 4 (selected operation: "run") 89
Figure 8.5: Degree of overlap of three data mediation algorithms for request 5 (selected operation: "getResult") 89
Figure 8.6: Degree of overlap of three data mediation algorithms for request 6 (selected operation: "base64toString") 89
Figure 8.7: Average degree of overlap of all six requests for three data mediation algorithms 90
Figure 8.8: Compare different annotation cases - request 1 (selected operation: "getIds") 94
Figure 8.9: Compare different annotation cases - request 2 (selected operation: "array2string") 94
Figure 8.10: Compare different annotation cases - request 3 (selected operation: "fetchBatch") 95
Figure 8.11: Compare different annotation cases - request 4 (selected operation: "run") 95
Figure 8.12: Compare different annotation cases - request 5 (selected operation: "getResult") 96
Figure 8.13: Compare different annotation cases - request 6 (selected operation: "base64toString") 96
Figure 8.14: Compare different annotation cases - average 97
Figure 8.15: Forward vs. bi-directional suggestion algorithms in request 1 (selected operation: "getIds") 100

Figure 8.16: Forward vs. bi-directional suggestion algorithms in request 2 (selected operation: "array2string") 100
Figure 8.17: Forward vs. bi-directional suggestion algorithms in request 3 (selected operation: "fetchBatch") 100
Figure 8.18: Forward vs. bi-directional suggestion algorithms in request 4 (selected operation: "run") 100
Figure 8.19: Forward vs. bi-directional suggestion algorithms in request 5 (selected operation: "getResult") 101
Figure 8.20: Forward vs. bi-directional suggestion algorithms - average 101
Figure 9.1: NetBeans manual data mediation GUI 103
Figure 9.2: NetBeans BPEL designer 104

CHAPTER 1

INTRODUCTION

Web services provide a standard way to publish, discover and invoke diverse software or applications distributed on the Internet. Many Web services have been developed and are available on the Web. Reusing existing Web services for new tasks can speed up people's work. However, in many cases, a complex task may require several Web services working together. Web service composition (WSC) targets this issue by reusing existing Web services and composing them into a process. Web services have well-defined interfaces and are platform and programming language independent, which makes them perfect functional building blocks for WSC.

Composing Web services is not an easy job in general. Over the last decade, a considerable amount of research has been performed on the problem of WSC. Part of the work on WSC is concerned with making it easier to design and develop Web service compositions by increasing the automation level of WSC.

The focus of this dissertation is the development of techniques and algorithms for WSC that have some degree of automation, in order to reduce the work of WSC. The approach presented is to make timely and effective service suggestions and handle data mappings during the design of a WSC process. Using the SAWSDL W3C standard [1], semantic annotations are used in service suggestion algorithms to assist a designer in developing WSC processes. Three data mediation algorithms are developed to establish data mappings during WSC and support the suggestion algorithms. A concept similarity measure is used to

calculate similarity scores for the data mediation and service suggestion algorithms. The service suggestion algorithm also offers the option for users to specify preconditions and effects to guide service suggestions.

1.1. Web Services

Web services have been widely used in many different areas including both business and science. For example, in business, Amazon[1] provides many Web services for its cloud computing services, and the USPS[2] offers various Web services for customers to handle shipping related processes. In the science, EBI[3], DDBJ[4], NCBI[5] and KEGG[6] host many Web services for biologists to retrieve or analyze biological data.

Web services can be categorized as either SOAP Web services and RESTful Web services. REST stands for Representational State Transfer. RESTful Web services use the HTTP protocol and interact with other RESTful Web services through a set of standard HTTP operations, e.g., GET, POST, PUT and DELETE. RESTful Web services focus on stateful resources rather than messages or operations. The Web Application Description Language (WADL)[7] may be used to describe the interfaces of RESTful services.

According to W3C [2], the definition of a SOAP style Web service is: "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its interface description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards." This dissertation

[1] Amazon Web services: http://aws.amazon.com/
[2] USPS Web services: http://www.usps.com/webtools/
[3] EBI Web services: http://www.ebi.ac.uk/Tools/webservices/
[4] DDBJ Web services: http://www.xml.nig.ac.jp/index.html
[5] NCBI Web services: http://www.ncbi.nlm.nih.gov/entrez/query/static/esoap_help.html
[6] KEGG Web services: http://www.genome.jp/kegg/soap/
[7] WADL: http://www.w3.org/Submission/wadl/

concentrates on the SOAP style Web services, but the algorithms can be extended to RESTful Web services, as stated in the Future Work section (Chapter 10).

The Web Service Description Language (WSDL) is an XML format language used to describe the interface of a Web service. A WSDL file of a Web service abstractly specifies the operations offered by the Web service as well as sufficient information to allow these operations to be invoked, including input, output, binding, etc.

There are two major versions of WSDL, i.e., WSDL 1.1[8] and WSDL 2.0[9]. The main elements of WSDL1.1 are <*service*>, <*port*>, <*binding*>, <*portType*>, <*operation*>, <*message*> and <*types*>. WSDL2.0 has substantial differences from WSDL1.1: WSDL2.0 also offers support for RESTful Web services. WSDL2.0 also has different XML elements, e.g., the <*message*> element is removed; <*portType*> is changed to <*interface*> and <*port*> is changed to <*endpoint*>. However, software support and implementations are still rare for WSDL 2.0 compared to those for WSDL1.1. This dissertation work is based on WSDL1.1, but the algorithms presented here can be extended to WSDL2.0 and further implemented as discussed in the Future Work section (Chapter 10).

SOAP once stood for Simple Object Access Protocol in version 1.1, but not in version 1.2[10] due to the less than ideal choice of work (e.g., it is not really object-oriented). It is a protocol for exchanging structured information with SOAP style Web services across the Internet. A Web service client can invoke a Web service by sending/receiving SOAP messages to/from the Web service. A SOAP message has an XML

[8] WSDL 1.1: http://www.w3.org/TR/wsdl
[9] WSDL 2.0: http://www.w3.org/TR/wsdl20/
[10] SOAP 1.2: http://www.w3.org/TR/soap12/

format and can be exchanged over various Internet application layer protocols, such as HTTP. SOAP is designed to be independent of any particular program model or implementation.

Universal Description Discovery and Integration (UDDI) is a platform independent XML-based registry that can be accessed over the Internet. UDDI enables businesses to register Web services in the registry and discover Web services listed in the registry through SOAP messages. Web services listed in the registry can be discovered based on the information provided by the businesses, e.g., operation name, business name, etc. UDDI was proposed as a core Web service standard. However, usage of UDDI registries has diminished over the years.

1.2. Ontology

In the computer science area, an ontology is a formal knowledge representation for describing the world or part of it, or as Gruber [3] puts it, "a specification of a conceptualization." It provides a shared vocabulary to model the world through a set of concepts, their properties and the relationships between those concepts. An ontology language is a knowledge representation language used to specify the ontology. Currently, the Web Ontology Language (OWL) is the chosen language for the ontology layer of the Semantic Web. Therefore, the implementation presented in this dissertation works with OWL. Another reason we chose OWL is because of the large number of high quality available tools (e.g., Protege[11] for editing, Pellet[12] for reasoning/correctness checking, OntoViz[13] for visualization and Jena[14]/OWL-API[15] for programmatic access, etc.)

[11] Protege: http://protege.stanford.edu/
[12] Pellet: http://clarkparsia.com/pellet/
[13] OntoViz: http://ontoviz.sourceforge.net/
[14] Jena: http://jena.sourceforge.net/

OWL is developed following the Resource Description Framework (RDF)[16] and RDF Schema (RDFS)[17], as well as earlier ontology language projects including Ontology Interchange Language (OIL) [4], DARPA Agent Markup Language (DAML)[18] and DAML+OIL[19]. However, OWL has more facilities for expressing meaning and semantics than these earlier languages. All the elements of OWL, e.g., class, property, individual, can be expressed as RDF resources and identified by URIs. OWL is intended to be used for information processed by computer. It aims to make it easier for machines to automatically process and integrate information on the Web.

Endorsed by W3C, OWL currently has two versions, i.e., OWL 1[20] and OWL 2[21]. OWL 1 includes three sublanguages with increasing levels of expressiveness, namely, OWL Lite, OWL DL and OWL Full. OWL 2 has a very similar overall structure to OWL 1, but adds following new features: increased expressive power for properties, extended support for datatypes, simple metamodeling capabilities, extended annotation capabilities, and keys. The basic elements of OWL 2 include classes, properties, individuals, and data values. OWL 2 has three sublanguages, namely, OWL 2 EL, OWL 2 QL, and OWL 2 RL. The EL acronym reflects its basis in the EL [5] family of description logics, logics that provide only Existential quantification. The QL acronym reflects the fact that query answering can be implemented by rewriting queries into a standard relational Query Language. The RL acronym reflects the fact that reasoning can be implemented using a standard Rule Language. Each of them trades off different aspects of expressive power in return for different computational and/or implementation benefits.

---

[15] OWL API: http://owlapi.sourceforge.net/
[16] RDF: http://www.w3.org/RDF/
[17] RDFS: http://www.w3.org/TR/rdf-schema/
[18] DAML: http://www.daml.org/
[19] DAML+OIL: http://www.w3.org/TR/daml+oil-walkthru/
[20] OWL 1: http://www.w3.org/TR/owl-features/
[21] OWL2: http://www.w3.org/TR/owl2-overview

1.3. Semantic Web Services

WSDL is the standard description language for the interface of a Web service. It mainly deals with syntax and technical aspects such as the schema of messages or the addressing of certain operations. WSDL lacks support for semantics to enable automation that would allow machines to sufficiently understand the descriptions of Web services. Therefore, adding semantics to represent the requirements and capabilities of Web services and to address the heterogeneity challenges is essential for achieving (semi) automated discovery and composition. To bridge the gap between the standards for Web services and the Semantic Web, a semantic Web service model must transform normal Web services into semantic Web services. Semantic Web services have been studied for several years and a few well-known semantic Web service models/languages have been developed, including Web Service Modeling Ontology (WSMO) [6], OWL-S [7] / DAML-S[22], Semantic Web Services Framework (SWSF)[23], and SAWSDL [1] / WSDL-S [8]. We selected SAWSDL/WSDL-S for this work, mainly because it is lightweight and easier for users to generate (annotating to existing WSDL files rather than creating new files).

SAWSDL stands for Semantic Annotation for WSDL and XML Schema. Based on WSDL-S, SASWDL provides flexible mechanisms for users to annotate existing WSDL documents with semantic concepts. Semantic content is added to service descriptions in a simple and lightweight manner as first suggested in the work on WSDL-S [8]. Annotations to a semantic model (e.g., an OWL ontology) are added at a few strategic points in a WSDL specification. In essence, SAWSDL provides reference mechanisms via extensibility elements in WSDL, i.e., *<interface>*, *<operation>*, *<message>*, etc., to

[22] DAML-S: http://herman.w3.org/services/daml-s/0.9/daml-s.html
[23] SWSF: http://www.w3.org/Submission/SWSF/

make them point to the semantic concepts defined in the externalized domain models for services. This allows designers to add as many or as few semantic annotations as they wish and can help disambiguate the description of Web services during automatic discovery or composition of Web services.

SAWSDL provides three extended attributes to WSDL elements, to enable semantic annotation of WSDL components, namely, "modelReference", "liftingSchemaMapping" and "loweringSchemaMapping." Table 1.1 shows the commonly annotated parts of WSDL 1.1 (see Appendix A for the commonly annotated parts for WSDL 2.0).

- **ModelReference:** This extension attribute specifies the association between a WSDL component and the concepts in a semantic model. It can be used to semantically annotate different parts of WSDL/XSD documents, such as <*complexTypes*>, <*simpleTypes*>, <*element*> and <*attribute*> tags in XSD as well as <*portType*>, <*operation*>, and <*fault*> tags in WSDL to relate them to the ontology.
- **LiftingSchemaMapping**: This attribute specifies how to map an XML component to an individual in an ontology (instance of a class). In this sense, the data content is lifted from the XML schema definition (XSD) level to the ontological level.
- **LoweringSchemaMapping**: This attribute specifies how to map an individual in an ontology to an XML component, in other words, to lower the data content down to the XSD level.

These two annotations, "liftingSchemaMapping" and "loweringSchemaMapping," apply only to XSD. They specify mappings between XML (input/output of a Web service) and an ontology. These mappings work in tandem to facilitate data mediation. If one operation wishes to send data to a second operation, if they cannot communicate directly due to the heterogeneities between their output and input, they may be

able to communicate indirectly by first lifting the output data to the ontological level and then lowering it back down in the form required by the succeeding operation. However, this technique has its limitations, i.e., it requires the lifting mapping to fill in enough properties of the ontological class so that the lowering mapping will not fail due to missing property values. In other words, the lifted output values plus default values must cover all the needed lowered input values.

Table 1.1. Allowable SAWSDL annotations for WSDL 1.1

Annotation / Tag	**modelReference**	**lifting SchemaMapping**	**lowering SchemaMapping**
<portType>	Yes	No	No
<operation>	Yes	No	No
<message>	Yes	Yes	Yes
<part>	Yes	Yes	Yes
<element>	Yes	Yes	Yes
<complexType>	Yes	Yes	Yes
<simpleType>	Yes	Yes	Yes
<attribute>	Yes	No	No
<fault>	Yes	No	No

Besides the above three extension attributes defined in SAWSDL, WSDL-S provides two more types of annotations [8]:

- **Preconditions**: Preconditions are required to be satisfied (ensured to be true) prior to the execution of a Web service operation, if the Web service operation has a precondition annotation.
- **Effects**: Effects are what will be true upon the successful execution of a Web service operation. Precondition and Effect are specified as child elements of <*operation*>. They are used to describe

the semantics of Web service operations. Preconditions and effects are primarily used in automatic service composition or discovery.

Category: An extension attribute on the <*interface*> element in WSDL-S, it consists of service categorization information that could be used when publishing a service in a Web Services registry such as UDDI. It functions in the same way as a "modelReference" annotation on the <*interface*> in SAWSDL.

1.4. Web Service Composition and Research Motivation

For complex real world tasks and applications, multiple Web service operations are often required. For instance, if a dealer uses a USPS Web service to ship goods to his customer, several Web services have to be used, e.g., to validate the customer's mailing address, request a shipment, make a payment, make an appointment to pick up goods, and check status. In the biological domain, even for a task as simple as using a BLAST program to find sequences similar to a given DNA or protein sequence, multiple Web service operations are required to be executed in a coordinated manner: one operation runs the main BLAST algorithm and returns a "jobid" that is an identifier of the BLAST execution; a second operation is then used to obtain the actual results (a collection of ranked matching sequences). Web service composition is a way to compose many Web services into a process to perform a complex task.

1.4.1. Composition Issues

Creating a WSC process that meets all the requirements, such as the correct inputs/outputs or minimizing the cost of the process, etc. is not easy in general. In software engineering, requirements can be categorized as functional requirements or non-functional requirements. A functional requirement defines

the function of the software, such as inputs, outputs and behavior. If a generated WSC process meets all the functional requirements, the process will function correctly. On the other hand, a non-functional requirement describes the criteria to judge the software, e.g., cost, security, or reliability. It is good for a WSC process to meet all the non-functional requirements, but it does not ensure that the process will function correctly. Therefore, any WSC approach has to at least make sure that the created process meets the functional requirements to have the correct functionality.

To create a WSC process that functions correctly, users have to handle many problems, such as which Web services should be selected, how to connect them, and how the inputs of a Web service will be fed by the preceding Web services, etc. Studies have been performed by many researchers on data mediation and process mediation to tackle these problems of WSC (see details in related work chapter 9).

Data mediation for WSC addresses the problem of how the inputs of a Web service will be fed by the preceding Web services [9]. It focuses on the data communication between the Web services in a WSC process. Web services that are to be composed together often convey heterogeneous messages between them, which means that the output messages from one operation cannot directly feed into the input message of the succeeding operation. Data mediation is the process of resolving message heterogeneities and possibly transforming one message format into another.

Process mediation in this context focuses on construction of the whole process [10]. More specifically, it deals with which Web services will be used in the WSC process and how they are chained together.

Process mediation and data mediation are not completely independent of one another. For example, if two Web services are selected to be connected one after another in a process, the inputs of the second Web

service have to be correctly provided by the outputs of the preceding Web services, which is handled by data mediation.

Data mediation and process mediation can deal with more than just functional requirements. For instance, quality of service (QoS) based Web service composition, which is one approach for WSC, takes the QoS parameters into consideration during WSC [11]. A process mediator may use the more reliable (given as QoS parameters) Web services in the process to increase the stability of the whole WSC process or use the cheaper Web services to reduce the total costs. The cost and stability are all non-functional requirements.

1.4.2. Automation of WSC

To make it easier for a user to create a WSC process, several GUI designers have been developed, such as ActiveVOS[24], NetBeans BPEL designer[25], Oracle BPEL designer[26], and Taverna[27]. GUI designers are useful because users can visually create a WSC process through drag and drop or menus or other types of graphical interfaces. However, creating a WSC process is still painstaking work, even using GUI designers. Currently, these designers do not offer sophisticated support for partial automation or computer assisted development of compositions. Users still have to handle data dependencies and control structure and feed outputs to inputs, etc., which are not easy. Still, there is room for increased automation/assistance to be incorporated into these GUI tools, based upon research done in automatic and semi-automatic composition.

[24] ActiveVOS: http://www.activevos.com/
[25] NetBeans BPEL: http://soa.netbeans.org/soa/
[26] Oracle BPEL: http://www.oracle.com/technetwork/middleware/bpel/overview/index.html
[27] Taverna: http://www.taverna.org.uk/

Studies have been performed to increase the automation level for WSC to ease users' work when composing a WSC. According to the automation level, WSC approaches can be categorized as manual approaches (with GUI), automatic approaches and semi-automatic approaches. Manual approaches let users manually select services and compose them together into a WSC process using GUI designers such as those mentioned above.

In automatic Web service composition, given a request created by a user, the WSC process will be generated automatically. The request formulates the description of the problem and all other required knowledge to create the WSC process. There is no manual intervention during the WSC generation process. Thus, it hides the complexity of the service composition from the user and reduces the development time and effort for the new WSC process. Automatic Web service composition is a challenging and a rather interesting problem. Many researchers have been working on it, such as [11-19]. They mainly map the WSC design problem into a planning problem, for which a considerable amount of research in AI has contributed to ever improving planners. A planner will look for actions/operations that will take the state of a WSC process from the initial state to a goal state. An operation may be invoked, if the current state satisfies both the precondition and input requirements of the operation. When invoked, an operation may produce output and/or change the current state (i.e., apply its effects).

From a practical point of view, the fruits of research on automatic WSC are likely to come some distance in the future. Some current limitations include (1) the difficulty of specifying control dependencies (branching, looping, etc.), (2) the handling of data heterogeneity between outputs and inputs, (3) and the difficulty of giving complete and correct specifications of operations and goals. A skeptic might even view automatic Web service composition as a modern version of automatic programming [20]. Automatic

programming is an area of Artificial Intelligence that began in the 1960's and is still a work in progress. On the other hand, the granularity of the problem in the case of Web service composition is much coarser, leading to smaller state spaces and hence improved chances for success.

Semi-automatic Web service composition recognizes the complexity of the WSC design problem and works by combining human and machine capabilities. It has some degree of automation compared to manual approaches, but aims to be more practical to use than automatic approaches. As discussed in the related work chapter, some work on semi-automatic WSC focuses on increasing the automation of data mediation, and other work concentrates on increasing the automation of process mediation (see Chapter 9 for details).

This dissertation presents a WSC approach that fits in the semi-automatic category. A user starts with a GUI designer and augments it with computer assistance at critical points in the design process to add a new service operation to a partially designed WSC process. Semantic annotations on input, output, functionality, precondition and effects (IOFPE) are used to rank service operations that may be plugged in at a particular point in a WSC process. A user can pick the desired Web service operations and then connect them into a process. Three different data mediation algorithms are developed to automatically handle the data flow as well as support the service suggestion algorithms. An evaluation is presented in the dissertation to compare these three data mediation algorithms.

Because human-computer interaction is enabled during WSC, some non-functional requirements may be included indirectly by the user's decision. For example, a user may pick the second choice of the suggested Web services because the user might know the second is cheaper than the first suggestion. Therefore, fewer requirements might need to be encoded as machine understandable documents that will be

handled by the program. The fewer documents that the user has to create, the less work for the user. Moreover, direct user interaction with a computer reduces the possibility of misunderstanding between humans and computers caused by errors or incomplete specifications in the documents.

In this work, minimally, the annotations may simply be "modelReferences" to ontological concepts that capture the notion of a semantic type with a well-defined notion of subsumption determined by description logic reasoning. Additional annotations may be provided to specify lifting and lowering schema mappings that enable more Web services to communicate with each other. Finally, if provided, preconditions and effects can be used for additional checking and/or local planning (at this point we do not attempt to create an entire WSC process using planning). In other words, our approach is incremental and based on the amount of semantics that Web service providers and WSC designers are willing to specify.

In this dissertation, another evaluation of the different degrees of semantic annotations for their effectiveness of service suggestions has also been performed. The study aims to find some ideal degree of semantic annotations that result in effective results with less effort to create.

1.5. Structure of This Dissertation

The rest of this dissertation is organized as follows: Chapter 2 introduces a motivating scenario that will be used in the rest of this dissertation. Chapter 3 covers the details of service suggestion algorithms. Three data mediation algorithms are presented in Chapter 4. Chapter 5 gives a formal service model, which is a base model for our service suggestion and data mediation algorithms. Similarity measures are discussed in Chapter 6, which are used by both the service suggestion and data mediation algorithms. Chapter 7 briefly describes the implementation and architecture of this work. Three evaluations are given in Chapter 8

to study the relative effectiveness of the three data mediation algorithms, the tradeoff between the annotation complexity and the effectiveness of service suggestions as well as the performance of the different suggestion algorithms. Related work is discussed in Chapter 9 followed by conclusions and future work presented in Chapter 10.

CHAPTER 2

MOTIVATING SCENARIO

In this chapter we develop a motivating scenario that will be used as an example to explain the ideas and algorithms presented in the rest of this dissertation. Furthermore, since it is moderately complex and practically useful, it will be used in the evaluations of our approach.

The goal in the scenario is to discover more information about a particular protein sequence, such as Multiple Sequence Alignment (MSA) of its related known proteins. A WSC process can be developed to find this information. The input to this WSC process would be a protein sequence of interest. The WSC process would utilize two popular bioinformatics tools: BLAST [21] and ClustalW[28]. First, BLASTP would search the given protein sequence against a user-specified database of protein sequences, for example the UniProt[29] dataset, to identify other similar proteins. The purpose of this is to assess the relatedness of the given protein to others based on sequence similarity. Moreover, a biologist would take the set of related proteins and align them to each other using ClustalW or some similar multiple sequence alignment tool to map the sequences to each other, and to then visualize the conserved and divergent regions at the sequence level. Further study, such as to illustrate how the sequences are related to each other in terms of evolution, could use the multiple sequence alignment to generate a phylogenetic tree.

[28] ClustalW: http://www.clustal.org/
[29] UniProt: http://www.uniprot.org

The workflow described above represents the two major analytical steps. However, there are numerous other actions which occur along this path, for example obtaining the sequences for the top-*k* hits identified in the BLASTP analysis to input into ClustalW and proper formatting of the multiple sequence alignment into a format that can be accepted for further analysis.

The Web services that could be used in this scenario are listed in Table 2.1 including some real Web services often used by biologists and bioinformaticians. Even though this scenario is from the biological domain, our approach, algorithms and prototype implementation are not limited to this domain, but can be applied to other areas as well.

Table 2.1. Candidate Web services

Web service	Provider	WSDL
WU-BLAST	EBI	http://www.ebi.ac.uk/Tools/webservices/wsdl/WSWUBlast.wsdl
WSDbfetch	EBI	http://www.ebi.ac.uk/ws/services/WSDbfetchDoclit?wsdl
ClustalW2	EBI	http://www.ebi.ac.uk/Tools/services/soap/clustalw2?wsdl
T-Coffee	EBI	http://www.ebi.ac.uk/Tools/services/soap/tcoffee?wsdl
NCBI BLAST	EBI	http://www.ebi.ac.uk/Tools/services/soap/ncbiblast?wsdl
WSConverter	Local	http://cs.uga.edu/~guttula/Galaxy/wsdlfiles/pe/WSConverter.wsdl

A possible solution to the above scenario is a WSC process depicted in Figure 2.1. The following service operations are required in the WSC process:

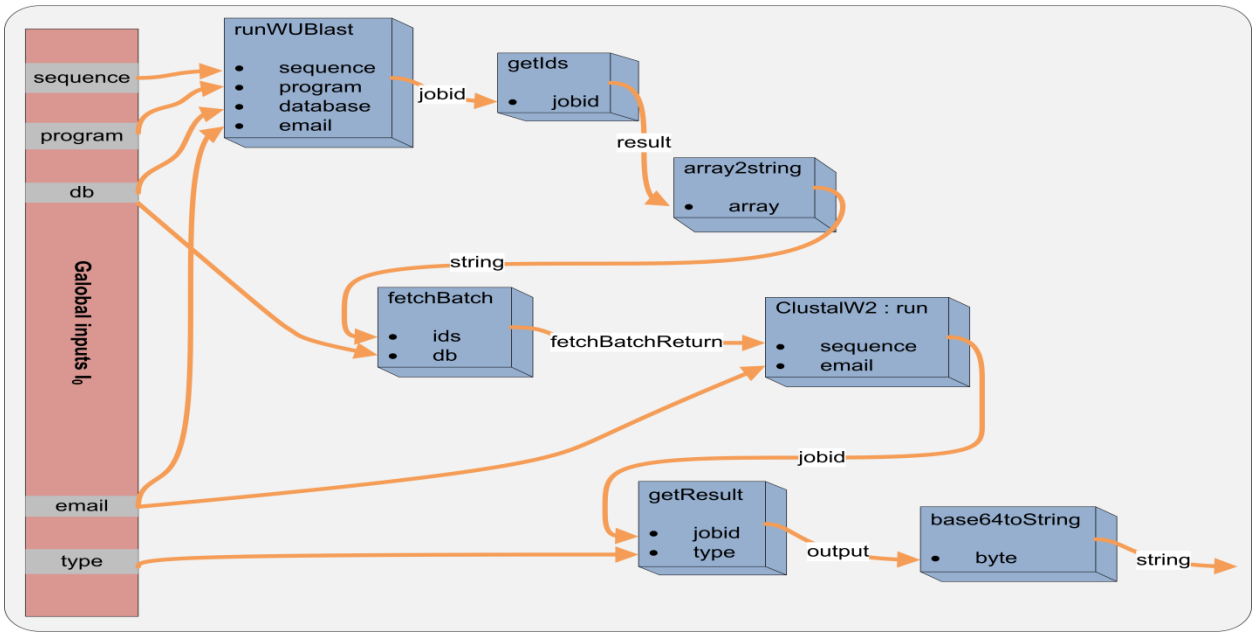

Figure 2.1: A scenario: a WSC process that performs multiple alignments on related protein sequences of a given protein sequence

1. **WU-BLAST: runWUBlast** is used to find similar sequences to a given protein sequence. This step performs a similarity search on the given protein sequence against protein databases such as UniProtKB/Swiss-Prot. The reason for doing this step is to assess the relatedness of the given protein sequence to other protein sequences, based on sequence similarity. The desired output of this step is a collection of protein sequences related to the given protein sequence; however, this operation gives "jobid" as the output instead of actual results.
2. **WU-BLAST: getIds** is used as a complementary operation to the first step. This step gives the sequence ids of the similar sequences found in the previous step. The reason we invoke this operation is that the results of the runWUBlast operation are stored in the server and if we wish to retrieve the details, e.g., obtain the sequence ids of the related proteins, we have to take the "jobid" returned from the last step as the input to invoke this operation.

3. **WSConverter: array2String** resolves the data format compatibility issue between step 2 and step 4. Step 2 gives an XML formatted array as the output, whereas, step 4 takes sequence ids as a string of comma separated values. This step converts an XML formatted array to a comma separated string, so the output of step 2 can be fed as the input to step 4.
4. **WSDbfetch: fetchBatch** takes sequence ids as input and retrieves the corresponding sequences as a string from the specified database. Until now we have the sequence ids of similar sequences, but in order to do a multiple sequence alignment, the actual sequences, in some format like FASTA, are required. The results from this step are all the related protein sequences for the given protein sequence provided in step 1.
5. **ClustalW2: run** performs multiple alignments of the similar sequences returned from the prior step. This step is vital, as comparing related proteins in a multiple alignment is the starting point for assessing the evolutionary relationships between the proteins. What we expect to learn by doing this step is to identify conserved sequence regions across a group of sequences to aid in establishing evolutionary relationships. From the results of multiple alignments, phylogenetic analysis can then be conducted to assess the sequence's shared evolutionary origins.
6. **ClustalW2: getResult** is a retrieval step for the "ClustalW2: run" operation. Similar to the "runWUBlast" operation, the multiple alignments produced by the " ClustalW2: run" operation are saved on the server. Given the "jobid," this operation will retrieve the user requested results according to the user specified value of the "type" parameter. Here, the value of "type," which specifies the type of the alignment output, is given as "aln-clustalw" to retrieve a ClustalW

alignment. The results returned are the multiple alignments of the given similar protein sequences.

7. **WSConverter: base64toString** is a decoding converter used as a translation step for the "ClustalW2: run" operation, which converts a base64 encoded string to an ordinary string. The converter is necessary because the multiple alignments returned from the prior step are presented as a base64 encoded string, which is not readable by humans.

In the scenario outlined above, a user may know of only some of the operations, e.g., 1 and 5. The user might also notice that there are some missing steps needed to make the WSC process complete, but may not know exactly what they are. Thus, if there is some mechanism that could suggest the missing steps 2, 3, 4, 6, 7, etc., it would be very helpful for the user. This scenario provides the motivation for the service suggestion work presented in this dissertation.

The inputs depicted in Figure 2.1 are all the required inputs of the Web services. Other inputs are defined as optional in the corresponding WSDL file and users do not have to give a value for them, e.g., the "runWUBlast" operation has almost twenty inputs, but most of them are optional. Our data mediation algorithm will differentiate the required and optional inputs and assign different weights to them. This alters their matching score contribution to the service suggestion score.

CHAPTER 3

SERVICE SUGGESTIONS

As discussed in section 1.4, process mediation is one of the major issues a user has to deal with to create a WSC process. It answers the questions about which Web services should be used in the WSC and in what manner they will be connected together. Most automatic WSC approaches handle process mediation automatically, utilizing AI planners such as [11-19]. They use domain knowledge and problem specifications to remodel the WSC problem as an AI planning problem. The planner will automatically generate a plan, which will be translated into a WSC process. However, the costs, difficulties and complexities of generating all the required annotations for the automatic approaches may limit their usability and practicality. Semi-automatic approaches try to reduce those limitations while still providing some degree of automation to ease the user's workload.

This chapter presents our semi-automatic process mediation approach. More specifically, it suggests Web services to aid the user designing a WSC process. When designing a WSC process using a graphical design tool, a user may ask the system to suggest services that can be plugged into the WSC process at a particular place and receive a ranked list of candidate services. Conversely, the user may ask for feedback on a service that they have placed in the WSC process. For example, for our motivating scenario, let us suppose that it is clear that the designer wishes to run the operation for the BLAST and ClustalW Web services. They select these operations and drag them onto the design canvas. Now they need help. What

services may be needed to provide input data to BLAST and how does one get the necessary information out? How should this information be reformatted to serve as input to ClustalW and again how is the desired information extracted and formatted in the way the user wants? Many steps and details are required to complete the WSC process. Unfortunately, after the design is complete, many errors are likely to occur when the WSC process is running (debugging WSC processes, e.g., Business Process Execution Language (BPEL) processes, is more challenging than debugging ordinary programs). By using service suggestion algorithms, the human designer can be assisted in coping with these details and design steps can be checked to reduce the burdens of debugging to some degree.

The service suggestions are computed based on any level or combination of semantic annotations on the Web service operation's input, output, functionality, precondition and effects as well as syntactic information. Our service suggestion algorithms rank available Web services and suggest to the user those services scoring over a certain threshold. The ranking score is calculated based on the following three aspects: data mediation (input-output compatibility), functionality and precondition/effects. Users can ask the system to make suggestions as to which services to connect after, before, or in the middle of the current process, referred to as forward, backward and bi-directional suggestions, respectively.

In comparing the similarity of the annotations of the input and output messages, the service suggestion score is analogous to a Web service discovery score [22-24] that is used to rank services when discovering Web services. However, those discovery approaches only compare the annotations of the <*message*>. For example, [22] defines the search template of a service operation as $<N_{OP}, D_{OP}, Os_{st}, Is_{st}>$, where N_{OP} and D_{OP} are the name and description of the operation, and the output and input of the operation are described by Os_{st} and $Is_{st,}$ which ignore the structure of the input/output and only include the

name and the annotation of the <message> of the corresponding input/output. Our path-based data mediation algorithm traverses through the whole input/output message, which provides richer and more complete information for the input/output and results in a more accurate matching score. Our service suggestion and ranking algorithms start where discovery algorithms leave off. In discovery, often, the goal is to find several relevant services. Our goal, though, is to minimize the need for an experienced programmer. Therefore, our suggestion criteria must be appropriate to the need to connect the suggested services into a WSC process. Prior work on discovery considers this in a limited way (e.g., plug-in match in OWL-S MX [24]), but we go further in this direction by considering type safety and data mediation within our service suggestion algorithms. Service selection also differs from service discovery in that discovery finds services matching a request specification, while composition-oriented selection tries to find services that fit (possibly on multiple sides) into a growing WSC process.

3.1. Forward Suggestion

As shown in Figure 3.1, the forward suggestion recommends a Web service operation (OP_x) to be placed after a specific operation or, more generally, the current process. The forward suggestion ranking score (S) is calculated as a weighted sum of three sub-scores, as indicated by formula (3-1).

$$S = w_{dm} \cdot S_{dm} + w_{fn} \cdot S_{fn} + w_{pe} \cdot S_{pe} \qquad \text{(formula 3-1)}$$

where w_{dm}, w_{fn} and w_{pe} are the weights for data mediation (dm), functionality (fn) and preconditions/effects (pe), respectively. Initially, $w_{dm} = w_{fn} = w_{pe} = 1/3$, they can also be trained by machine learning algorithms after gathering enough experimental data (see future work section 10.2).

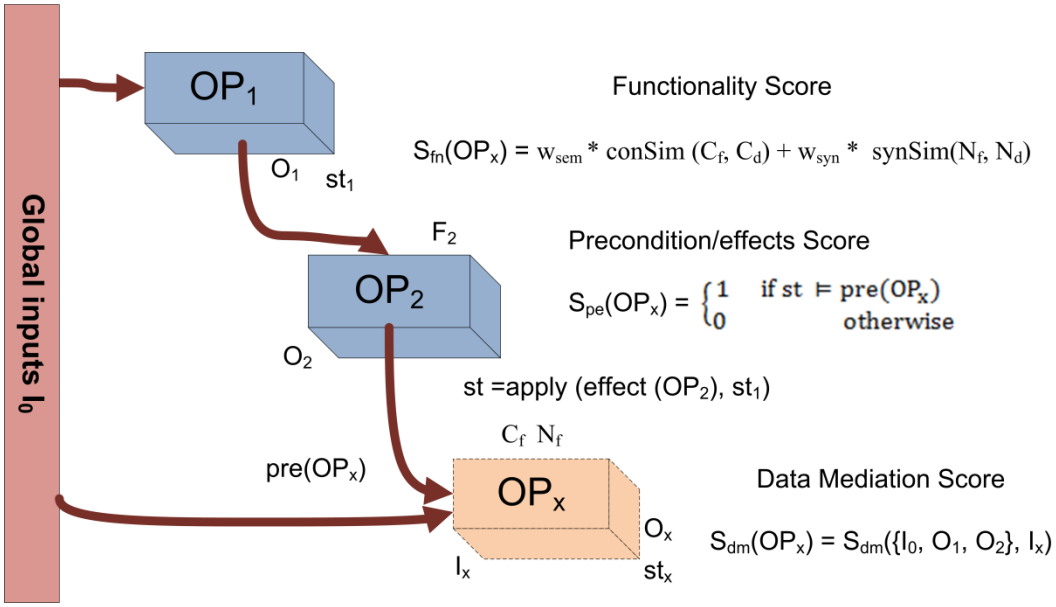

Figure 3.1. Forward suggestion

Using SAWSDL, each operation has a syntactic name implicitly indicating functionality and a semantic annotation more precisely indicating functionality. Therefore, the S_{fn} score is calculated based on the functionality of the Web service operation, which includes both semantic and syntactic functionality of the Web service operation as shown in formula (3-2).

$$S_{fn}(OP_x) = w_{sem} \cdot S_{sem} + w_{syn} \cdot S_{syn} = w_{sem} \cdot conSim(C_f, C_d) + w_{syn} \cdot synSim(N_f, N_d) \quad \text{(formula 3-2)}$$

where w_{sem} and w_{syn} are the weights for the semantic and syntactic functionality score, $w_{sem} = w_{syn} = 0.5$; C_f is the semantic functionality annotated on the candidate service, N_f is the syntactic functionality (name) of the candidate service, C_d/N_d is the user desired functionality that can be a keyword (N_d) or an ontological concept (C_d); *conSim* is the concept similarity and *synSim* is the syntactic similarity (see Chapter 6 for detail). If the user does not provide a desired functionality (C_d/N_d), the S_{fn} score will be set to zero.

For each operation of a Web service, its semantic functionality is specified using an ontological concept C_f. This concept is the value of the "modelReference" provided with the <*operation*> tag in the SAWSDL document. If the user provides a concept for the desired functionality (C_d) for the operation to be added, it will be compared to the functionality annotation (C_f) of each candidate operation.

The syntactic functionality of a Web service operation is determined by the value of the attribute "*name*" of the <*operation*> tag, e.g., <*operation name= "getIds"*>, i.e., "*getIds*." If the user provides a keyword (N_d) for the desired functionality, it will be compared with the name of each candidate operation N_f using the syntactic similarity measure (*synSim*) (see Section 6.1). The syntactic functionality score is the syntactic similarity score between this and the user desired functionality, N_d. Many string metrics can be used, such as N-Gram [25], or the Jaro-Winkler algorithm [26] (see Section 6.1 for further discussion of different string metrics).

Since our approach is applicable to various types and levels of annotations, the further expression of formula 3-2 will be shown as below. There are three cases of annotations for the calculation of the functionality score: the first is that no desired functionality (both C_d and N_d) is presented. In this case, both the semantic and syntactic functionality scores will be zero, so the final functionality score will be zero. In the second case, the user provided the desired functionality (N_d or both C_d and N_d), but there is no "modelReference" annotation (C_f) for the operation tag in the SAWSDL. Hence, the semantic functionality score S_{sem} will be zero and $S_{fn} = w_{syn} \cdot S_{syn}$. The third case is that all the required annotations to calculate the functionality score are provided, i.e., desired functionality and "modelReference" for the operation.

The S_{dm} sub-score of formula 3-1 is calculated based on the data mediation algorithm. A higher score means the input message of the candidate operation achieves a better match than its competition. It utilizes

our data mediation algorithm to retrieve the score for every input component of the candidate operation. The weighted sum of the scores for all input component will give the S_{dm} score as specified in formula 3-3. There are three data mediation algorithms presented in this dissertation, i.e., leaf-based, structure-based and path-based data mediation. The detailed calculation for the input component may vary with the different data mediation algorithms (see Chapter 4 for detail).

$$S_{dm} = \sum_{i=1}^{3} [\, w_i \cdot \sum_{j=1}^{n_i} S_{ij} \,]$$

(formula 3-3)

where S_{ij} is the score of the j^{th} input that is either required ($i=1$), unknown ($i=2$) or optional ($i=3$) and n_1, n_2 and n_3 are the number of required, unknown and optional inputs, respectively ($n_1+n_2+n_3 = n$, the total number of inputs).

Inputs to a Web service operation may be specified as required or optional in the WSDL/XSD documents (unknown indicates our software could not determine whether the input was required or optional). A required input, defined in the XSD of the WSDL file, has attribute "*nillable = false*" or "*minOccurs = 1*." For example, *<xsd:element name = "program" nillable = "false" type = "xsd:string">* defines a required input "*program*" for the "*runWUBlast*" Web service operation in our scenario.

An optional input has attribute "*nillable = true*" or "*minOccurs = 0*." For instance, *<xsd:element minOccurs = "0" maxOccurs = "1" name = "exp" nillable = "true" type = "xsd:string">* describes the optional input "*exp*" for the "*runWUBlast*" Web service operation.

The unknown type of input has neither the "*nillable*" nor the "*minOccurs*" attribute defined in the WSDL file, so our program cannot detect whether the input is required or not. For example, *<xsd:element*

name = "type" type = "xsd:string"> defines an input without any hint about whether it is required or optional for the "*runWUBlast*" Web service operation.

We argue that the required inputs are more significant, the optional inputs are the least significant and the other type of inputs might or might not be significant since it is not clearly defined whether the input is required or not. Therefore, we give different weights for these three types of inputs as follows: required input weight $w_1 = 1/(n_1 + 0.8n_2 + 0.2n_3)$, unknown input weight $w_2 = 0.8/(n_1 + 0.8n_2 + 0.2n_3)$ and optional input weight $w_3 = 0.2/(n_1 + 0.8n_2 + 0.2n_3)$. The ratio between the three type of weights is: $w_r : w_u : w_o = 1:0.8:0.2$. All the weights can be adjusted or trained by machine leaning algorithms in the future after gathering considerable of experimental data (see the details in chapter 10).

The S_{pe} score of formula 3-1 is calculated based on preconditions, effects, and the states of the WSC process. These annotations require the use of WSDL-S [8], a richer specification which served as the basis for SAWSDL.

In our current work, which is less dependent on using a planner, we intend to allow more expressive preconditions and effects by using a subset of first order logic. We originally planned to use the Rule Interchange Format (RIF)[30], but to the best of our knowledge there is no mature RIF engine and Java support library available, so we use the Prolog language, the SWI- Prolog[31] engine and the JPL Java library[32] support for WSI- Prolog instead. Prolog is a well-established programming language. Pure Prolog is based on a subset of first-order predicate logic, Horn clauses, so it falls within the RIF Core. RIF also has

[30] RIF: www.w3.org/TR/REC-rdf-syntax/
[31] SWI-Prolog: http://www.swi-prolog.org/
[32] JPL: http://www.swi-prolog.org/packages/jpl/java_api/index.html

Prolog-like syntax. Therefore, in the future, it might be straightforward to translate the Prolog rules into RIF Core rules.

To calculate the S_{pe} score, the SWI- Prolog engine is used for logical reasoning. The S_{pe} score is decided by whether the current state (*st*) entails[33] the precondition of the candidate operation *pre*(OP_x) as presented in formula 3-4. A candidate operation (OP_x) will be connected after the current process.

$$S_{pe} = \begin{cases} 1 & if\ st \models pre(OP_x) \\ 0 & otherwise \end{cases} \qquad \text{(formula 3-4)}$$

The current state (*st*) is maintained for the current process in, for example, a Prolog knowledge base. The knowledge base can be queried for entailment. After a candidate operation is selected to be connected to the process, the current state (*st*) will be updated to a new state (*st'*). The new state (*st'*) is determined by applying the effects of the candidate operation *effect*(OP_x) to the current state (*st*) (see Chapter 5 for details).

3.2. Backward Suggestion

A backward suggestion is used to recommend a service operation (OP_x) to be placed before a particular service operation or, more generally, the current process as shown in Figure 3.2. The backward suggestion ranking score (*S*) is similarly calculated based on the three sub-scores used for forward suggestions, i.e., the data mediation (S_{dm}), functionality (S_{fn}) and preconditions/effects (S_{pe}) scores. However, there are some differences in calculating each sub-score.

[33] http://en.wikipedia.org/wiki/Math_symbol
entail, $A \models B$ means the sentence *A* entails the sentence *B*, that is in every model in which *A* is true, *B* is also true.

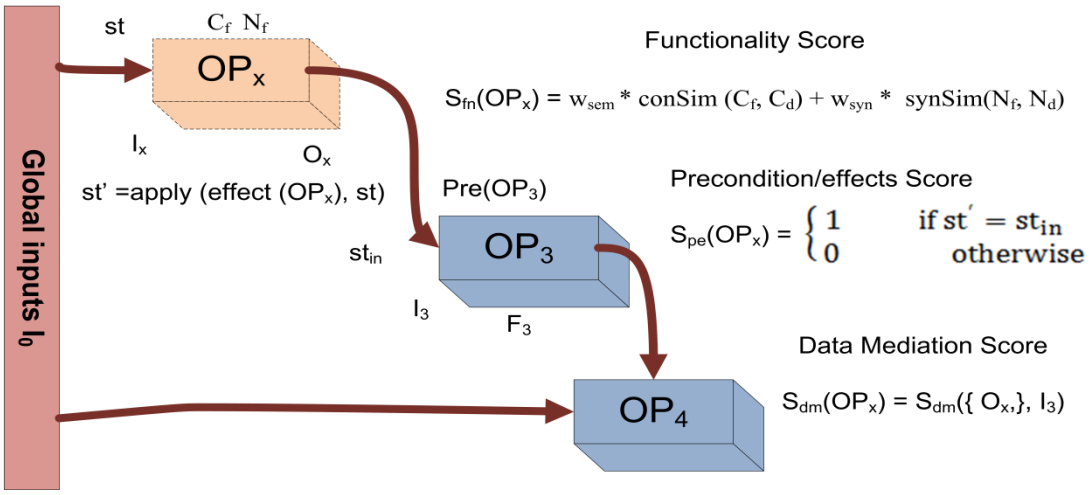

Figure 3.2. Backward suggestion

The data mediation score for a backward suggestion is based on how well the input message of the first operation (OP_3) of the current process is matched by the output message of the candidate operation (OP_x). The functionality score (S_{fn}) is calculated in the same way as for the forward suggestion using formula 3-2.

Compared to the forward suggestion score algorithm, the preconditions/effects score (S_{pe}) is decided based on the following criteria: It is a sufficient condition for the state (st') to be the same as the initial state (st_{in}) of the current process. The state (st') is obtained by applying the effects of the candidate operation to a state (st) that is given as the new initial state for the new process. If this condition is not satisfied after the candidate operation is placed in front of the current process, the initial state of the current process will be the state (st'). Since it is different from the original initial state, all the states after execution of each operation will be changed. The newly changed states might or might not entail the precondition of the succeeding operation. Moreover, the final state of the current process might be changed too, which might or might not be what the user desires. Therefore, if the state (st') is the same as the initial state of the current process, such that the new state (st') achieved by the execution of the candidate operation (OP_x) will automatically

entail the precondition of the first operation (OP_3) in the current process ($st' = st_{in}$ and $st_{in} \vDash pre(OP_3)$). If the candidate operation is placed in front of the current process, the initial state for the new process will be the state (*st*). The new state (*st'*) is produced by applying the effects of (OP_x) to the state (*st*), i.e., *st' = apply* (*effect* (OP_x), *st*).

$$S_{pe} = \begin{cases} 1 & if\ st' = st_{in} \\ 0 & otherwise \end{cases} \quad \text{(formula 3-5)}$$

where st_{in} is the initial state of the current state.

3.3. Bi-directional Suggestion

A bi-directional suggestion is used when one wishes to insert a service, so that it is wired in on both sides, its input on one side and its output on the other side. With respect to a chosen point of insertion, the current WSC process is logically divided into two parts. We name the front part the *WSC prefix* and the back part the *WSC suffix*. The user needs a Web service operation to be placed in between the WSC *prefix* and *suffix* to connect them together.

Figure 3.3 shows that the bi-directional suggestion is to recommend a service (OP_x) to place between two operations (OP_2, OP_3) in the current process or more generally a WSC *prefix* and *suffix*. Its suggestion score calculation also includes the three sub-scores: the data mediation (S_{dm}), functionality (S_{fn}) and preconditions/effects (S_{pe}) scores.

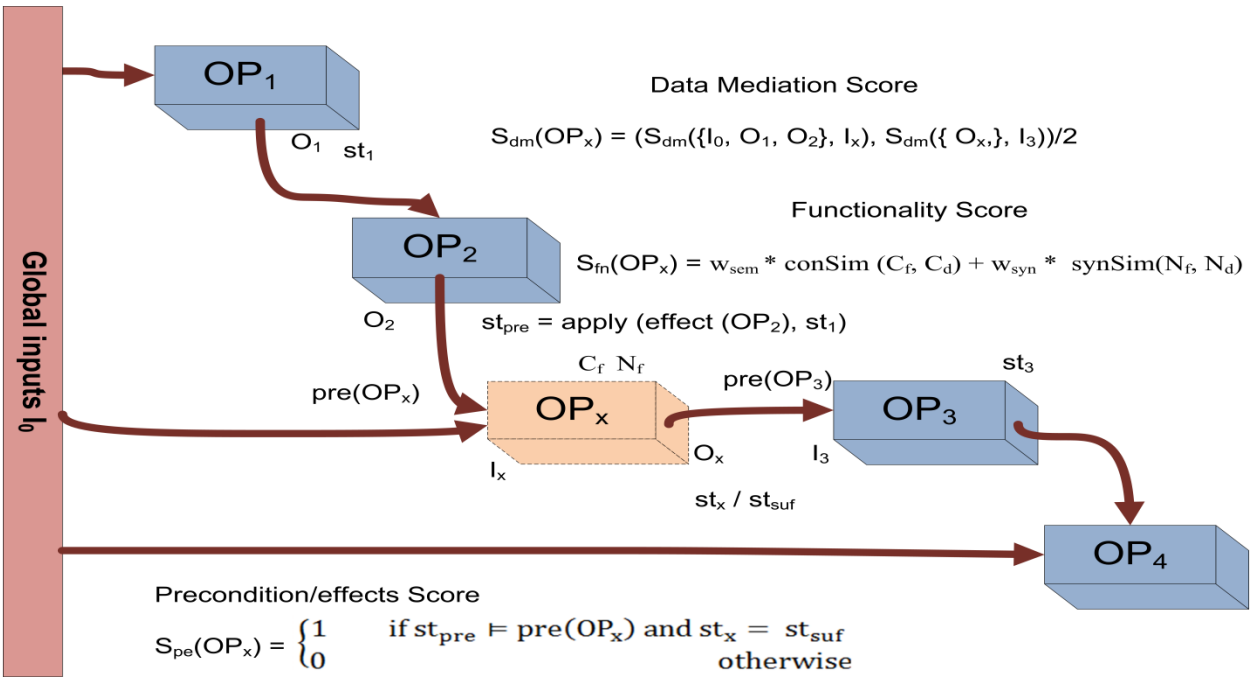

Figure 3.3. Bi-directional suggestion

The data mediation score algorithm for the bi-directional suggestions combines both the forward and the backward data mediation scores, S_{dm}^{for} and S_{dm}^{bak}, respectively. As shown in Figure 3.3, to calculate S_{dm}^{for}, the WSC prefix is taken as the current process for the forward data mediation, S_{dm}^{for} ($\{OP_1, OP_2\}$, OP_x) = S_{dm} ($\{I_0, O_1, O_2\}, I_x$). The WSC suffix is considered the current process to calculate the backward data mediation score S_{dm}^{bak} (OP_x, $\{OP_3, OP_4\}$) = S_{dm} ($\{O_x\}, I_3$). The final score for bi-directional data mediation is the average of these two scores, i.e., S_{dm} = (S_{dm} ($\{I_0, O_1, O_2\}, I_x$) + S_{dm} ($\{O_x\}, I_3$))/2.

The functionality score (S_{fn}) is computed in the same manner as the other two types of suggestions using formula 3-2. The preconditions/effects score (S_{pe}) is now based on satisfying two conditions as described in Fomula-3-6. The first is that the current state (st_{pre}) of the WSC prefix has to entail the precondition ($pre(OP_x)$) of the candidate operation (OP_x). The second condition, a sufficient condition, is

that the new state (st_x) has to be as same as the initial state (st_{suf}) of the WSC suffix, since it will affect the current state of the WSC suffix or the connections in the WSC suffix if its initial state is changed. The new state (st_x) is the state obtained by applying the effects of the candidate operation to the current state of the WSC prefix. Since the initial state (st_{suf}) of the WSC suffix entails the precondition of the first operation in the WSC suffix, the second condition implies that the new state (st_x) entails that precondition too.

$st_x = apply\ (effect(OP_x), st_{pre})$

$$S_{pe} = \begin{cases} 1 & if\ st_{pre} \vDash pre(OP_x)\ and\ st_x = st_{suf} \\ 0 & otherwise \end{cases} \quad \text{(formula 3-6)}$$

where st_{suf} is the initial state of the WSC *suffix*.

CHAPTER 4

DATA MEDIATION

To compose Web services, one of the major challenges is how to feed the inputs of a Web service from the outputs of preceding Web services. The way to tackle this problem is often referred as data mediation. More specifically, data mediation is the process of resolving message heterogeneities and mapping one message into another.

Deciding which output can be mapped to a specific input is not easy for many users. Therefore, we developed a data mediation approach, which finds the optimal mappings between outputs and inputs automatically for the user. In order to precisely address the mapping of the output of one service into the input of another service, we rely on prior work done in databases, i.e., data mediation [27] and programming languages, i.e., type theory [28]. Data mediation has a long history in the database field for data integration [27]. A common use is to send a query to a mediator that decomposes the query to send sub-queries to component databases. Each component database may have its own schema or even its own query language. It is the mediator's job to perform the necessary translations. In our case, the overall goal is the same: Have multiple services talk to each other and let the system act as a mediator, so that translations and format conversions need not be hand coded by the WSC designer every time a WSC process is developed. A good overview of the challenges involved in applying data mediation techniques

to Web service composition is given in [9], which itemizes the kinds of heterogeneities and proposes solutions.

In our data mediation approach, the input or the output of a Web service operation will be represented as a semantic structural data type. If the type of the output of a Web service operation is a subtype of the type of the input of another Web service, then the output can be fed to the input in a type safe manner. We define the rules for subtyping in our type system and depict our data mediation solution with the typed representations. In this work, type checking is performed at WSC design time using structural subtyping. Although structural subtyping [29] is more complex than nonstructural, it is more flexible and appropriate for Web services.

In this Chapter, a brief overview of type theory will be given at first. Section 4.2 will describe how the metadata of the input/output of a Web service operation is modeled as a graph. Based on this graph model, sections 4.3, 4.4 and 4.5 will describe our leaf-based, structure-based and path-based data mediation approaches, respectively.

4.1. Type Theory

Type theory [28] in computer science refers to the design, analysis and study of type systems. A type system associates values with types and seeks to guarantee that operations expecting certain types of values do not receive values that lead to type errors. Moreover, a type system specifies rules to control how the typed program may behave, which makes any behavior outside the rules illegal. Type systems provide some level of type safety through type checking, an important form of correctness checking.

Type checking and, more recently, type inference are two problems that have received considerable study [30-36] for type systems. Type checking is the process of verifying and enforcing constraints on types. When assigning a value to a variable of a type, type checking decides whether the assignment is type safe. Type inference, on the other hand, is used to determine the type of an expression. Type checking can be done either statically at compile time or dynamically at runtime. It can be performed based on name equivalence (most programming languages) or structural equivalence (O'Caml[34] and Modula-3[35]). The best known type-inference algorithm is probably the one developed by Hindley [37] and later rediscovered and extended in a programming language context by Milner [38]. The algorithm is today an integrated part of compilers for the ML family of functional programming languages.

Subtyping appears in a variety of type systems [28, 31-33, 36, 39-42]. One of the common uses is "coercions," such as automatic conversion from integer numbers to real numbers and subclassing in object-oriented languages. If T_x is the type of variable x and T_y is the type of variable y, then it is type safe to make the assignment $x = y$, *iff* T_y is a subtype of T_x. We use "$\preccurlyeq$" to denote subtype, i.e. $T_x \preccurlyeq T_y$ means the type T_x is subtype of type T_y.

The subtype relation is reflexive: $\vdash T \preccurlyeq T$

and transitive: $\dfrac{\vdash T_1 \preccurlyeq T_2 \quad \vdash T_2 \preccurlyeq T_3}{\vdash T_1 \preccurlyeq T_3}$

A typing rule has the form of: $\dfrac{J_1\ J_2 \cdots J_n}{J_{n+1}}$ where all the J_i are typing judgments. The denominator is the conclusion and the numerator is all of the hypotheses. The rule indicates that the typing judgment

[34] Objective Caml: http://caml.inria.fr/ocaml/
[35] Modula-3: http://modula3.org/

J_{n+1} will hold if all the hypothesis type judgments J_1 J_2 … J_n hold. A typing judgment is of the form: $\vdash t : T$, which means that the term *t* has the type *T*. Here, *T* is a type. A typing judgment will hold if it is an axiom or if it can be inferred by some typing rules.

4.2. Graph Model of Input / Output Types

In this subsection, we will develop a graphical model to represent WSDL/XSD data structures for Web service inputs and outputs. The input/output types of a Web service operation are specified in a WSDL document/file using WSDL/XSD tags. In WSDL 1.1 the following tags are used: <*message*>, <*part*> and related XSD tags, while in WSDL 2.0, only XSD tags are used.

In choosing an appropriate graphical representation the obvious candidates are Trees, Directed Acyclic Graphs (DAGs) or general Directed Graphs. For complete generality, the schema specified using WSDL/XSD for an input or output would require a general directed graph, since recursive structures are permitted in XSD. However, since they are less common and using general directed graphs will make some of the problems we are trying to solve NP-Hard, we will limit our work to DAGs. More specifically, node-labeled DAGs as below:

Definition: An **Input / Output DAG (*IODAG*)** is a node-labeled DAG *G* = <*N*, *E*, *md*, *Metadata*>, which represents the schema of an input or output of a Web service operation described using WSDL/XSD.

- $N = \{n_1, \ldots, n_m\}$ is a set of nodes in the DAG. Each node of *G* corresponds to an element defined in a WSDL/XSD document (more specifically, to the <*message*>, <*part*> in the WSDL 1.1 file

and to the <*element*> in the XSD defined / imported in the WSDL 1.1 file). A parser has to be coded to extract all these element.

- $E \subset N \times N$ is a set of directed edges in the DAG. The edges indicate the relationships between the elements defined using WSDL/XSD. For example, the <*message*> element has one or more <*part*> elements in WSDL, so there will be corresponding edges from the <*message*> node to every <*part*> node in the DAG. Each edge $e_{ij} \in E$ is defined as a tuple $e_{ij} = <n_i, n_j>$ indicating a directed edge from n_i to n_j, where $n_i, n_j \in N$
- *md:* $N \rightarrow$ *Metadata* is a function assigning labels to nodes. The labels are the metadata that are used in our service suggestion and data mediation algorithms.
- *Metadata = Name* $\times$ *XSDtypes* $\times$ *OWLtypes*

G is a rooted graph, which has exactly one root node. The root node corresponds to the <*message*> node in the WSDL 1.1 file. We consider the root as the top level. The next lower level nodes that are children of the root node correspond to <*part*> nodes of the message in the WSDL file.

Definition: The **root node** $n_{root} \in N$ of an *IODAG* is the node that has no incoming edge.

Definition: A **leaf node** $n_{leaf} \in N$ of an *IODAG* is defined as a node that has no outgoing edge. The leaf node set $N_{leaf} \subseteq N$ is defined as the set of all the leaf nodes in an *IODAG*. We consider the leaf nodes as the bottom level nodes of the *IODAG*.

Definition: A **path** in an *IODAG* is an ordered list of nodes from a leaf node to the root node along the edges in the *IODAG*. It corresponds to the reversed path of a conventional path of a tree.

path = $(n_1, \ldots, n_i, \ldots, n_m)$

where $n_i \in N$, n_1 is a leaf node $n_1 \subseteq N_{leaf}$ and n_m is the root node $n_m = n_{root}$ in the *IODAG*

4.3. Leaf-based Data Mediation Approach

One aspect of WSC is data dependency analysis in terms of possible data flows within the WSC process. In a broad sense, data mediation finds out how the inputs of a Web service operation should be fed by the outputs of the preceding Web service operations. The simplest form of communication happens when one operation sends its output to the succeeding operation, which gets its input values from the output. A leaf-based approach tackles this simply by directly looking for a matching component from the output for each component of the input. For example, in our scenario, the "*getIds*" Web service operation takes a *jobid* as its input. If we look at its WSDL file, *<part name="jobid" type="xsd:string" />* is the element that actually defines this input, as the "*part*" element of the "*message*" element "*getIdsRequest*." Similarly, the *<part name="jobid" type="xsd:string" />* "part" element of the "*message*" element "*runWUBlastResponse*" defines the output of "*runWUBlast*" Web service operation that will be fed into the input of the "*getIds*" Web service operation. Basically, if we could find this pair of elements in the two WSDL files, make sure they match, and then assign the output element to the input element, we would be on our way to solving the data mediation problem.

Our leaf-based data mediation approach works in two stages. In the first stage, we parse the WSDL/XSD files to create IODAG graphical representations: one DAG for the input and one DAG for the output. (In the more general case of having multiple preceding services, the output would be represented as a list of DAGs.) The second stage involves some form of graph matching. Each input must be paired with some output (either in an exclusive or shared fashion) in a way that maximizes the overall quality of the match.

For the first problem, recall the graph model we defined for the input/output meta-data in a WSDL document in Section 4.2. The leaf nodes of an *IODAG* are always elements in its WSDL document, which define the elements that hold the value of the input or output. Therefore, the first step of our leaf-based data mediation approach is to parse the WSDL document and discover all the leaf nodes of the input / output. For details about parsing, please see the implementation section.

To deal with the second problem, our leaf-based approach somewhat naively ignores the non-leaf nodes and just collects all of the leaf nodes to form two sets, one for outputs and one for inputs. Moreover, it finds the elements from the output set matching the input set (structure-level match). If we impose the restriction that all the matches must be one to one (exclusive), this problem becomes a weighted bipartite graph problem, where the edge weights are the matching scores between every pair of elements. Thus, it can be solved by a typical weighted bipartite graph algorithm, such as the weighted Hungarian (also known as the Kuhn-Munkres (KM)) algorithm [43]. Otherwise, in the shared case, it can be solved by a simpler matching algorithm that runs in O(*mn*) time, where *m* is the number of leaves in the output and *n* is the number of leaves in the input (note, this ignores the time taken in computing the similarity measures). This is more efficient than the KM algorithm, which runs in cubic time [43].

The shared and exclusive formulations are defined as below. They indicate that finding the best match may be expressed as an optimization problem to pick the optimal pairings where $x_{ij} = 1$ ($x_{ij} = 0$ means this pairing was not chosen). The shared case uses a constraint to ensure that every input component is matched with exactly one output component. The exclusive case requires an additional constraint to ensure that an output component is fed to at most one input component.

shared: $max\{\sum_{i=0}^{m}\sum_{j=0}^{n} S_{ij} * x_{ij} \mid \sum_{i=0}^{m} x_{ij} = 1\}$

exclusive: $max\{\sum_{i=0}^{m}\sum_{j=0}^{n} S_{ij} * x_{ij} \mid \sum_{i=0}^{m} x_{ij} = 1 \ and \ \sum_{j=0}^{n} x_{ij} \leq 1\}$

where *m* is the number of leaves in the output and *n* is the number of leaves in the input.

For the element-level match (compare two leaves), the leaf-based data mediation approach utilizes both the syntactic information of the XML elements and the semantic annotations for the XML elements in the SAWSDL file. The data mediation score (S_{dm}) has been defined in formula 3-3. For leaf-based data mediation, the score is the weighted average of the similarity scores S_{ij} of the *n* matched pairs of nodes in the optimal matching. The element matching score (*ES*) defined in formula 4-5 is used as the S_{ij}.

Figure 4.1 shows an input message on the right that will be fed by the output message on the left. Our leaf-based data mediation algorithm will collect the leaf nodes and form two set: {N_4, N_5, N_6}, {N_8, N_0}. By comparing their labels, the best match for N_8 is N_5. N_4 and N_6 have the same matching score for N_0.

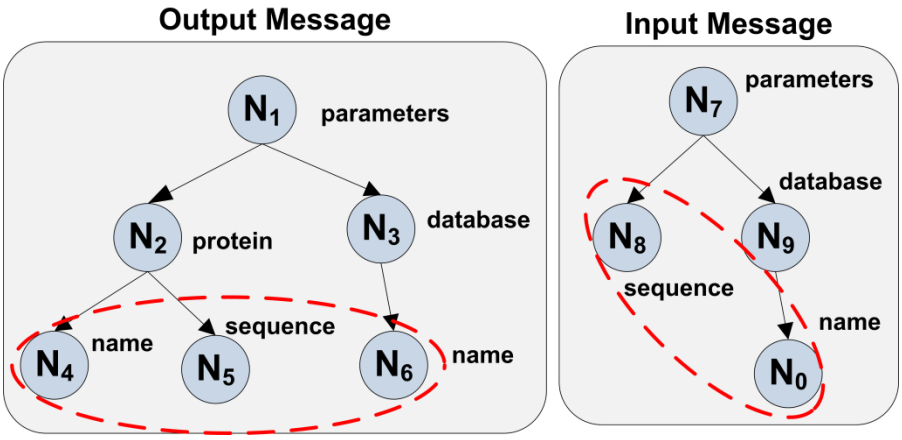

Figure 4.1. An input message that will be fed by the output message

Type checking of the leaves of *IODAGs* is one part of the type checking used in path-based data mediation presented later, so we will give a type representation to explain the details of the leaf-based algorithm using type checking as well as semantic annotations. Leaf-based data mediation has been implemented as an external engine for a GUI WSC designer, e.g., our previous work, WS-BioZard [44] utilized this leaf-based data mediation engine to handle data mediation for its WSC designer.

4.3.1. A Typed Representation for Leaf-based Data Mediation

Based on the *IODAG* defined in Section 4.2, we define a formal type representation for the meta-data of input / output of a Web service operation. It is similar to a conventional type system of a programming language. This type representation contains a set of base types and rules to judge the subtype relations between two types. This type representation will serve as the formal representation for leaf-based data mediation during WSC.

We define two basic types, a leaf type T_{leaf} and a leaf-based IO type T_{LIO}. The notation we use here is similar to the one used in [28, 30, 32, 33], as described in section 4.1.

Leaf type: The leaf type is the type for a leaf node in an *IODAG*. It is a structured data type, which is defined as a tuple.

$T_{leaf} = md\ (n_j) = <name, xsdtype, annotation>$,

where $n_j \in N_{leaf}$, $name \in Name$, $xsdtype \in XSDtypes$, $annotation \in OWLtypes$

The *name* field is a string. It is the value of the name attribute of the node in the corresponding WSDL document. The *annotation* expression is an ontology concept. It is the semantic annotation for the node, e.g., the value of the "modelReference" for the XML element in the SAWSDL document. The *xsdtype* field

corresponds to the value of the type attribute of an XML element in the WSDL document. A user-defined data type can be either a simpleType or a complexType, but for a leaf, the *xsdtype* expression can be empty, an XSD built-in data type or a simpleType. According to the XSD specification from W3C [45], XSD has 44 built-in data types and the subtype relationships between *xsdtype* expressions may be defined based on the XSD built-in data type hierarchy shown in Figure 4.2.

Leaf-based data mediation is based on the idea of only considering the XML elements in a SOAP message that contain values of an input/output of a Web service operation. The meta-data of these XML elements in a SOAP message are defined in their corresponding WSDL/SAWSDL and XSD documents. These meta-data correspond to the leaf nodes in an *IODAG* for a service operation. Therefore, the leaf type T_{leaf} is the type of these meta-data, i.e., the meta-meta-data.

Leaf-based IO type: The Leaf-based IO type is defined as an unordered set whose components correspond to all the leaf nodes in an *IODAG*, $T_{LIO} = \{leaf_1, \ldots, leaf_n \mid leaf_i : T_{leaf}\}$. An instance of IO type is a set of leaf type components that correspond to leaf nodes in the *IODAG*. The Leaf-based IO type will be used in our leaf-based data mediation as the structural type of the schema of the input/output of a Web service operation.

In leaf-based data mediation, the Leaf-based IO type T_{LIO} is the type of the meta-data of the input/output of a Web service operation. It is a set of leaf nodes that correspond to the meta-data of all the XML elements that will contain values of the input/output in a SOAP message to invoke the operation.

Below is a set of rules for inferring subtype relationships for the base types defined above as part of our typed representation. We use the same notations presented in section 4.1, e.g., subtype as "$\preccurlyeq$" and type equivalence as "$\equiv$."

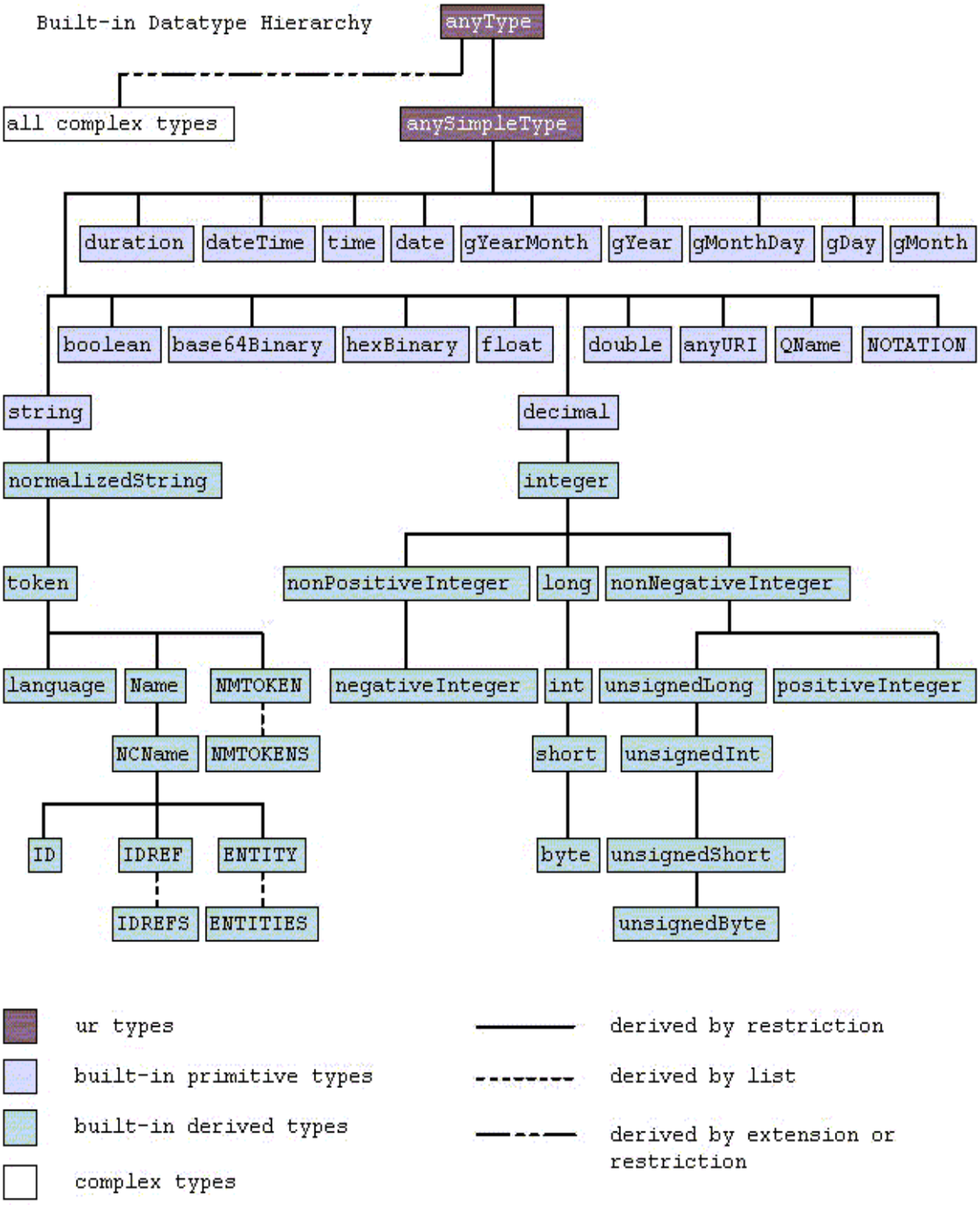

Figure 4.2. XSD built-in data type hierarchy [45] (ur types are user defined types)

Rules for subtype of type T_{leaf}:

$$\frac{\begin{array}{c}\vdash t, t' : T_{leaf} \\ \vdash t'[xsdtype] \preccurlyeq t[xsdtype] \\ \vdash t'[annotation] \sqsubseteq t[annotation]\end{array}}{\vdash t' \preccurlyeq t}$$

(formula 4-1)

The square bracket "[ ]" indicates the syntactic argument, e.g., $t[annotation]$ denotes the annotation of t. $\vdash t'[annotation] \sqsubseteq t[annotation]$ is a subsumption relationship between two semantic concepts in an ontology, and is discussed in the section on concept similarity measures (Section 6.2). $\vdash t'[xsdtype] \preccurlyeq t[xsdtype]$ is the subtype relationship between two built-in data types defined in XSD, as shown in Figure 4.2. This rule can be explained as that t' is a subtype of *t*, if *t* and t' are both instances of T_{leaf}, the *xsdtype* argument of t' is a subtype of the *xsdtype* argument of *t*, and the *annotation* of t' is subsumed by the *annotation* of *t*.

Formula 4-1 describes the elemental level comparison (an output component to an input component) for leaf-based data mediation. If the type of an output component is a subtype of the type of an input component of a succeeding operation, then it will be safe to feed this output component to this input component.

Rules for subtype of type T_{LIO}:

$$\frac{\vdash \tau, \tau' : T_{LIO} \qquad \dfrac{\vdash \forall t.\tau}{\vdash \exists t'.\tau'} \qquad \vdash t' \preccurlyeq t}{\vdash \tau' \preccurlyeq \tau}$$

(formula 4-2)

$\forall t.\tau$ indicates that t is any component of τ. Here, τ is an instance of leaf-based IO type T_{LIO}, which is a set of leaf nodes, so t has leaf type T_{leaf}. Similarly, $\vdash \exists t'.\tau'$ denotes there exists t' that is a component of τ'.

This subtype rule describes that for two leaf-based IO type instances τ' and τ, where τ' is a subtype of τ ($\tau' \preccurlyeq \tau$) holds, if for any component t of τ, there exists a component t' of τ' that is a subtype of t ($t' \preccurlyeq t$).

The leaf-based IO type is a structural type. The subtype rule for T_{LIO} actually defines the structural subtype relationships between the meta-data of the output and input. If the type of the output of an operation can be inferred to be a subtype of the input of another operation according to the subtype rule described in formula 4-2, it will be safe to feed the output to the input. This rule depicts how type checking finds the matching output for the input of the succeeding operation in a WSC process. For every elemental input, if there exists an elemental output of the preceding operation whose type is a subtype of the type of the elemental input, the whole structural output type will be a subtype of the whole structural input type.

4.4. Structure-based Data Mediation

The leaf-based data mediation approach only considers the leaf nodes in an *IODAG*, which leads to the following questions: Are all the other non-leaf nodes useless for data mediation? Does the structure of the *IODAGs* not matter for data mediation? To address these questions, we consider another approach for data mediation: structure-based data mediation, which utilizes a sub-graph homeomorphism algorithm.

In data mediation, determination of the similarity and/or type compatibility between the output type of a preceding operation and the input type of a succeeding operation can be modeled as a sub-graph

homeomorphism problem between two DAGs: *out.IODAG* (output type) and *in.IODAG* (input type). In other words, *out.IODAG* is sub-graph homeomorphic to *in.IODAG*, if *out.IODAG* contains a sub-graph that is homeomorphic to *in.IODAG*. The homeomorphism mapping from *out.IODAG* to *in.IODAG* specifies how the input of the succeeding operation will be fed by the outputs of the preceding operations. In the rest of this section, we will give a short introduction to sub-graph homeomorphism and then describe the algorithm used for our structure-based data mediation approach.

4.4.1. Sub-graph Homeomorphism

The concept of homeomorphism is a topological notion. In graph theory, a graph homeomorphism [46] preserves topological properties between two graphs. More specifically, a homeomorphism mapping between two graphs preserves the ancestor relations between the nodes in a graph.

Definition: The graph G and H are **homeomorphic** graphs, if there exists a graph G' that is isomorphic to graph H. Graph G' is obtained by performing a sequence of subdividing and smoothing operations on graph G.

Definition: Two graphs $G = (N, E)$ and $G' = (N', E')$ are **isomorphic** if there exists a bijection $f : N \rightarrow N'$ such that $(n_i, n_j) \in E$ *if* and only *if* $(f((n_i), f(n_j)) \in E'$.

Definition: Let $e = <n_i, n_j>$ be an edge of G. The **subdividing** operation adds a new vertex n_k to the graph G and replaces edge e with two new edges $<n_i, n_k>$ and $<n_k, n_j>$.

subdividing $(G, <n_i, n_j>) = G\,(N \cup \{n_k\}, E - \{<n_i, n_j>\} \cup \{<n_i, n_k>, <n_k, n_j>\})$

Definition: Let n_k be a node of degree of two, such that two edge $e' = <n_i, n_k>$ and $e'' = <n_k, n_j>$ meet at n_k. The **smoothing** operation replaces edges e' and e'' with a new edge $e = <n_i, n_j>$.

smoothing $(G, n_k) = G(N - \{n_k\}, E \cup \{<n_i, n_j>\} - \{<n_i, n_k>, <n_k, n_j>\})$

Based on the definition of graph homeomorphism, sub-graph homeomorphism can be defined as:

Definition: Graph *G* is **sub-graph homeomorphic** to graph *H* if *G* contains a sub-graph *G'* that is homeomorphic to *H*.

Figure 4.3 shows how to apply sub-graph homeomorphism to solve the same data mediation problem showed in Figure 4.1. Graph G' is a sub-graph of G. Graph G' is homeomorphic to H, because smoothing node N_2 from graph G' will obtain an isomorphic graph of H. Therefore, graph G is sub-graph homeomorphic to graph H and the node matches between G' and H indicate how to feed the output to input of the succeeding operation.

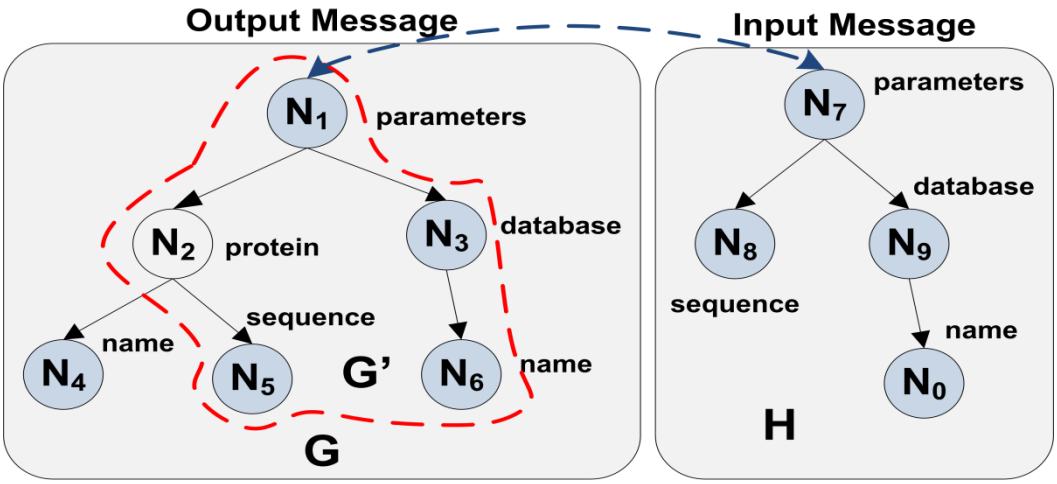

Figure 4.3. Graph *G* is sub-graph homeomorphic to graph *H*

For this work, we apply an algorithm for sub-graph homeomorphism to *IODAGs*. The subdividing and smoothing operations change an *IODAG* by either inserting or removing a node that has one incoming and one outgoing edge, so that the structure is not changed in a fundamental way. Of course, in

the optimization, a penalty can be applied to minimize these "small" changes (see section 4.4.2 for more details).

4.4.2. Algorithm for Structure-based Data Mediation

The decision problem of sub-graph homeomorphism is known to be NP-complete [47]. However, when the problem is restricted to be a sub-tree homeomorphism (which is often the case of an IODAG representing an input/output of a Web service operation), the algorithm to solve it can be efficient, e.g., a cubic algorithm is presented in [48]. Therefore, we will begin with the case where the IODAGs are trees and apply the approximate labeled sub-tree homeomorphism (ALSH) algorithm from [48] to data mediation.

Figure 4.4 describes the approximate labeled sub-tree homeomorphism algorithm. The inputs of the algorithm are two trees: $G = (N_G, E_G, r_G)$ with root r_G and $H = (N_H, E_H, r_H)$ with root r_H, where H represents the input of an operation and G represents the output of the preceding operation. The algorithm returns the root of a sub-tree of G that has the highest similarity score (*RScore*) to H. To compute the *RScore*, the algorithm traverses H and G in a post-order and computes the similarity score between every pair of nodes $n_H \in N_H$ and $n_G \in N_G$. If n_H and n_G are both leaves, the similarity score will be calculated as the elemental similarity (*ES*) score in formula 4-5 (see Section 4.5.1). Otherwise, the procedure ComputeScoresForNode is invoked to compute the similarity score unless n_H has a higher level than n_G in the tree, which implies that it is impossible for the tree with root n_G to be isomorphic to the tree with root n_H via smoothing operations and the score will be negative. The ComputeScoresForNode procedure computes the *Rscore* (n_H, n_G) using the following steps: First, it constructs a bipartite graph between the

children of n_H and the children of n_G, and computes the *AssignmentScore*, which is the score of the optimal matching of the bipartite graph (divided by out-degree of n_H to make the value range from 0 to 1). Second, it computes the *BestChild*, which is the highest *RScore* between n_H and all children of n_G. Finally, the higher value between ($ES(n_G, n_H)$ + *AssignmentScore*)*/2* and (*BestChild* + ρ) will be the *Rscore* (n_H, n_G). $ES(n_G, n_H)$ is the elemental level similarity score between n_G, n_H, which is calculated using formula 4-5, and ρ = - 0.001 (used in our evaluation, section 8.3) is the penalty for removing the node n_G. The higher penalty will discourage removing nodes, and the lower penalty will encourage removing nodes.

Algorithm for approximate labeled sub-tree homeomorphism

Input: Rooted trees $G = (N_G, E_G, r_G)$ and $H = (N_H, E_H, r_H)$.
Output: The root of the sub-tree of G that has the highest similarity score to H, if G has a sub-tree that is homeomorphic to H.

```
for each node n_H of H in post-order do
  for each node n_G of G in post-order do
    if n_H is leaf then
      if n_G is leaf then
        Rscore(n_G, n_H) = ES (n_G, n_H)
      else
        Rscore(n_G, n_H) = ComputeScoresForNode(n_G, n_H)
      end if
    else
      if Level(n_H) > Level(n_G) then
        Rscore(n_G, n_H) = −∞
      else
        Rscore(n_G, n_H) = ComputeScoresForNode(n_G, n_H)
      end if
    end if
  end for
end for
```

Procedure ComputeScoresForNode(n_G, n_H)
Let k denote the out-degree of node n_H and m denote the out-degree of node n_G.
if $k > m$ **then**
    AssignmentScore = $-\infty$

else

Construct a bipartite graph G with node bipartition X and Y such that $X = \{x_1, \ldots, x_k\}$ is the set of children of n_H, $Y = \{y_1, \ldots, y_m\}$ is the set of children of n_G, and every node $x_i \in X$ is connected to every node $y_j \in Y$ via an edge whose weight is $Rscore(y_j, x_i)$.

Set *AssignmentScore* to the maximum weight of a matching in G divided by k.

end if

$BestChild = max_{j=1}^{m}\{Rscore(y_j, n_H)\}$

return $max\{(ES(n_G, n_H) + AssignmentScore)/2, BestChild + \rho\}$

Figure 4.4. Algorithm for approximated labeled sub-tree homeomorphism [48]

Pinter et al. [48] prove that this algorithm has an $O(m^2 n + mn\log(n))$ time complexity. They point out that the dominant cost of this algorithm is for the weighted bipartite graph algorithm. They employ a variant of the Hungrian Algorithm from Fredman et al. [49] to solve the weighted bipartite graph problem, which has an $O(m^2 n + mn\log(n))$ time complexity, where m and n are the sizes of the two sets of nodes in the bipartite graph.

The data mediation score (S_{dm}) is calculated using formula (4-3), which is the optimal sub-tree homeomorphism similarity score between a sub-tree of G and H. The node of G with the optimal score is the root of the sub-tree that is homeomorphic to H.

$$S_{dm} = max_{j=1}^{m} \{Rscore(y_j, r')\} \qquad \text{formula (4-3)}$$

where $y_i \in N_{G,}$ a node of the output tree and r' is the root of the input tree H.

4.5. Path-based Data Mediation

To summarize what we have so far, each of our approaches has limitations: The leaf-based approach is simple and efficient, but it does not consider the structure of the input/output, which may lead to frustrating runtime errors in the WSC process. The structure-based approach has the potential to reduce runtime errors

that are due to input/output structural incompatibility, but is too strict and may rule out some useful compositions. Our third approach decomposes the input and output into many simple paths, rather than the sub-trees used in the structure-based approach. In the evaluation chapter, an evaluation will be performed to compare these three approaches.

While the leaves of a DAG provide important information, certainly the full path from a leaf to the root of the DAG should provide more useful information. Comparing full paths of a DAG is somewhat like comparing sub-graph of this DAG. In the worst-case, the number of paths in a DAG may be exponentially large in terms of the number of nodes, |N|. However, when the structure of the *IODAG* is restricted to be planar (often the case with common data structures), the number of paths is bounded by a polynomial [50]. This allows the development of polynomial time algorithms for path matching.

4.5.1. A Typed Representation for Path-based Data Mediation

This section describes in detail our path-based data mediation approach through a typed representation. The input and output of a Web service operation will be represented as semantic ‘structural’ data types based on the *IODAG*. (Note, structural type equivalence, e.g., as in Modula-3 and OCaml, is more general and complex than conventional name equivalence found in most programming languages). If the output type of an operation is a *subtype* of the input type of another operation, then the output can be fed to the input in a type-safe manner. Compared to the leaf-based data mediation, this approach still considers type compatibility between the output of the current WSC process and the input of a candidate Web service operation. The difference is that for the structure-level typing, this approach respects the structure

of the input/output, using a path-based algorithm to find a mapping between the two graphs corresponding to the two types.

We extended the type representation described in section 4.3.1 to represent our path-based data mediation approach. More types are defined below including non-leaf type $T_{non\text{-}leaf}$, path type T_{path} and path-based IO type T_{PIO}.

Non-leaf type: The non-leaf type is the type for any node in *IODAG* except leaf nodes. It is a structured data type, which is defined as a tuple.

$T_{non\text{-}leaf} = \prod_{name,\ annotation} md\ (n_j) = <name, annotation>$

where $n_j \in N_{non\text{-}leaf}$, $annotation \in OWLtypes$

Similar to the leaf type T_{leaf}, the *name* field is the syntactic name of the XML element in the corresponding WSDL document, and is the value of the "*name"* attribute of the XML element.

The *annotation* field is a concept from an ontology. It is the semantic annotation for the XML element in the corresponding SAWSDL document. More specifically, it is the value of a "modelReference" attribute in a SAWSDL document.

In contrast to a leaf type, a non-leaf type has no *xsdtype* component. The syntactic types of the corresponding XML elements for the non-leaf nodes are either empty or ComplexType. Since we currently have not yet defined a type hierarchy for all *XSDtypes* including complex types, we have to project the *XSDtypes* out of the *non-leaf* type.

We add non-leaf types for the non-leaf nodes in *IODAGs* to complement the type representation defined in Section 4.3.1, since our path-based data mediation respects the structure containing all the nodes in an IODAG including both leaf and non-leaf nodes.

Path type: Based on our earlier definition of a path in an *IODAG*, a path type is defined as an ordered list of nodes: $T_{path} = (t_1 \ldots t_i \ldots t_m)$, where t_1 has the leaf type ($t_1 : T_{leaf}$) and $t_2, \ldots t_i, \ldots, t_m$ has the non-leaf type ($t_i : T_{non\text{-}leaf}$ *for* $i > 1$). An instance of T_{path} has a corresponding path in the *IODAG*.

Path-based IO type: The path-based IO type is defined as the set of all paths in one or multiple *IODAG*(*s*): $T_{PIO} = \{ p_i \mid p_i : T_{path}, i \in [1 \ldots n] \}$. Every component in T_{PIO} is a path type T_{path}, and *n* is the total number of paths.

Path-based IO type actually sets up the mapping between the *IODAG* and a set of paths. The reverse of this mapping will transform an instance of path-based IO type to an *IODAG*.

Below are some rules for inferring subtype relationships for the base types defined above, as an extension of the type representation defined in Section 4.3.1. These rules are used in type checking for type compatibility at both structural and elemental levels.

Rules for subtype of type $T_{non\text{-}leaf}$:

$$\frac{\vdash t,\ t' \colon T_{non-leaf} \qquad \vdash t'[annotation] \sqsubseteq t[annotation]}{\vdash t' \preccurlyeq t}$$

(formula 4-4)

$\vdash t'[annotation] \sqsubseteq t[annotation]$ indicates those two concepts annotated on the two XML elements have a subsumption relationship in the ontology. Please check the concept similarity measures discussion (Section 6.2) for details.

The subtyping rule described in formula 4-4 can be explained as follows: if the *annotation* for a non-leaf type instance *t′* is subsumed by the *annotation* for another non-leaf type instance *t*, then *t′* is subtype of *t*.

This rule is used for the elemental level type checking for data mediation, where *t* and *t'* represent two WSDL/XSD elements to be compared based on the definitions in their WSDL files. The subtype relationship between *t* and *t'* will be used to judge the structure-level type checking.

Beyond the type checking for a subtype relationship between the two XML elements (*t, t'*), the element-level similarity score (*ES*) between t and t' is calculated using formula 4-5, which is applicable to the XML element that corresponds to a leaf node in IODAG as well.

$$ES(t, t') = w_1 \cdot conSim(t.annotation, t'.annotation) + w_2 \cdot synSim(t.name, t'.name)$$

(formula 4-5)

where w_1 and w_2 are the weights, which are both set to 0.5 initially, and *t* and *t'* represent the two XML elements to be compared. *conSim* is the concept similarity, as defined in Section 6.2.1. *synSim* is the syntactic similarity, as presented in Section 6.1, which is used to compute the similarity score between the names of *t, t'*.

Formula 4-5 can work with or without semantic annotation. If there is no annotation on either of the elements, the first part of the formula will be zero and the ES score will be calculated based only on the syntactic information, but with a lower value. This allows our data mediation approach to work with any level of semantic annotation, from fully annotated input/output to no annotation at all.

Rules for subtype of type T_{path}:

$$\frac{\vdash p, p' : T_{path} \quad \vdash p'[t_1] \preccurlyeq p[t_1] \quad \ldots \quad \vdash p'[t_m] \preccurlyeq p[t_m]}{\vdash p' \preccurlyeq p}$$

(formula 4-6)

Here, p and p' are two instances of path type T_{path}, m is the minimum length of the two path lengths of p and p', and $p[t_1]$ means the component t_1 of p, where $p[t_1]$ has leaf type T_{leaf} as defined in path type T_{path}.

This subtyping rule indicates that for two instances p and p' of path type $T_{path,}$ starting from the first pair of components of p and p', if the component of p' is a subtype of component p in every pair, then p' is a subtype of p.

This rule shows that to compare two paths, every pair of nodes in the two paths will be compared starting from the pair of leaf nodes. The similarity score of two paths (*SP*) is calculated as shown in formula 4-7. It is calculated by computing the optimal sum of node matches for an input path, node by node without gaps (allowing gaps reduces the efficiency of our algorithms.), starting at the bottom of the input path (leaf node) and retrieving the match score against the output leaf node. Then, the next higher input node is matched to the corresponding output node, until we reach the root of the input path. The score can be used to find the best matching score based on the comparison of paths.

$$SP(p, p') = \sum_{i=1}^{m} w_i \cdot ES(p[t_i], p'[t_i]) \qquad \text{(formula 4-7)}$$

where p and p' are the two paths to be compared. $m = min\{|p|, |p'|\}$ is the length of the shortest path of the two paths. *ES* is the element similarity score defined in formula 4-5. $p[t_i]$ denotes the i^{th} node of path p from leaf to the root. Similarly $p'[t_i]$ indicates the i^{th} node of path p'. w_i is the weight of the i^{th} node. A geometric series is used for all these weights, which are decreasing from the leaf node to the root node and $w_1 + \ldots, + w_m = 1$, e.g., $w_1 = 0.571$, $w_2 = 0.286$ and $w_3 = 0.143$. The weights are set to prioritize the leaf nodes because the leaf nodes represent the type of the XML element that holds the value of an input/output.

Another issue we would like to mention here is the path alignment. The path comparison described above aligns the two paths to be compared at the leaf nodes, for the same reason that we set the weights prioritizing the leaf nodes. However, in some cases, aligning the two paths at nodes other than leaf nodes may get a higher matching score than aligning at the leaf nodes. In this case the output and input that are represented by the paths will need a converter in between. We plan to do more research on this path alignment issue in our future work.

Rules for subtype of type T_{PIO}:

$$\frac{\vdash \tau, \tau' : T_{PIO} \ , \quad \dfrac{\vdash \forall p.\tau}{\vdash \exists p'.\tau' \qquad \vdash p' \preccurlyeq p}}{\vdash \tau' \preccurlyeq \tau}$$

(formula 4-8)

Formula 4-8 is the subtyping rule for type $T_{PIO,}$ $\forall p.\tau$ specifies that p is any component of τ. τ has the type T_{PIO}, which has a set of paths, so p has the path type T_{path}. . This rule can be explained as, given two instance of T_{PIO} (τ and τ'), if for every component p of τ, there exists a component p' of τ' and p' is subtype of p, then τ' is subtype of τ. The subtyping rule for T_{path} is defined in formula 4-6.

The subtyping rule in formula 4-8 involves some major points of our path-based data mediation. Firstly, it implies how to resolve data mediation based on path comparisons. The path IO type decomposes the input/output into many paths based on *IODAGs*. The goals of data mediation now take on a different perspective. If for each path of type T_{path} for the input *IODAG* of an operation, we can find a matching path of type T_{path} from all the output *IODAGs*, we can in some sense feed the appropriate outputs into the operation. Although decomposing the problem into a set of paths (i.e., not fully considering the DAG or tree structure) may lose some structural information, e.g., sibling relationships, it

relaxes some restrictions to avoid screening out some useful compositions compared to the structure-based data mediation. One of our evaluations is intended to see whether this compromise is worthwhile.

Secondly, it specifies our path-based data mediation based on type checking. It presents how to infer subtype relationships between two T_{PIO} types. Since the two types are the type of an input and the type of an output data to be fed to the input, a subtype relationship indicates whether it is safe to transfer the data. This rule is used for the structure-level type checking. It is based on path level type checking (T_{path}) and elemental level type checking (T_{leaf}, $T_{non\text{-}leaf}$).

Thirdly, the rule in formula 4-6 requires that all pairs from nodes of two paths have a subtype relationships, which might be too strict. As a looser solution, we use formula 4-7 to calculate the path similarity score (*SP*) to find the best matching path for every path representing the input of a candidate operation (note, the best matching path might or might not be type safe). Based on the scores for all the best matching paths for the input paths, a data mediation score (S_{dm}) can be calculated using formula 4-9 (a rewritting of formula 3-3 with more details), which is part of the suggestion score, to rank all the candidate Web service operations. S_{dm} is the weighted sum of the best path scores, *BSP*, for the paths in the optimally matched input, where the matching is between the output and the input as shown in formula 4-9.

$$S_{dm} = \sum_{i=1}^{3} \left(w_i * \sum_{j=1}^{n_i} BSP(p_{ij}) \right)$$

(formula 4-9)

where p_{ij} is the path for the j^{th} input that is either required ($i=1$), unknown ($i=2$) or optional ($i=3$) and n_1, n_2 and n_3 are the number of required, unknown and optional inputs, respectively ($n_1+n_2+n_3 = n$, total number of inputs). Inputs to a Web service operation may be specified as required or optional in the WSDL/XSD documents (unknown indicates our software could not determine whether the input was required or optional). Since required inputs tend to be more significant than optional ones, the weights are determined as follows: $w_1 = 1/(n_1 + 0.8n_2 + 0.2n_3)$, $w_2 = 0.8/(n_1 + 0.8n_2 + 0.2n_3)$ and $w_3 = 0.2/(n_1 + 0.8n_2 + 0.2n_3)$.

Given a path type p in an input *IODAG*, we find the best matching path type p' in an output *IODAG*. $BSP(p)$ is the best score for path p, compared with the paths representing the outputs of all the operations and global inputs in the current WSC process. The score for this best match *BSP* is defined in formula 4-10.

$$BSP(p) = max\ \{SP(p, p') \mid p' \text{ is from } out.IODAG\} \quad \text{(formula 4-10)}$$

The rule in formula 4-8 requires all the paths of the input have to have a type compatible (subtype) path, which might be too strict and might filter out some possible Web service operations. We loosen this restriction through the application of formula 4-7 and formula 4-9. If only parts of the paths of the input have a type compatible path, we can still use formula 4-7 to find the best matching paths for other paths and use formula 4-9 to calculate the data mediation score to contribute to the service suggestion score. Therefore, all the candidate Web service operations will be ranked and have their data mediation score. At the same time the type compatibility for the input of each candidate operation is given to show whether this candidate operation is type safe to connect to the current WSC process, according to the subtyping rule defined in formula 4-8. If the candidate operation is not type safe to connect to the current process, we can still use formula 4-9 to calculate the data mediation score to suggest a ranked list of candidate

operations to the user. If the candidate operation is selected by the user, maybe not type safe, using formula 4-10 our data mediation engine can still find the best matching output for each input of the candidate operation.

Figure 4.5 gives an input message that need to be fed by the output message on the left, as in the example described in section 4.3. Now, we use path-based data mediation to solve it. In Figure 4.5, there are three paths for output: $p_1 = (N_4, N_2, N_1)$, $p_2 = (N_5, N_2, N_1)$, $p_3 = (N_6, N_3, N_1)$, and two paths for input: $p_4 = (N_8, N_7)$, $p_5 = (N_0, N_9, N_7)$. We need to calculate $BSP(p_4)$ and $BSP(p_5)$ to find the best matches for p_4 and p_5. To simplify the explanation of the path ranking algorithm, this example assumes that the ES scores are either one (for a perfect match) or zero (otherwise). As shown below, the best matching path for p_4 is p_2, and their matching score is 0.667. The best matching path for p_5 is p_3, with matching score 1. If we look at Figure 4.5, the best matching for N_0 (p_5) is N_6 (p_3), not N_4. While the path-based algorithm can find this matching correctly, the leaf-based algorithm cannot.

In Figure 4.5, if the input is the left half graph and output is the right half graph, our path-based data mediation can still find the best matching path for p_2 is p_4, best matching path for p_3 is p_5, with a lower matching score since p_1 has no match (need an additional global input). However, the structure-based algorithm is too strict for structure matching compared to path-based algorithm and will fail in this case because it has to find a sub-graph of the right half graph to match the whole left half graph. As we can see the right half graph is smaller than the left half graph and it is impossible to make it. Therefore, too strict structure matching, e.g., our structure-based algorithm, may rule out some useful services, and our path-based algorithm considers structure in a less strict way may have better performance. Our evaluation (section 8.3) also shows that path-based data mediation has a higher performance compared to the other

two data mediation algorithms.

$BSP(p_4) = max\ \{SP(p_4, p_1\ ), SP(p_4, p_2), SP(p_4, p_3)\}$

$= SP(p_4, p_2)$

$= 0.667 \cdot ES(N_{8,} N_5) + 0.333 \cdot ES(N_{7,} N_2)$

$= 0.667 + 0$

$= 0.667$

$BSP(p_5) = max\ \{SP(p_5, p_1\ ), SP(p_5, p_2), SP(p_5, p_3)\}$

$= SP(p_5, p_3)$

$= 0.571 \cdot ES(N_{0,} N_6) + 0.286 \cdot ES(N_{9,} N_3) + 0.143 \cdot ES(N_{7,} N_1)$

$= 0.571 + 0.286 + 0.143$

$= 1$

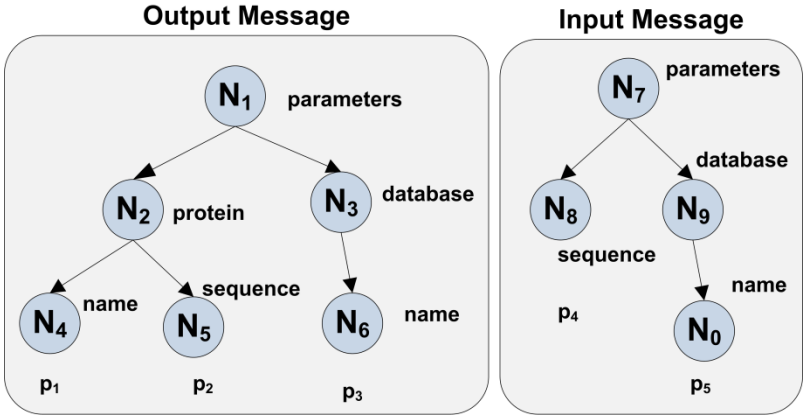

Figure 4.5. An example for path-based data mediation algorithm

CHAPTER 5

FORMAL SERVICE SPECIFICATION

In the Web service composition literature, a Web service model precisely and abstractly states the information concerning a Web service that can be used to compose a WSC process. The model presented here can be considered to be a refinement and a subset of the notion of semantic templates developed in our prior work on METEOR-S [51]. A semantic template is a placeholder for a Web service that gives names and semantic annotations for a service including the following aspects: interface/porttype, operations, input, output and fault types, as well as QoS specifications. These are formalized into a nested 3-tuple consisting of meta-data about the service, a collection of 7-tuples Semantic OPeration Templates (SOPTs) defining the operations of the Web service and a set of policy assertions. A SOPT contains the following specifications: (1) FunctionalConcept, (2) SemanticType-Input, (3) SemanticType-Output, (4) Precondition, (5) Effect, (6) SemanticFault and (7) OperationLevelPolicy. This dissertation work focuses on the first five specifications. We refine the model by drilling specifications for (2) and (3) down to a level of detail sufficient to allow more precise matching of inputs and outputs as well as to support a form of type safety. In the future, we plan to drill (1) down to the next level.

A Web service usually has one or more operations, so the formal specification of an operation is the basic building block of a Web service model. The input, output, precondition and effects (IOPE) are generally used to describe a Web service operation [11]. Moreover, the syntactic name of a Web service

operation may imply its functionality and the semantic annotation of a Web service operation specifies its functionality as well. Therefore, we formally model a Web service with *n* operations using formula 5-1. A Semantic Web Service (*SWS*) is defined as a set of semantically annotated operations (*SOP*) and *n* is the number of operations of the Web service. Each operation is an instance of *IOFPE*, which is the metadata for a Web service operation.

$$SWS = \{SOP_j \mid SOP_j : IOFPE, 1 \leq j \leq n\} \quad \text{(formula 5-1)}$$

$$IOFPE = \langle input, output, functionality, precondition, effects \rangle \quad \text{(formula 5-2)}$$

As shown in formula 5-2, *IOFPE* defines the metadata of an operation as a 5-tuple. The "input" and "output" describe the metadata of the inputs and outputs, respectively, which are first modeled as IODAGs in Section 4.2 and then are further specified as semantic structural types later in Chapter 4. The formal specification of "input" and "output" are used in data mediation and in the calculation of suggestion scores based on data mediation (see Chapter 4 for details). The functionality is defined as a tuple: *functionality* = <*name, annotation*>. The "*name*" is the syntactic name of the operation specified in the WSDL document and the "*annotation*" is the semantic annotation for the operation in the corresponding SAWSDL document. The formal specification of functionality is used to calculate the service suggestion score based on functionality (see Chapter 3 for details). The precondition and effects utilize a logic-based language to formally describe a Web service operation and maintain the current state for a WSC process.

5.1. Precondition and Effects

In formula 5-2, we model a Web service operation as a 5-tuple, IOFPE, where P and E stand for precondition and effects, respectively. In the Web service literature, preconditions are logical statements that are required to be satisfied (ensured to be true) prior to the execution of a Web service operation. Effects are logical statements that indicate what will be true upon the successful execution of a Web service operation. Preconditions and effects are not required to invoke a Web service operation, but they are often used in Web service composition or discovery.

We intend to use Horn logic (a subset of first-order logic) to specify preconditions and effects, which is more expressive than propositional logic. We were trying to use the Rule Interchange Format (RIF), which is a rule language, to specify preconditions and effects for our approach. Moreover, since RIF is a Prolog style language, it would make it easy to rewritten Prolog statements to RIF statements in the future. However, as a new language, to the best of our knowledge, there is no mature inference engine for RIF available. Therefore, we use Prolog instead, which has been out for decades and has many mature inference engines available.

We utilize WSDL-S to annotate the precondition and effects of a Web service operation. Preconditions and effects are added as extensible elements on an operation in a WSDL-S file (for a WSDL2 file). If the Web service is specified as a WSDL1.1 document, since the <*operation*> element is not extensible, the <*document*> element is used to annotate preconditions and effects. Figure 5.1 shows a partial WSDL-S file, which illustrates how precondition and effect annotations are added to a WSDL1.1 document. The operation "*getIds*" is one of the operations used in our scenario. The precondition and

effects are part of the <*document*> element of the <*operation*> element. The Prolog clauses are used as the statements of the preconditions and effects, which are the values of the *wssem:expression* attribute. The precondition "*hasWUBLASTjobid*(*X*)" indicates that the operation can be executed if there exists "*x*" that is a value for variable "*X*", and that "*hasWUBLASTjobid*(*x*)" is true (i.e., "$\exists x$ (*hasWUBLASTjobid*(*x*) $\wedge$ ($x \in X$))"). The effect "*assertz*(*hasWUBLASTHitIds*(*wuBLASThitid*))" indicates that after successfully executing the operation, a new fact is added to the end of the knowledge base, which is that "*hasWUBLASTHitIds*(*wuBLASThitid*)" is true. The effect "*assertz*(*isProtein*(*wuBLASThitid*) :- *hasDb*(*proteinDb*))" means that a new rule is add to the knowledge base and the rule expresses that if a protein database is used then the BLAST hit-IDs are protein sequence IDs.

```
<operation name="getIds" parameterOrder="jobid">
    <documentation>
      <wssem:precondition name="getIdsPre" wssem:expression= "hasWUBLASTjobid(X)."/>
      <wssem:effect name="getIdsEff" wssem:expression=
            "assertz(hasWUBLASTHitIds(wuBLASThitid)),assertz(isStringArray(wuBLASThitid)),
assertz(isProtein(wuBLASThitid) :- hasDb(proteinDb))."/>
    </documentation>
    <input name="getIdsRequest" message="tns:getIdsRequest" />
    <output name="getIdsResponse" message="tns:getIdsResponse" />
</operation>
```

Figure 5.1. Precondition and effects for operation "*getIds*"

Preconditions and effects are used to maintain the current state of the current WSC process. A Prolog knowledge base is used to express the current state, which can be queried for entailment and updated through a Prolog inference engine. As we mentioned before, a knowledge base is a collection of facts and rules. Figure 5.2 shows how to compute the current state. The knowledge base is initialized to the initial

state, which includes one or more facts or rules. For every operation (*OP*) in the current WSC process, if the current state (*st*) entails the precondition of the operation (*pre*(*OP*)), successful execution of the operation will update the current state with the effect of the operation (*effect*(*OP*)).

Algorithm to compute the current state for a WSC process

Input: st_0 : initial state of the process
process = { OP_1, OP_2, …, OP_n}: a WSC process that has a set of operations

```
begin
    st ← update(KB, st0)    //initialize KB with initial state, st is current state
    for OP in process
      if st ⊨ pre(OP)
          st ← update(KB, eff(OP))
      else
          error ("current state does not entail the precondition of OP ")
          break
      end if
    end for
end
```

Figure 5.2. Algorithm to compute current state for a process

Figure 5.3 shows a simplified view of a BPEL process. Focusing on the state variables, which are typically changed by "invoke" BPEL activities, if the candidate operation OP_x will be placed in between OP_1 and OP_2, the state st_1 has to entail pre (OP_x). Similarly, the application of the eff (OP_x) to state st_1 must entail *pre* (OP_2). st_0 is the initial state that represents the state after receiving inputs.

The preconditions/effects score S_{pe}, which is part of the calculation of the service suggestion score in Chapter 3, is calculated based on preconditions and effects. Formula 3-4 in chapter 3 presents the

calculation of the preconditions/effects score. If the current state (*st*) entails the precondition of the candidate operation *pre*(OP_x), the score will be one, otherwise the score will be zero.

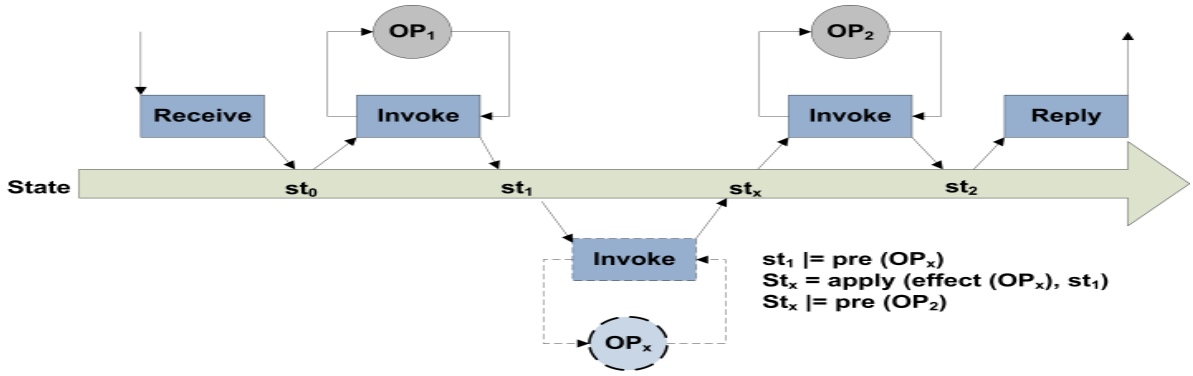

Figure 5.3. States in a BPEL process

In the automatic Web service composition literature, logic-based languages are used to specify preconditions and effects as well as to describe states of a WSC process. This allows planning algorithms to be utilized to build complete process specifications. We are doing a similar thing here, but for only a small portion of the overall design. It is well-known that the complexity trade-off is a challenge to deal with (i.e., low complexity logic leads to efficient reasoning, but limited expressivity). Considering the cost of utilizing a planner to generate a plan every time we calculate the suggestion score, we do not include a planner into our approach but leave the potential, which will be discussed in the future work chapter.

CHAPTER 6

SIMILARITY MEASURES

Two types of similarity measures are involved in our approach, i.e., syntactic similarity and semantic similarity. For data mediation, the syntactic similarity score between two XML tags is calculated by comparing the names of these two XML tags, while the semantic similarity score is achieved by comparing the semantic annotations on the two XML tags. Similarly, to calculate the functionality score, the syntactic similarity score used to calculate the functionality score is based on a comparison of the names of the Web service operation with the given keywords of the user desired functionality, while the semantic similarity score used to calculate the functionality score is determined by comparing the semantic annotation on the Web service operation with the given ontology concept of the user desired functionality.

6.1. Syntactic Similarity (*synSim*)

Syntactic similarity measures the similarity between two syntactic information objects, i.e., two text strings. In WSC syntactic similarity can be used to compare the names of two XML tags in the WSDL files. String metrics, a class of textual based metrics resulting in a similarity score between two text strings for approximate string matching or comparison, are used to measure this type of similarity.

A number of string metrics have been proposed, such as edit-distance metrics (also known as Levenshtein Distance) [52], token-based distance metrics (e.g., N-Gram (Q-Gram) [25]), fast heuristic string comparators, hybrid methods and so on. Moreover, some researchers, e.g., Cohen, et al. [53], Piskorski, et al. [54] and Lin [55] compared several string metrics based on their costs, performance, etc. Cohen [53], and Piskorski [54] reported good results for variants of the Jaro Metric for similarity of meaningful strings. Cohen also stated that the Jaro and Jaro-Winkler metrics seem to be intended primarily for short strings, and are close in average performance among the edit-distance based metrics and are also substantially more efficient than other edit-based metrics. Under our WSC circumstance, most strings to be compared would be short strings, such as the value of the name attribute of a tag defined in a WSDL document. Furthermore, since we are making suggestions to the user, time cost has also to be taken into consideration, in addition to accuracy. Chapman[36] presents a performance evaluation of the execution cost of string metrics for various size input. Based on all of those comparison results and our WSC circumstance, we choose the Jaro-Winkler metric [26] for our syntactic similarity measure between strings *s* and *t* using formula 6-1. It is faster than other edit-distance based metrics, but still performs well. It is primarily for short strings and is tested to have good results as a similarity metric for meaningful strings. In the future, we might explore or test additional string metrics that might fit better for WSC.

$$synSim(s,t) = Jaro\text{-}Winkler\ (s, t) \qquad \text{(formula 6-1)}$$

[36] SimMetrics: http://staffwww.dcs.shef.ac.uk/people/S.Chapman/stringmetrics.html

6.2. Semantic Similarity

Semantic similarity measures the similarity between semantic concepts based on an ontology. In our WSC approach, it is used to compare the semantic annotations in SAWSDL documents of Web services, which is important for both process mediation and data mediation for WSC. For instance, to calculate the functionality score for a service suggestion, we have to measure the semantic similarity between the annotations on the candidate Web service operation and the user desired Web service operation. In data mediation, a semantic similarity measure is used to calculate the score that results from comparing two semantic annotations in SAWSDL at the elemental level.

Semantic similarity measures have been studied by a number of researchers. Some of them, e.g. [56-59], calculate the similarity between two semantic concepts based on the distance in the ontology. Authors of [60, 61] measure the semantic similarity based on the subsumption relationship between semantic concepts. Properties of a concept are also considered by some researchers, such as Cardoso et al., [62] when calculating the similarity score between semantic concepts. Some other researchers combine several different algorithms together to calculate the semantic similarity between concepts. For example, Verma [22] developed a similarity measure used for his semantic Web service discovery algorithm, which is based on property matching and the subsumption relationship between concepts. Furthermore, Garlapati [23] compared several algorithms for Semantic Web service discovery including semantic similarity measures. According to his evaluation, he claims that Verma's [22] similarity measure for semantic concepts is a better matching technique and has a fairly performance since it measures the

concept similarity by taking a weighted average of the syntactic similarity, property similarity and coverage similarity between the two semantic concepts.

Therefore, we adopted his algorithm for our semantic similarity measure, and improved and customized it to fit our WSC approach. For instance, we change the *NGram* algorithm used in the syntactic similarity measure for concept similarity to the *Jaro-Winkler* algorithm, which has better accuracy and is still quite fast as we discussed in the Section 6.1. We will describe more details about the semantic similarity measure used in our approach in the rest of this section.

6.2.1. Concept Similarity (*conSim*)

Formula 6-2 computes the semantic similarity score (*conSim*(C_I,C_O)) between two concepts C_I and C_O from the same ontology, where C_O is the concept annotated on an output and C_I is the concept annotated on the input that will be fed by the output.

$$conSim(C_I, C_O) = w_1 \cdot conSynSim(C_I, C_O) + w_2 \cdot conPropSim(C_I, C_O) + w_3 \cdot conCvrgSim(C_I, C_O)$$

(formula 6-2)

where w_1, w_2, w_3 are the weights for the concept syntactic, property and coverage similarity, respectively. $w_1+w_2+w_3 = 1$, initially, they are set to be *1/3*. The concept similarity, concept syntactic, property and coverage similarity all range from 0 to 1.

In Verma's original calculation, there is another part, context similarity, which is taken into consideration for the concept similarity. We remove it because it is time-consuming since it requires building two more sets of concepts for every concept used in every application context. Our WSC approach

is to suggest Web services to the user, which require a shorter overall response time for the user and needs to take out the time consuming part.

6.2.1.1. Concept Syntactic Similarity (*conSynSim*)

Concept syntactic similarity (*conSynSim*) measures the syntactic similarity between two concepts, which compares the names and labels of the two concepts (Garlapati only compares labels). If no label is attached for any of the concepts in the ontology, only name comparison is used to compute concept syntactic similarity as shown in formula 6-3. C_I and C_O are the concepts to be compared.

$$conSynSim(C_I, C_O) = \begin{cases} w_4 \cdot synSim(C_I.name, C_O.name) + w_5 \cdot synSim(C_I.label, C_O.label), \\ \qquad\qquad C_I.label, C_O.label \neq \emptyset \\ synSim(C_I.name, C_O.name) \qquad\qquad otherwise \end{cases}$$

(formula 6-3)

$w_4 + w_5 = 1$, to make the final concept syntactic similarity score between 0 to1. Initially, both of the weights are set to 0.5, so the name and label have the same priority. Later on, these weights could be trained by a machine algorithm, after gathering enough experimental data. The similarities between the names and labels are calculated using the syntactic similarity algorithm described in Section 6.1, which both range from 0 to 1.

6.2.1.2. Concept Coverage Similarity (*conCvrgSim*)

Concept coverage indicates the extent of knowledge that the concept covers. In an ontology, a concept is more general than the concepts it subsumes, so it has higher knowledge coverage than its sub-concepts.

An ontology may be modeled as a DAG structure based on this subsumption relationships, so our concept coverage similarity measures the similarity based on the relative position of the two concepts in the DAG model of an ontology (i.e., the concept taxonomy or subsumption hierarchy). Subsumption relationships are determined by a reasoner, for efficiency we use a Jena built-in reasoner "OWL mini" (see implementation details in Chapter 7).

This work adopts the terminologies used for subsumption relationships between Web services in the OWLS-MX [24] and SAWSDL-MX [61] matchmakers and customizes them to classify different types of subsumption relationships of semantic concepts defined in an ontology for our similarity measures. We also extend and formalize Verma's concept coverage similarity.

Different types of subsumption relationships between two concepts in an ontology are discussed below. C_I and C_O are the concepts to be compared from the same ontology, the C_O plays an output role and C_I plays an input role, as shown in formula 6-2.

Exact match: $C_O \equiv C_I$

C_O has exactly the same knowledge coverage as C_I, so the concept coverage similarity score between them is the highest, i.e. 1.

Plug-in match: $C_O \in LSC(C_I)$

$LSC(C_I)$ is the set of least specified concepts of C_I, which are direct children of C_I in the ontology subsumption hierarchy, i.e., immediate sub-concepts of C_I, so C_O has smaller knowledge coverage than C_I.

Subsumed-by match: $C_O \sqsubseteq C_I$

C_O subsumed-by matches C_I indicates that C_O is more specific than C_I, and has smaller knowledge coverage than C_I. This match is a relaxation of plug-in match as shown in Figure 6.1.

Subsumes match: $C_O \sqsupseteq C_I$

C_O subsumes C_I means that C_O is more general than C_I and has wider knowledge coverage. However, in general, it is not type safe for a Web service operation to take a more general input than its requirement.

In addition to those subsumption relationships listed above, another relationship between two concepts in the ontology subsumption hierarchy is that the two concepts share one common ancestor. Figure 6.1 depicts the relationships between all different types of subsumption relationships discussed in this section. In general, an OWL ontology has "*Thing*" as the root concept of its subsumption hierarchy, so any two concepts in an ontology will share a common ancestor. Subsumes match and subsumed-by match are two more specific subsumption relationships than just sharing a common ancestor. Plug-in match is a special case of subsumed-by match. Exact match is the conjunction of subsumes match and subsumed-by match, which indicates: $C_O \equiv C_I \Leftrightarrow (C_O \sqsupseteq C_I) \wedge (C_O \sqsubseteq C_I)$.

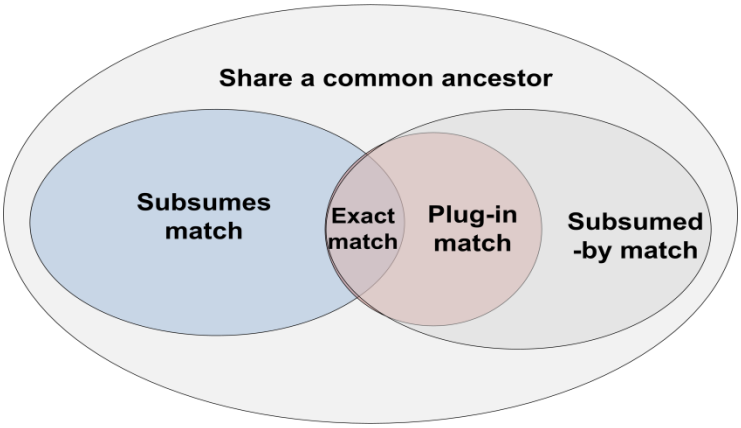

Figure 6.1. Different types of subsumption relationships: exact match is a special case of plug-in match and plug-in match is a special case of subsumed-by match

According to the different subsumption relationships presented above, we changed Verma's calculation of the concept coverage similarity (*conCvrgSim*) to formula 6-4. Case I is the exact match, $C_O \equiv C_I$, and $conCvrgSim(C_I,C_O) = 1$. Case II is the subsumed-by match excluding case I, $C_O \sqsubset C_I$. Since in this case the output is actually perfectly fine to be fed as the input of the succeeding operation, we set decay rate $\lambda_1 = 0.01$ such that its coverage similarity score will be very close to 1 shown as the case II in Figure 6.2. Case III is that C_O subsumes C_I excluding case I, $C_O \sqsupset C_I$. Since in this case the output is not type safe to be the input of the succeeding operation, $\lambda_2 = 1$ and its coverage similarity score decreases very quickly to 0, shown as case III in Figure 6.2. Case IV is that C_I and C_O share a common ancestor excluding case I, II and III. Their knowledge coverage may not overlap one another. This case experiences the fastest decay for the concept coverage similarity measure, $\lambda_3 = 2$ shown as case IV in Figure 6.2. Figure 6.2 depicts the decay rates of various cases of subsumption relationship between concepts in ontology. The decay rates are in increasing order $\lambda_1 = 0.01 < \lambda_2 = 1 < \lambda_3 = 2$ and the final *conCvrgSim* score ranges from 0 to 1.

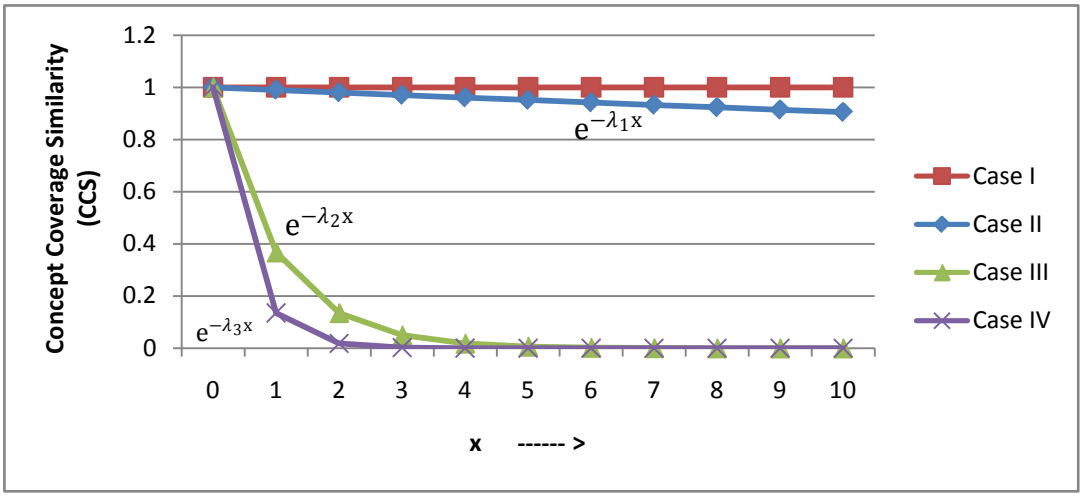

Figure 6.2. The decay graphs of concept coverage similarity of four cases

$$conCvrgSim(C_I, C_O) = \begin{cases} 1, & case\ I:\ C_O \equiv C_I \\ e^{-\lambda_1 x}, & case\ II:\ C_O \sqsubseteq C_I \\ e^{-\lambda_2 x}, & case\ III:\ C_O \sqsupseteq C_I \\ e^{-\lambda_3 x}, & case\ IV: \exists C_* \mid C_* \sqsupseteq C_O \wedge C_* \sqsupseteq C_I \\ & x = x_1 + x_2 \end{cases} \quad \text{(formula 6-4)}$$

where x is the distance between C_O and C_I via their common ancestor (C_*) in the subsumption hierarchy of the ontology, which is the sum of x_1 and x_2. x_1 is the specialization level between C_O and C_* in the subsumption hierarchy of the ontology. x_2 is the specialization level between C_I and C_* in the subsumption hierarchy of the ontology. Figure 6.3 gives some examples of the four cases in formula 6-4 for the two concepts C_O and C_I in the ontology subsumption hierarchy.

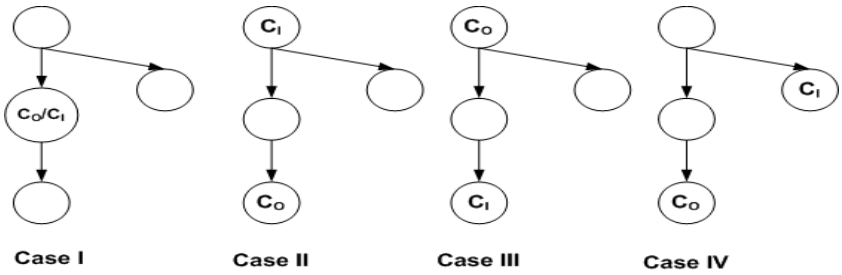

Figure 6.3. Examples of four cases in formula 6-4

6.2.1.3. Concept Property Similarity (*conPropSim*)

Given that concept C_I has a set of properties $PI = \{pi_1, pi_2, \ldots, pi_m\}$ and concept C_O has a set of properties $PO = \{po_1, po_2, \ldots, po_n\}$, concept property similarity ($conPropSim\ (C_I, C_O)$) is the score of the optimal match between the two sets. Therefore, this optimal matching problem can be modeled as a weighted bipartite graph problem.

Garlapati [23] also tackles this problem as a weighted bipartite problem and utilizes the Hungarian algorithm to solve it. We adopt and customize his approach. As shown in formula 6-5, *conPropSim* is calculated based on the optimal matching score between the two sets of properties. Further, if the number of properties of C_I is more than that of C_O, there will be some unmatched properties of C_I. The final *conPropSim* score will have a penalty for those unmatched properties, which will be e to the -0.05 times the number of unmatched properties.

$$conPropSim(C_I, C_O) = max\left\{\frac{1}{L}\sum_{j=1}^{L} propSim\left(pi_j, po_j\right)\right\} \cdot e^{-0.05 \cdot N_{unmatch}}$$

(formula 6-5)

$L = min\{m, n\}$, m and n are the size of the properties of concept C_I and C_O respectively. $pi_j \in PI$ is a property of C_I. $po_j \in PO$ is a property of C_O, which is the optimal match of pi_j making the maximum matching score between *PI* and *PO*. $propSim(pi_j, po_j)$ is the property similarity between the two properties pi_j and po_j, which is calculated by formula 6-6. $N_{unmatch}$ is the number of unmatched properties of *PI*

$$N_{unmatch} = \begin{cases} |m-n|, & m \neq n \\ 0 & otherwise \end{cases}$$

6.2.1.3.1. Property Similarity (*propSim*)

Property similarity (*propSim* (*pi*, *po*)) measures similarity between two individual properties $pi \in PI$ and $po \in PO$ as described in Formula 6-6.

$$propSim(pi, po) = w_6 \cdot \sqrt[3]{(propRangeSim(pi, po) \cdot propCardSim(pi, po) \cdot propSynSim(pi, po))}$$

(formula 6-6)

where w_6 is a weight affected by an inverse functional property. If an inverse functional property is compared with a non-inverse functional property, there will be a penalty for the similarity between the two properties as shown below.

$$w_6 = \begin{cases} 0.8 & if\ pi\ is\ an\ inverse\ functional\ property\ and\ po\ is\ not \\ 1 & otherwise \end{cases} \quad \text{(formula 6-7)}$$

1) *propRangeSim*(*pi*, *po*) measures the similarity between the two properties *pi* and *po* based on their range similarity. If both ranges (R_I, R_O) are described as primitive types, the type compatibility decides their range similarity score (1 compatible or 0 incompatible). Please see chapter 4 for the details about the XSD primitive type compatibility. If both ranges are presented as concepts, the shallow concept similarity ($conSim_{shallow}(R_I, R_O)$) will give their range similarity as shown in formula 6-8. Their range similarity will be zero when one range is presented as a concept and the other is a primitive type.

$$propRangeSim(pi, po)$$

$$= \begin{cases} 1 & ranges\ of\ pi\ \ and\ po\ are\ compatible\ \ primitive\ type \\ 0 & ranges\ of\ pi\ \ and\ po\ are\ incompatible\ primitive\ type \\ conSim_{shallow} & both\ ranges\ of\ pi\ \ and\ po\ are\ concepts \end{cases}$$

$$conSim_{shallow}(R_I, R_O)$$

$$= w_7 \cdot conSynSim(R_I, R_O) + w_8 \cdot conCvrgSim(R_I, R_O) + w_9 \cdot conPropSim_{shallow}(R_I, R_O)$$

(formula 6-8)

$w_7 = w_8 = w_9 = 1/3$ are the weights for the three parts.

$$conPropSim_{shallow} = \frac{|P_{RO} \cap P_{RI}|}{|P_{RI}|} \quad \text{(formula 6-9)}$$

P_{RO} is the set of properties of R_O and P_{RI} is the set of properties of R_I. Formula 6-9 shows the calculation of shallow concept property similarity ($conPropSim_{shallow}$) is the fraction of the number of common properties between R_I and R_O by the number of properties of R_I.

2) *propCardSim* (*pi*, *po*) measures the cardinality similarity between two properties using formula 6-10. The cardinality of a property defines the number of values required by the property.

$$propCardSim\ (pi, po) = \begin{cases} 1 & CP_i = CP_o \text{ or } pi \text{ and } po \text{ both are functional properties} \\ 0.9 & CP_i < CP_o \\ 0.7 & CP_i > CP_o \end{cases}$$

(formula 6-10)

CP_i is the cardinality of property *pi* and CP_o is the cardinality of property *po*.

3) *propSynSim* (*pi*, *po*) is the syntactic similarity between two properties. The two weights $w_{10} = w_{11} = 0.5$. Both name and description of the properties *pi* and *po* are compared using the syntactic similarity measure as described in formula 6-11.

$$propSynSim\ (pi, po) = \begin{cases} w_{10} \cdot synSim(pi.name, po.name) + w_{11} \cdot synSim(pi.descr, po.descr), & pi.descr, po.descr \neq \emptyset \\ synSim(pi.name, po.name) & otherwise \end{cases}$$

(formula 6-11)

CHAPTER 7

ARCHITECTURE AND IMPLEMENTATION

Implemented using the Java language our system works as an external engine of a WSC designer to aid users composing Web services into a process (see section 10.2 for how to attach our suggestion engine to a designer). Candidate Web service operations are ranked and top-k services are suggested to the user during the process of Web service composition. We also implemented a data mediation engine to handle the connections between service operations (how to feed the output of a service operation to the succeeding service operation). Our data mediation engine can work as a modular subsystem to support the service suggestion engine or a standalone system invoked by another WSC system or designer to resolve data mediation issues. The system architecture and the details of each module will be presented in the rest of this chapter.

7.1. System Architecture

This system has two major functional components: a service suggestion engine and a data mediation engine, as shown in Figure 7.1. Several utility components, i.e., parsers, similarity measures, etc., support the service suggestion and data mediation engines. The service suggestion engine also invokes the data mediation engine as one part of its ranking algorithm. The data mediation engine can also run independently, without the service suggestion engine.

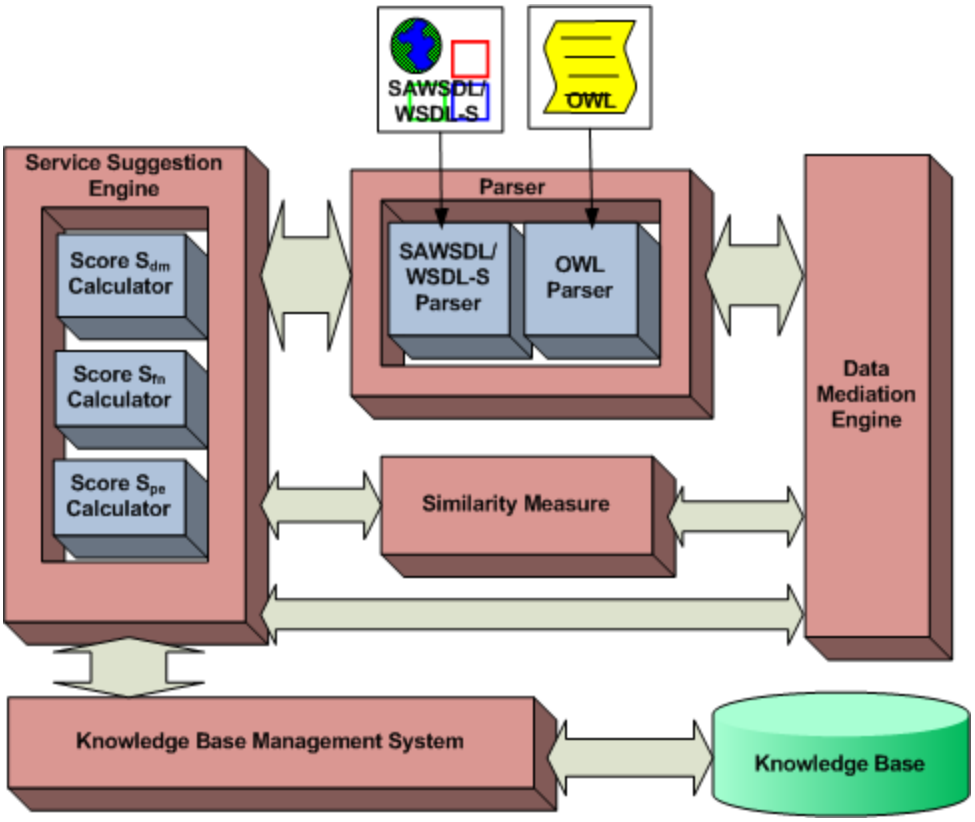

Figure 7.1. System architecture

7.2. System Components

The system contains five major components, namely, a service suggestion engine, a data mediation engine, similarity measures, parsers and a knowledge base management system (KBMS).

7.2.1. Service Suggestion Engine

The service suggestion engine is a major functional components that ranks and suggests Web service operations to the user for WSC. It requires parsers to parse the related SAWSDL/WSDL-S and OWL files

for the information used by all the ranking calculation modules. All the services available to the user are described by the SAWSDL/WSDL-S and WSDL files, which can be pre-added to a GUI designer that the user is using or discovered from some Web service registry/databases, e.g., BioCatalogue (http://www.biocatalogue.org/) or EMBRACE Service Registry (http://www.embraceregistry.net/). The service suggestion engine has three ranking calculation modules, as shown in Figure 7.1, i.e., S_{dm}, S_{fn} and S_{pe} score calculators. The S_{dm} score calculator invokes the data mediation engine and parser to calculate one part of the ranking score that is based on the data mediation algorithm discussed in Chapter 4. The S_{fn} score calculator invokes the similarity measure module and parser to compute part of the ranking score that is based on functionality. The S_{pe} score calculator invokes the KBMS and parser to calculate part of the ranking score that is based on the preconditions/effects specification discussed in Chapter 5.

7.2.2. Data Mediation Engine

Three data mediation algorithms are implemented: leaf-based, structure-based and path-based, as described in Chapter 4. The leaf-based data mediation and structure-based data mediation are compared with path-based data mediation algorithm, in the evaluation chapter (chapter 8).

The path-based data mediation engine consists of three components: the path generator, elemental matcher and structural matcher. The path generator invokes the parser to generate paths for the given SAWSDL/WSDL-S documents. Two extra Java libraries, Java Document Object Model (JDOM)[37] and

[37] Java Document Object Model (JDOM): http://www.jdom.org/

Jaxen[38] help to implement the path generator. The elemental matcher compares elements of two paths and returns the matching score. It invokes our similarity measure module to compare the syntactic and semantic information of the two elements. The JDOM library is used to extract information from an element for the implementation. The structural matcher performs structure-level matching; it finds matches for the given inputs and computes the matching score for every input.

The structure-based data mediation engine reuses the elemental matcher of the path-based mediation engine for elemental matching, and also uses three additional components: a DAG generator, a Depth First Search (DFS) traverser, and a homeomorphism detector. The DAG generator uses parsers to get the information of input/output and generate a DAG. The DFS traverser goes through the given DAG using a DFS algorithm to generate a list of nodes, which are represented as instances of the JDOM Element type. The homeomorphism detector invokes the elemental matcher and calculates the structure-level matching score with two given lists of nodes for the output and input to be mediated based on the sub-graph homeomorphism algorithm discussed in Section 4.4.

The leaf-based data mediation engine has a leaf-list generator that uses parsers to get the information of input/output and generate a list of leaves of its IODAG, which are represented as instances of the JDOM Element type. The structural matcher of the leaf-based data mediation engine reuses the elemental matcher of the path-based data mediation engine and finds the best matches between the two given lists of leaves as well as the best matching score for each leaf of the input leaf-list.

[38] Jaxen: http://jaxen.org/

7.2.3. Knowledge Base Management System (KBMS)

The knowledge base (KB) is a collection of Prolog facts and rules, which represents the states of a WSC process. The KBMS provides an isEntail() method to query the entailment of the KB and updateState() method to update the KB to reflect the updated state. The Prolog engine SWI-Prolog[39] and JPL[40] Java library are used to implement the KBMS. WSDL-S is used to provide the preconditions and effects annotations for the Web service operations. The KBMS requires the parser component to retrieve the actual logical statements of preconditions and effects in the WSDL-S document.

7.2.4. Parsers

This component includes two individual parsers, which provide all the required information to other components. The first is the SAWSDL/WSDL-S parser and another is the OWL parser. The SAWSDL/WSDL-S parser uses the JDOM and Jaxen libraries. Currently, the Jena library is used to implement the OWL parser. In the future we might switch to the OWL API for less execution time cost.

7.2.5. Similarity Measures

The similarity measures component has two modules: syntactic similarity and semantic similarity measure modules. The secondString library[41] is used to implement the syntactic similarity measure module. The secondString library has a number of string metrics implementations. As we discussed in Section 6.1, we choose to use the Jaro-Winkler string metric for this implementation. We implement the

[39] SWI-Prolog: http://www.swi-prolog.org/
[40] JPL: http://www.swi-prolog.org/packages/jpl/java_api/index.html
[41] SecondString: http://secondstring.sourceforge.net/

syntactic similarity measure module using the abstract factory design pattern, so that it is easy to switch to a different string metric.

The semantic similarity measure module is implemented based on Garlapati's [23] similarity measure implementation. We customized and extended his implementation as discussed in Section 6.2. This module invokes our OWL parser to parse the ontology. It has three sub-modules: a concept syntactic similarity (*conSynSim*) module, a concept coverage similarity (*conCvrgSim*) module and a concept property similarity (*conPropSim*) module. The *conSynSim* module utilizes the syntactic similarity measure module to compute the syntactic similarity between two semantic concepts. An implementation of the weighted Hungarian algorithm is used for the *conPropSim* module.

CHAPTER 8

EVALUATION

Three evaluations are performed and presented in this chapter. The first evaluation compares the three data mediation algorithms presented in this dissertation regarding how effectively they deal with various structures of input and output. The second evaluation studies the effectiveness of various semantic annotations used in our WSC approach. In particular, it tests whether incomplete annotations are sufficient for effective service suggestions. The third evaluation compares the accuracy of the different suggestion algorithms proposed in this dissertation.

8.1. Evaluation Settings

This evaluation is performed on a laptop: ThinkPad T61, processor: Intel Core 2 Duo CPU T9300 @2.50GHz, RAM: 4GB, System type: 64-bit windows Vista. Programs run inside Eclipse 3.5 with JRE 1.6_19. The Prolog engine SWI-Prolog 5.10.0 is installed in the system and invoked through the JPL API.

Some Web services from EBI are used for this evaluation, as listed in Table 2.1 in Chapter 2. The bioinformatics process discussed in Chapter 2 is used as the user desired WSC process. Suggestions are requested for the six steps of the WSC process, except that the first that must be chosen by the designer. (Note, for bi-directional suggestions, there will be five requests.) For each request, the fifty candidate service operations have been ranked by three expert human evaluators (see Appendix B for three ranked

lists of the fifty candidate Web service operations). Two ontologies, EMBRACE Data and Methods (EDAM)[42] and Ontology for Biomedical Investigations (OBI)[43] are used/enriched for annotating the Web services. The SAWSDL/WSDL-S and OWL files used in this evaluation as well as the source code of our implementation can all be downloaded from the following Website: http://cs.uga.edu/~guttula/Galaxy/wsdls.html.

8.2. Evaluation Measure

Since designing a Web service process is complex, it is difficult to determine precisely if a service is correct. Therefore, we simply compare how well our suggestion algorithms compare to the human evaluators in terms of the "degree of overlap" of the *top-k* suggestions as defined in formula 8-1. Degree of overlap measures the probability that a service operation suggested within the *top-k* by the program is also suggested by the evaluators, which is the number in the intersection of the *top-k* services suggested by program and *top-k* consensus of evaluators divided by *k*. We calculate the degree of overlap for top-5, top-10, top-15 and top-20 suggested services, respectively for each suggestion request. We do this by comparing each of the rankings calculated by our system with the consensus that is the average of the three expert human evaluators' rankings.

$$Degree\ of\ Overlap = \frac{|top-k_{algorithm} \cap top-k_{concensus}|}{k}$$

(formula 8-1)

where k = 5, 10, 15 or 20.

[42] EMBRACE Data and Methods (EDAM): http://sourceforge.net/projects/edamontology/
[43] Ontology for Biomedical Investigations (OBI): http://purl.obolibrary.org/obo/obi

8.3. Evaluation I

Three data mediation algorithms have been proposed in this dissertation. The main difference between the three algorithms is how much they consider the structure of the input/output of a Web service operation during data mediation. Therefore, this evaluation aims to test the performance of the three algorithms to assess how much the structure of the input/output affects the performance of the data mediation algorithms in terms of degree of overlap.

During the evaluation, three data mediation algorithms are used to suggest services using the forward suggestion algorithm for each request during the composition of the process described in Chapter 2. The degrees of overlap for the top-5, top-10, top-15 and top-20 suggestions for each request are calculated for the three data mediation algorithms, as well as for the three human rankings and a random ranking. To maximize the effectiveness of the data mediation algorithms, we fully annotate the inputs and outputs of all the candidate services. To further improve results, the functionality annotations are also utilized.

8.3.1. Hypotheses

Considering the structure of the input/output of a Web service operation, the leaf-based data mediation only considers the leaf nodes, so it should have the lowest degree of overlap. The structure-based data mediation utilizes a sub-graph homeomorphism algorithm and considers the whole structure of the input/output. Therefore, it should have a higher degree of overlap compared to leaf-based data mediation algorithm, but being too strict on structure may rule out some correct suggestions. The path-based data mediation decomposes the input/output into many simple paths rather than the sub-trees

used in the structure-based data mediation, so the path-based data mediation still takes the structure of input/output into consideration, but in a less strict way compared to structure-based data mediation (see details and examples in section 4.5). Therefore, the path-based data mediation should have the highest degree of overlap among all three algorithms.

8.3.2. Results

Figures 8.1 to 8.6 depict the degree of overlap comparison of the three data mediation algorithms for each request. Three human evaluators' rankings and a random ranking are also presented as comparison. Figure 8.7 gives the average degree of overlap of all six requests for three data mediation algorithms as well as three evaluators' rankings and the random ranking. (Please see the appendix C for all the statistical data for this evaluation.)

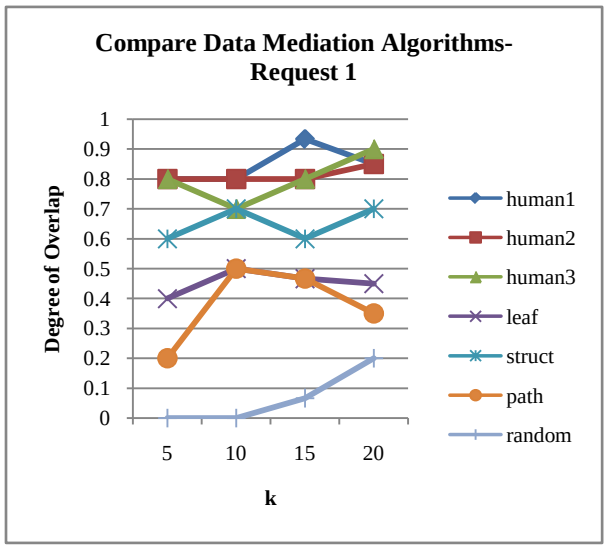

Figure 8.1. Degree of overlap of three data mediation algorithms for request 1 (selected operation: "getIds")

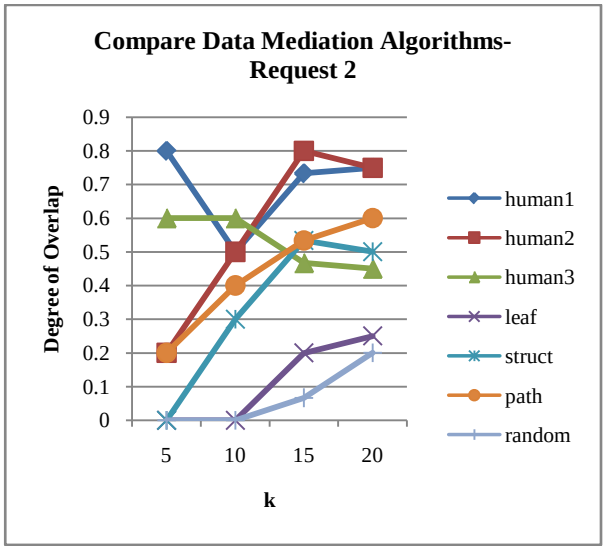

Figure 8.2. Degree of overlap of three data mediation algorithms for request 2 (selected operation: "array2string")

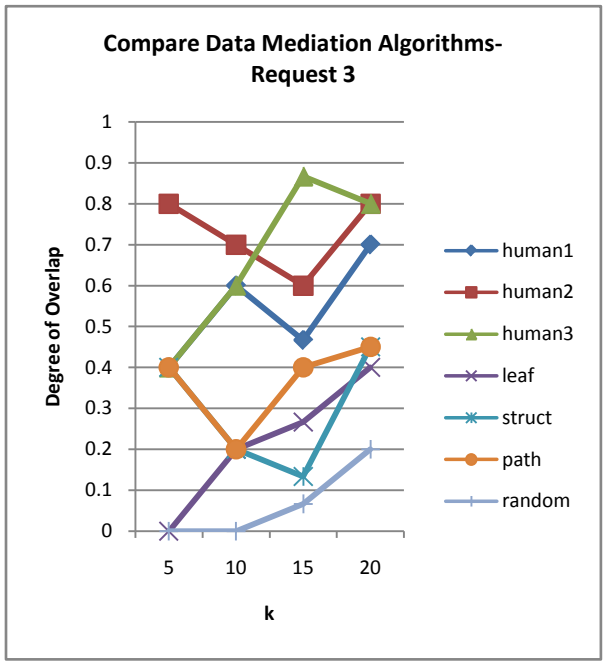

Figure 8.3. Degree of overlap of three data mediation algorithms for request 3 (selected operation: "fetchBatch")

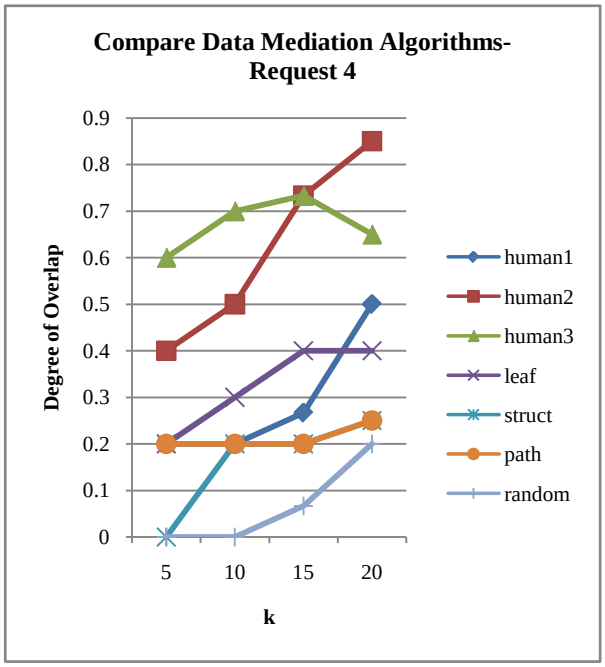

Figure 8.4. Degree of overlap of three data mediation algorithms for request 4 (selected operation: "run")

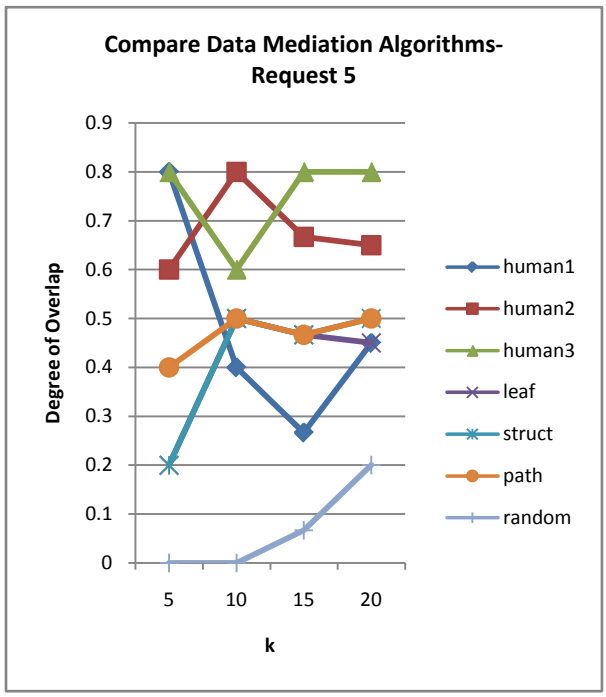

Figure 8.5. Degree of overlap of three data mediation algorithms for request 5 (selected operation: "getResult")

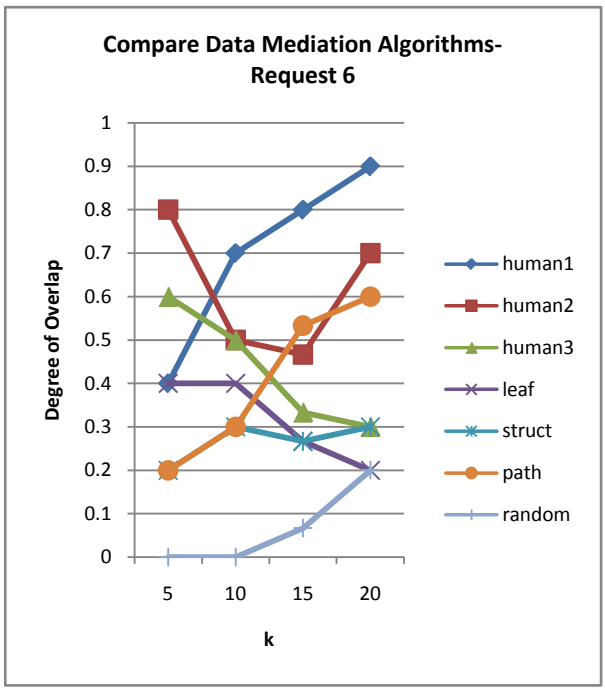

Figure 8.6. Degree of overlap of three data mediation algorithms for request 6 (selected operation: "base64toString")

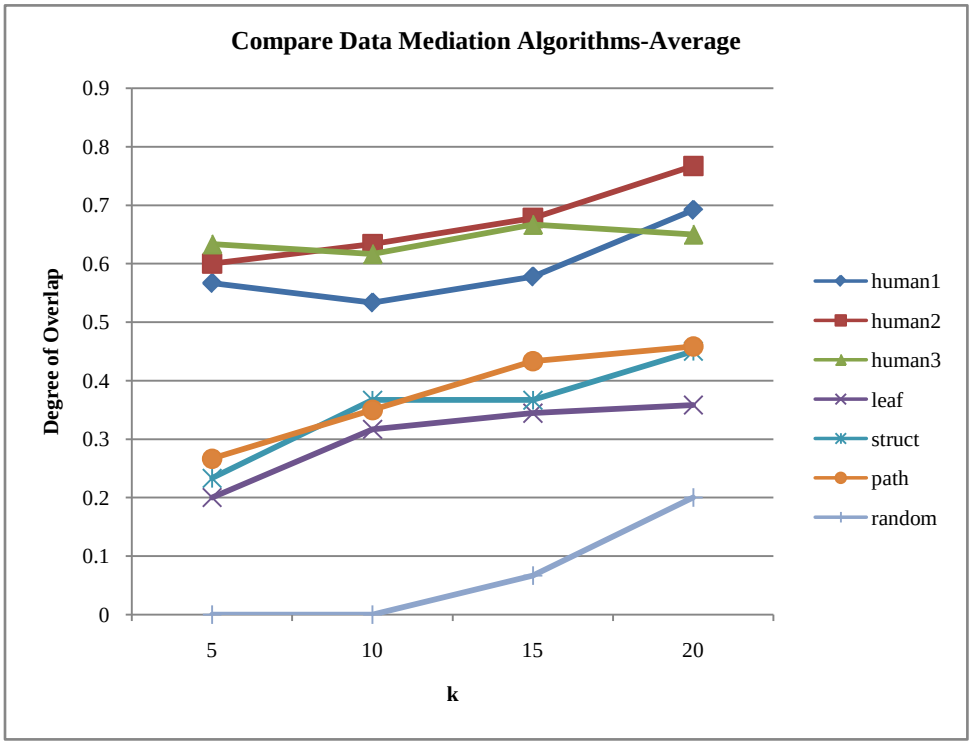

Figure 8.7. Average degree of overlap of all six requests for three data mediation algorithms

8.3.3 Findings

As shown in Figures 8.1-8.7, the evaluation results indicate that the path-based data mediation algorithm has a higher degree of overlap than the other two algorithms and the leaf-based data mediation has the lowest degree of overlap, which demonstrates that the structure of input/output does affect the performance. The structure-based data mediation algorithm is more complicated and may rule out some useful compositions, thereby resulting in lower performance, although it is still substantially better than the leaf-based data mediation algorithm.

Some of the Figures show poor results, e.g., Figure 8.4, in which the degree of overlap of top-5 and top-10 are very low. The reason may be that this request is looking for a service to perform multiple

sequence alignment on some sequences. The IODAG of the output of operation fetchBatch is a simple tree with one path consisting of four nodes: fetchBatchResponse – parameters – fetchBatchResponse – fetchBatchReturn. The structure of the input of operation run (ClustalW) is a complex tree with twenty six leaves, but only two of them are the required inputs. Among all the candidate operations, fifteen of them have an input IODAG that exactly match this simple path tree. Therefore, these fifteen operations result in much higher data mediation scores compared to the operation run (ClustalW). The leaf-based algorithm has a higher performance compared to other two algorithms in this case, but is still a poor result because of all the twenty four operational inputs of the operation run (ClustalW). The solution might be adjusting the weights of internal nodes in the IODAG or the weights of the three sub-scores (S_{dm}, S_{fn}, S_{pe}), but more importantly, we need test Web services with various structures in their input/output IODAGs other than what we have here (e.g., one third of our candidate services have input IODAGs that are single path trees).

8.4. Evaluation II

The WSC approach proposed in this dissertation can utilize various types and levels of semantic annotations and work based on the amount of semantics that Web service providers and human process designers specified. Generally, less complex semantics leads to lower costs, but limited expressivity. Therefore, the complexity tradeoff of the semantics is a challenge to deal with. This evaluation aims to study the effectiveness of different types/levels of semantic annotations used in WSC, i.e., how much the semantic complexity will affect the performance of WSC approach.

For each candidate Web service operation, twelve different annotation cases are prepared for testing using the forward suggestion with path-based data mediation algorithms. Suggestions are requested by a user to aid the composition of Web services for each step of our scenario, except the first, which must be chosen by the designer. The quality of the suggestions are measured by degree of overlap for the top-5, 10, 15 and 20 suggestions made for each request using twelve different annotation cases. As shown in Table 8.1, the twelve different cases of annotations indicate different combinations of information used to calculate S_{dm} (semantics and syntax), S_{fn} and S_{pe} for formula 3-1 in Chapter 3. The information used for S_{dm} includes the syntactic name and semantic annotations for all the input and output of a Web service operation, such as <*message*>, <*part*> and <*element*>. The semantic annotations for S_{fn} are the functionality annotations for the Web service operations. The annotations for S_{pe} consist of preconditions, effects and an initial state. From the Table 8.1, we can tell that case 0 is the least complex, because it includes nothing, and the ranking based on it is effectively a random ranking. Case 11 is the fully annotated case, which uses all the annotations.

Table 8.1. Different cases of annotations

Information / Case	**Syntax used for S_{dm}**	**Semantic annotations for S_{dm}**	**Semantic annotations for S_{fn}**	**Semantic annotations for S_{pe}**
Case 0	No	No	No	No
Case 1	Yes	No	No	No
Case 2	Yes	Yes	No	No
Case 3	No	No	Yes	No
Case 4	Yes	No	Yes	No
Case 5	Yes	Yes	Yes	No
Case 6	No	No	No	Yes
Case 7	Yes	No	No	Yes

Information / Case	Syntax used for S_{dm}	Semantic annotations for S_{dm}	Semantic annotations for S_{fn}	Semantic annotations for S_{pe}
Case 8	Yes	Yes	No	Yes
Case 9	No	No	Yes	Yes
Case 10	Yes	No	Yes	Yes
Case 11	Yes	Yes	Yes	Yes

8.4.1. Hypotheses

Case 0 includes no information, so the ranking based on it is effectively a random ranking, and should have the worst performance. Better suggestions, in terms of degree of overlap, should be obtained by using fully complete annotations compared to fewer annotations. However, making fully complete semantic annotations is much harder and more time-consuming than the simple annotation cases, so some cases, i.e., case 4 and case 5, that are not too difficult to create and still have sufficient degree of overlap are more practical.

8.4.2. Results

Figures 8.8 to 8.14 show the comparisons of the twelve annotation cases used in six service suggestions in terms of degree of overlap, the left part of each figure shows annotation cases 0-5 (without precondition/effects) and the right part depicts annotation cases 6-11 (with precondition/effects). Three human evaluators' rankings are also presented for comparison. Figure 8.15 gives the average degree of overlap of the six requests using twelve annotation cases as well as three human evaluators and a random ranking. (Please see appendix C for all the statistical data for this evaluation.)

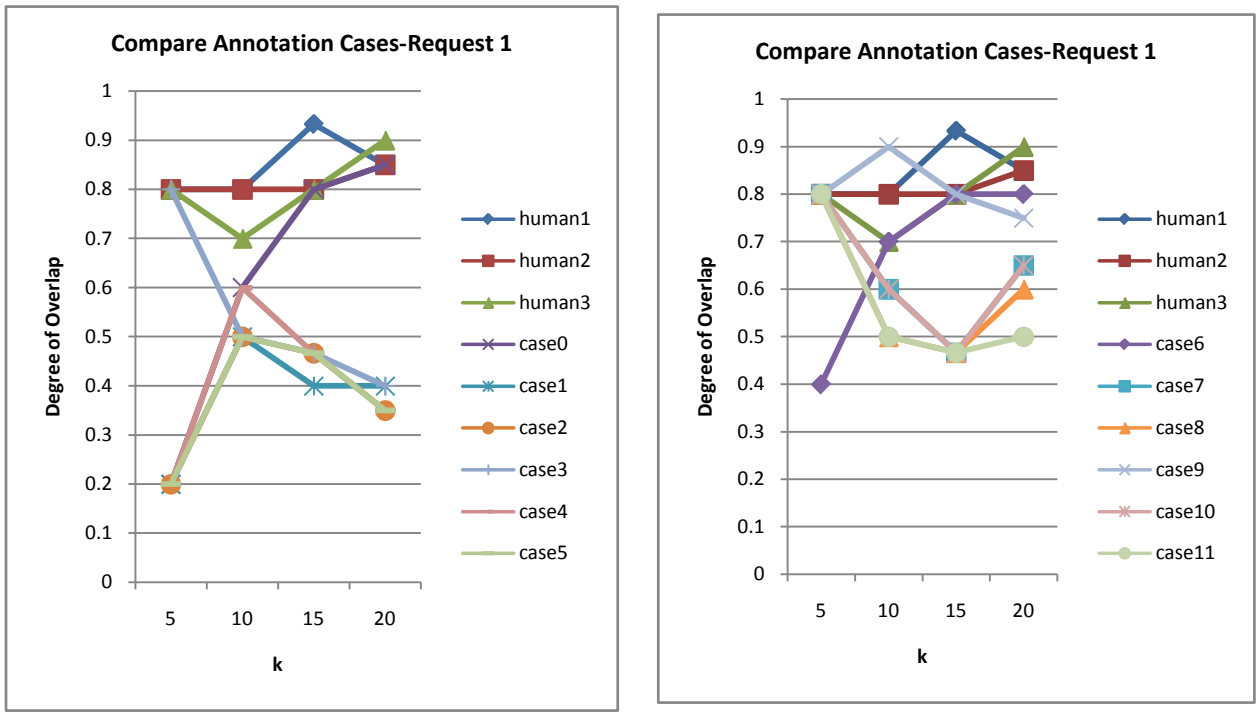

Figure 8.8. Compare different annotation cases - request 1 (selected operation: "getIds")

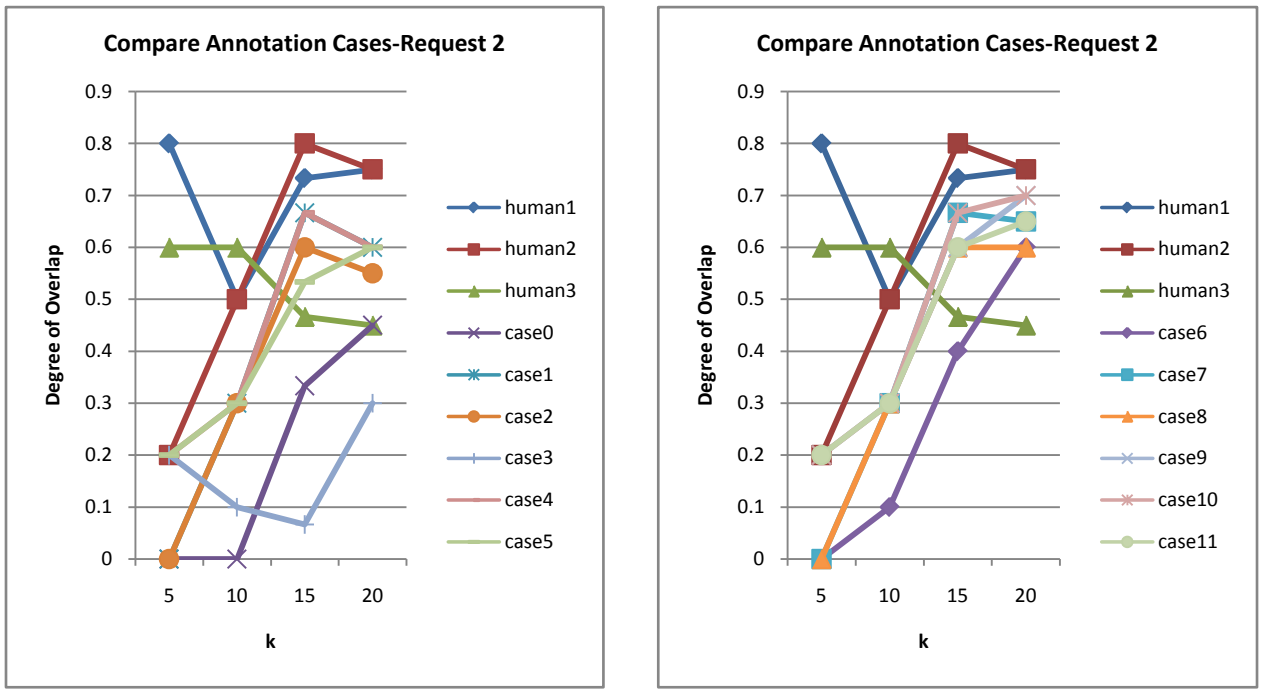

Figure 8.9. Compare different annotation cases - request 2 (selected operation: "array2string")

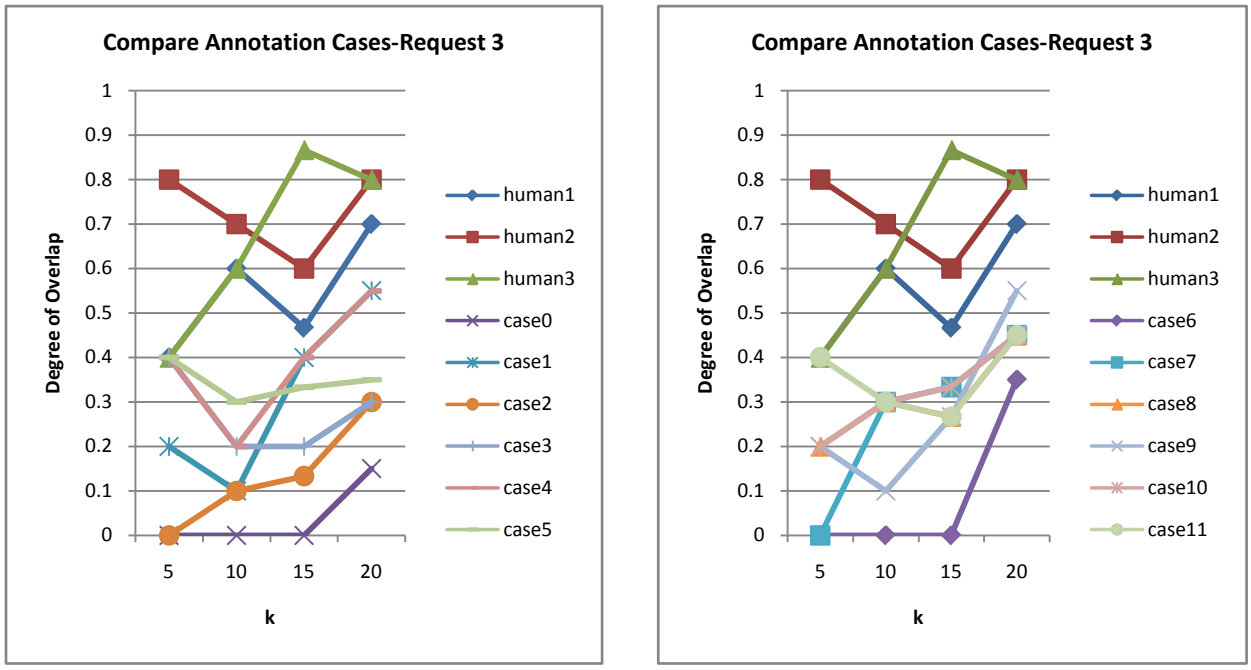

Figure 8.10. Compare different annotation cases - request 3 (selected operation: "fetchBatch")

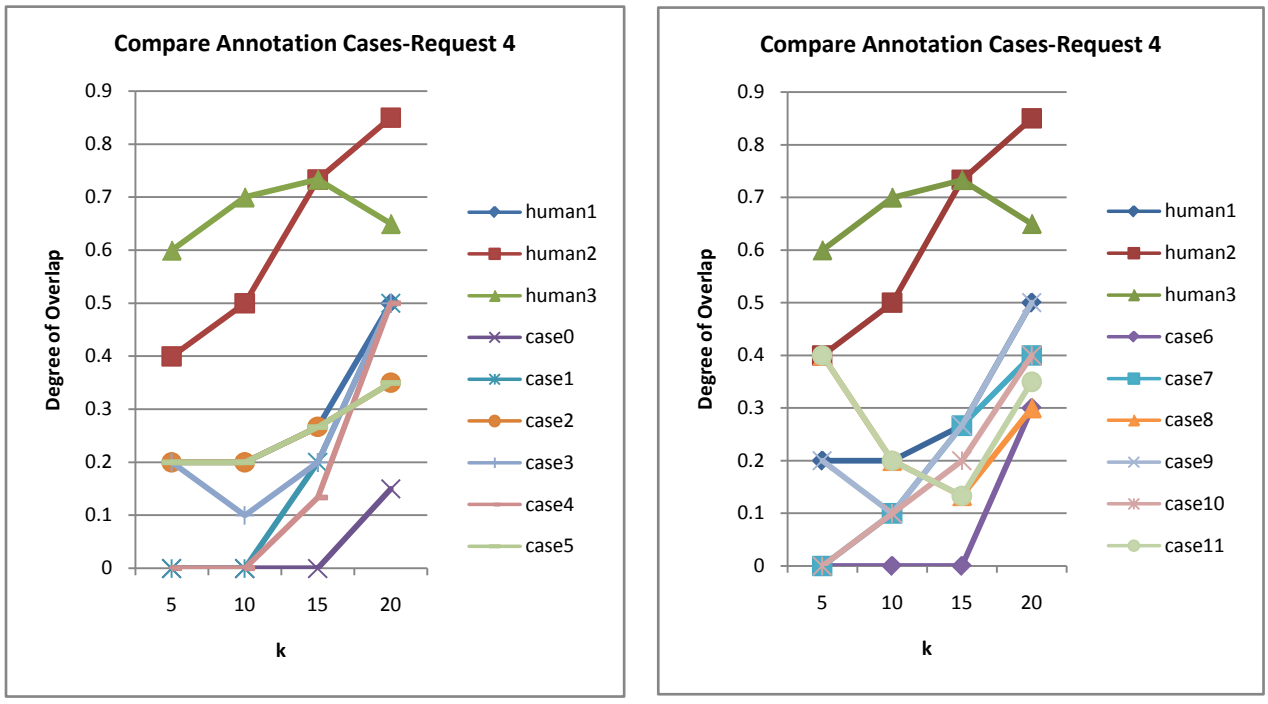

Figure 8.11. Compare different annotation cases - request 4 (selected operation: "run")

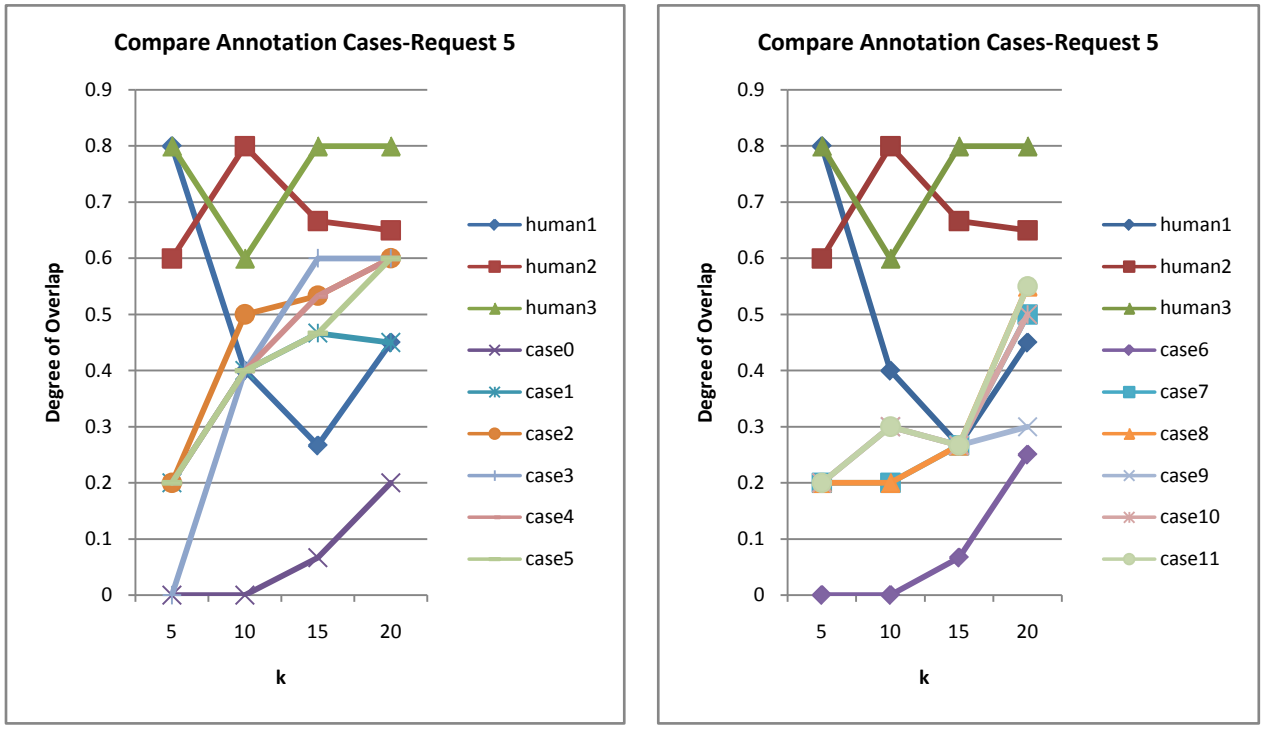

Figure 8.12. Compare different annotation cases - request 5 (selected operation: "getResult")

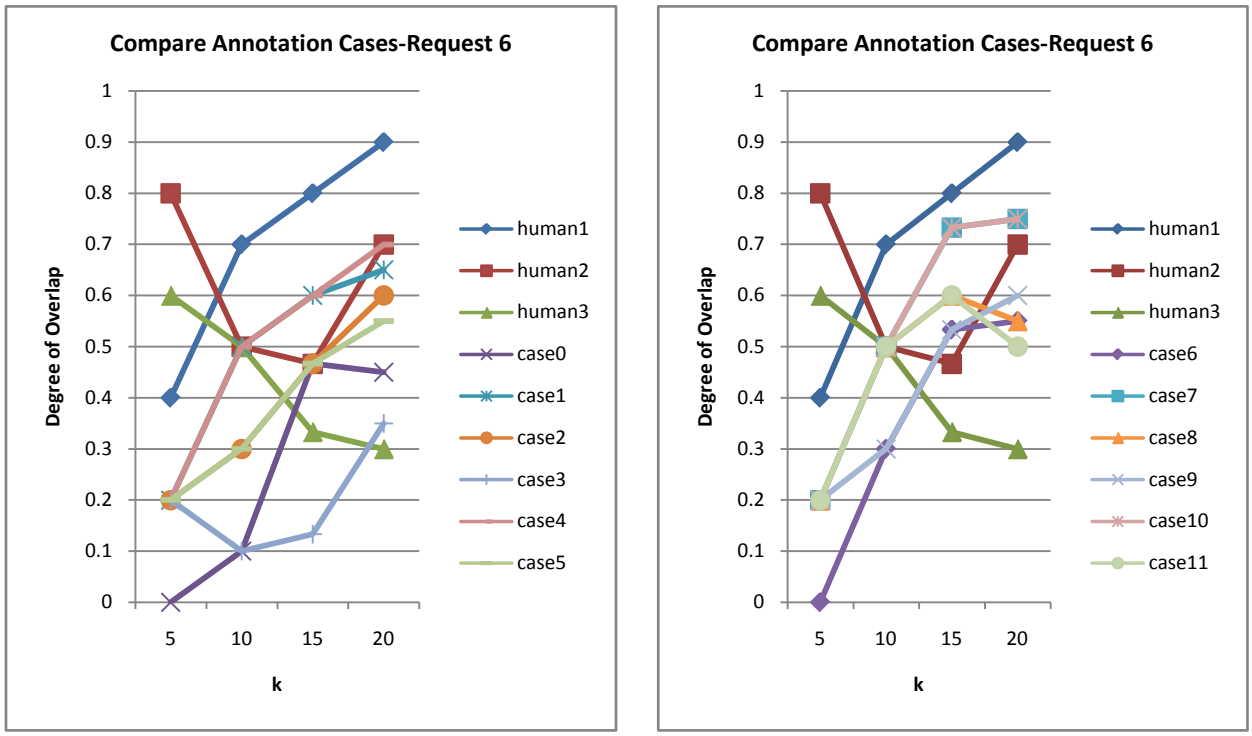

Figure 8.13. Compare different annotation cases - request 6 (selected operation: "base64toString")

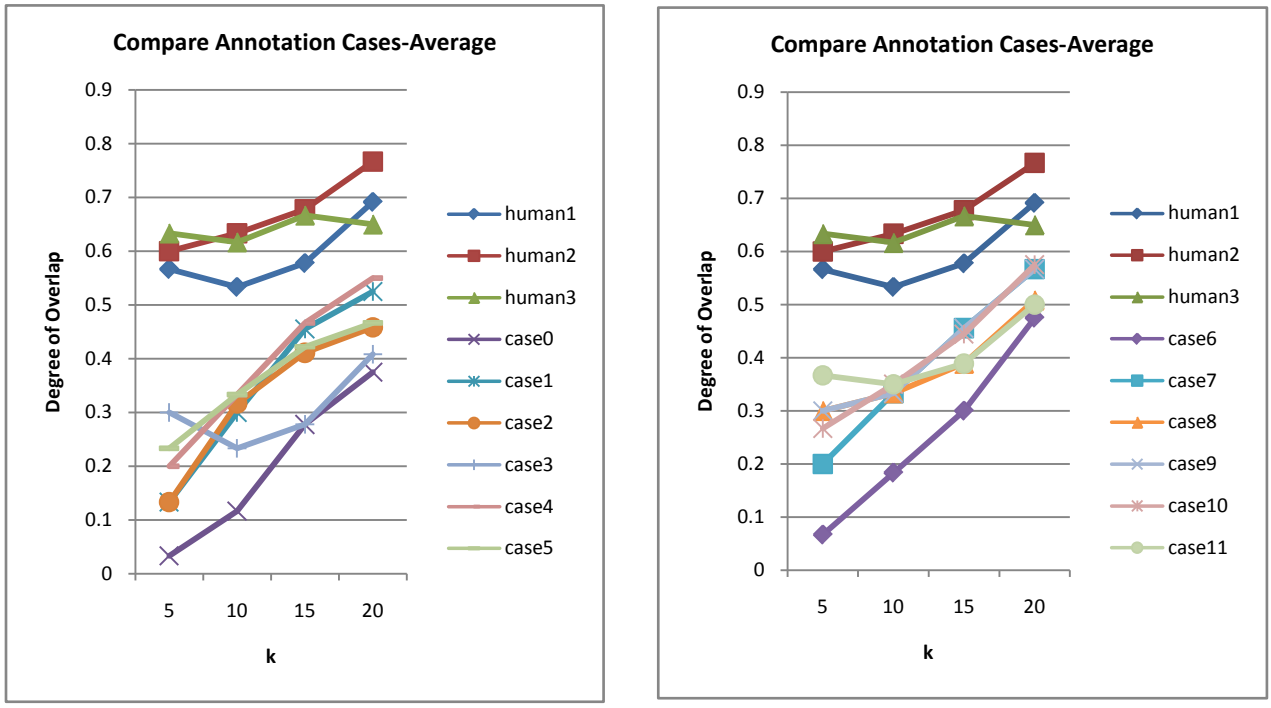

Figure 8.14. Comparing different annotation cases - average

8.4.3. Findings

As expected, case 0 has the lowest degree of overlap. Overall, complicated annotations result in higher degrees of overlap, e.g., averaging the top-5 and top-10, the best cases in order are 11, 9, 8, 10 and 5 (see Table C.14 in Appendix C). However, of these, the precondition/effects that require writing formal descriptions in a logic language can be difficult to specify. Therefore, some users may want to avoid such complicated annotations, but still keep sufficient performance. In this experiment, cases 2, 3 and 6 represent the pure annotations on input/output, functionality and precondition/effects, respectively. The results in Figure 8.14 indicate that the relative contributions towards suggesting correct Web services are ordered as follows: functionality annotations (case 3) contribute the most, followed by input/output

annotations (case 2) and finally precondition/effects annotations (case 6). Even better is to try combinations such as case 5 (ranked 5th) or case 4 (ranked 6th). Note, neither of these two cases requires writing complex annotations for precondition/effects, i.e., they require less effort to create, but still may have sufficient performance and hence may be good choices.

8.5. Evaluation III

Our service suggestion algorithms can make forward, backward and bi-directional suggestions. Since forward and backward suggestions are similar except for the difference in direction, in this evaluation we decided to just compare the performance of the forward and the bi-directional suggestion algorithms, which use both forward and backward suggestions for basic calculations.

Based on the results of the first two evaluations, this evaluation uses the path-based data mediation algorithm with annotations on input/output and functionality. Five suggestions are made in this evaluation to suggest services that can be plugged in the middle of service prefixes and suffixes. For the first request, the prefix and suffix are the first and third services in our scenario. Similarly, the second service prefix is the first two service operations and the second suffix is the fourth service operation. The third, fourth and fifth prefix and suffix are set in the same manner. Degree of overlap is calculated for all five requests of the top-5, 10, 15 and 20 suggestions made by the forward and bi-directional suggestion algorithms.

8.5.1. Hypotheses

The bi-directional suggestion algorithm uses both prefix and suffix to calculate ranking scores for all candidate services, so it would supposedly give more accurate suggestions compared to the forward

suggestion algorithm. However, to fairly compare the two algorithms, we have to use the same rankings made by three evaluators for the two algorithms. The three human evaluators rank all the services in the forward way that means they rank the candidate services based on the prefix rather than both prefix and suffix. Therefore, under this condition, the bi-directional suggestion algorithm may filter out some operations that are ranked in higher positions in the forward way, and may not have a higher degree of overlap than forward suggestion algorithm. (Ideally, the human evaluators should re-rank the operations from the perspective of making a bi-directional suggestion.) We also expect our suggestion algorithms could make sufficient suggestions that have much higher degree of overlap compared to the random rankings.

8.5.2. Results

Figures 8.15 to 8.19 show the degree of overlap of the top-5, top-10, top-15 and top-20 suggestions for each request made by the forward and bi-directional suggestion algorithms, as well as the suggestions made by the three human evaluators and a random ranking. In Figure 8.20, the average degree of overlap of all five requests is depicted as well as three human evaluators and a random ranking. (Please see appendix C for all the statistical data for this evaluation.)

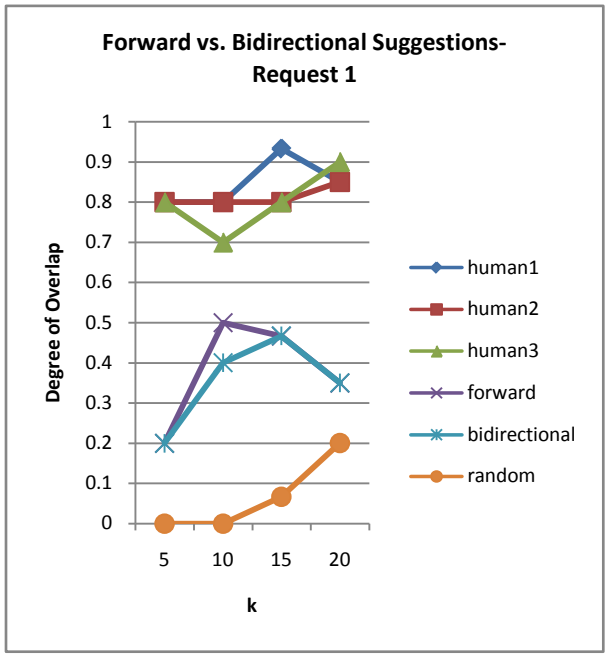

Figure 8.15. Forward vs. bi-directional suggestion algorithms in request 1 (selected operation: "getIds")

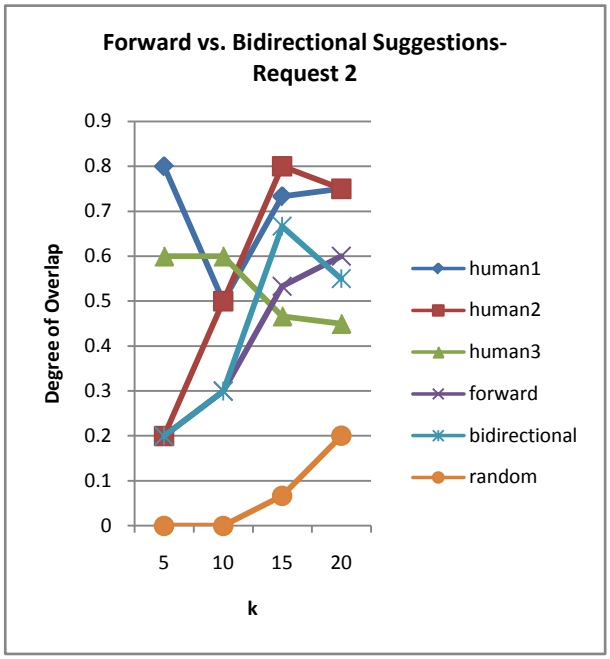

Figure 8.16. Forward vs. bi-directional suggestion algorithms in request 2 (selected operation: "array2string")

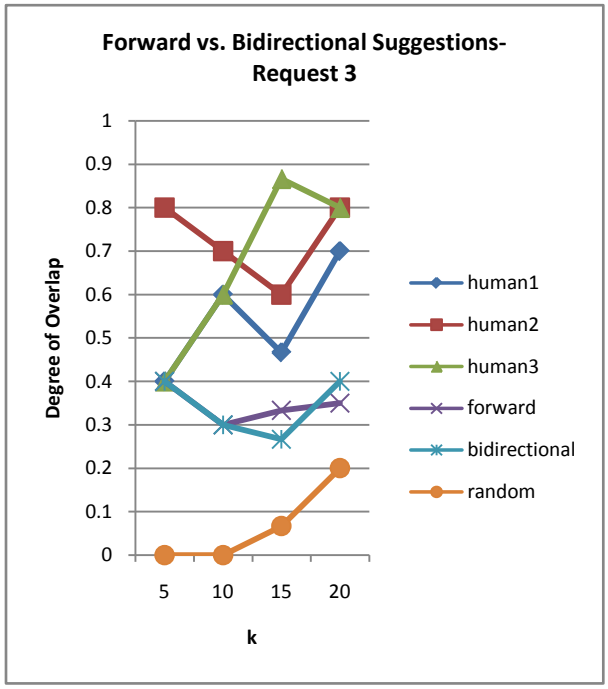

Figure 8.17. Forward vs. bi-directional suggestion algorithms in request 3 (selected operation: "fetchBatch")

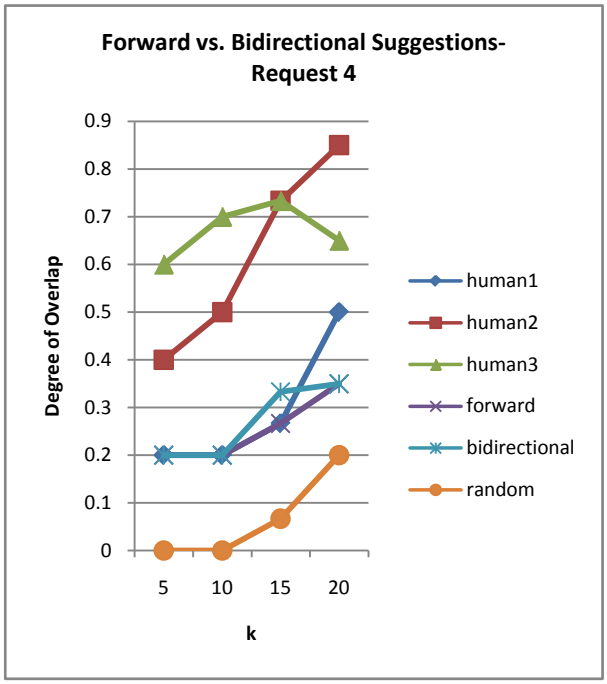

Figure 8.18. Forward vs. bi-directional suggestion algorithms in request 4 (selected operation: "run")

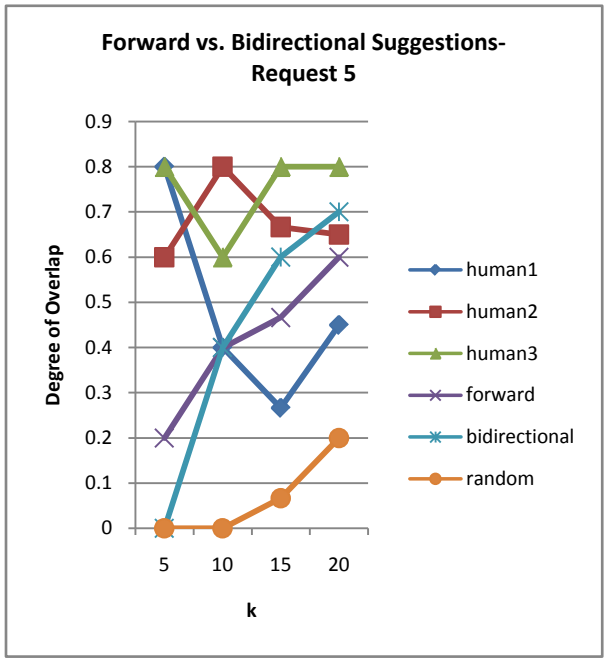

Figure 8.19. Forward vs. bi-directional suggestion algorithms in request 5 (selected operation: "getResult")

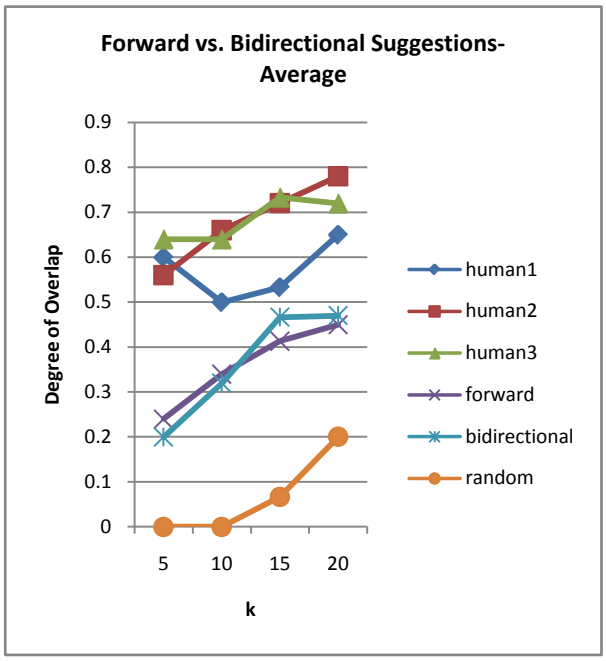

Figure 8.20. Forward vs. bi-directional suggestion algorithms - average

8.5.3. Findings

The results of this evaluation show that the forward and bi-directional suggestion algorithms have a roughly similar performance in terms of degree of overlap. The random ranking has a far lower degree of overlap, which indicates our suggestion algorithms can provide useful suggestions. In most of the requests, the degree of overlap of our suggestion algorithms are close to expert human evaluators, especially the first evaluator, which shows that our suggestion algorithms could help humans by suggesting some services needed to compose a WSC process.

CHAPTER 9

RELATED WORK

In the past decade, many studies and research projects have worked on different approaches for Web service composition. According to Charif-Djebbar et al. [63], Dustdar et al. [64], and Cheng et al. [65], approaches for WSC can be classified into three categories according to the automation level of the approach: manual (with GUI), semi-automatic and automatic approaches. Semantics usually can help with the heterogeneities during WSC and increase the automation level. However, providing more complex semantic annotations may cost too much. As we mentioned in Section 1.4, data mediation and process mediation are two major issues of WSC, so the rest of this chapter will discuss the work related to this dissertation from these points of view.

9.1. Manual Approach with GUI Designer

Many WSC GUI designers have been developed both in industrial and academic areas, which allow users to manually design a WSC process with the help of a graphical interface. Some of the popular WSC

designers are NetBeans BPEL designer[44], ActiveVOS[45], Oracle JDeveloper[46], Eclipse BPEL designer[47], Taverna[48], Kepler [66] and BioExtract [67].

Manual approaches tackle the data mediation and process mediation manually through their graphical interface. More specifically, for data mediation, users have to figure out how the inputs will be fed by other operations for each Web service operation in the WSC process through a menu (e.g. Taverna), popup window (e.g. Oracle JDeveloper), tab (e.g. Eclipse BPEL designer) or even visually drawing lines between matched outputs and inputs (e.g. NetBeans BPEL designer). Figure 9.1 shows that the NetBeans BPEL designer lets users draw lines between matched outputs and inputs. These different graphical interfaces help users create mappings without requiring coding (e.g., of BPEL assign activities) for a WSC process. However, figuring out those matched outputs and inputs is not easy in general.

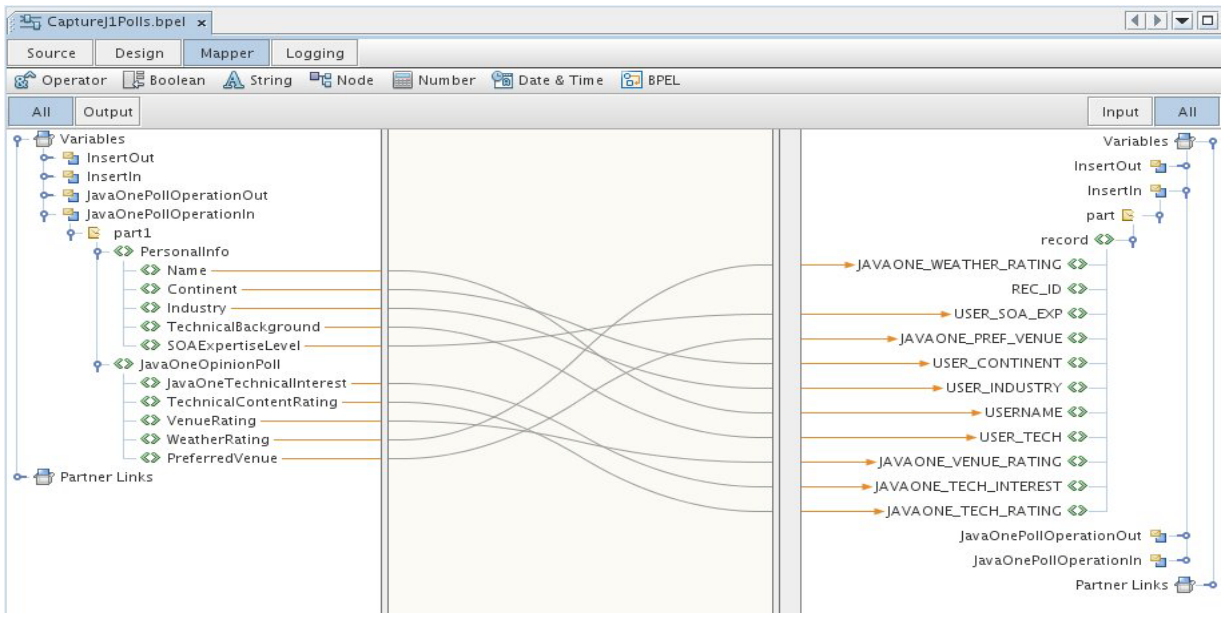

Figure 9.1. NetBeans manual data mediation GUI

[44] NetBeans BPEL: http://soa.netbeans.org/soa/
[45] ActiveVOS: http://www.activevos.com/
[46] Oracle JDeveloper: http://www.oracle.com/technetwork/middleware/bpel/overview/index.html
[47] Eclipse BPEL designer: http://www.eclipse.org/bpel/
[48] Taverna: http://www.taverna.org.uk/

For process mediation, manual approaches let users manually pick Web service operations and connect them together visually through the graphical designers. They generally provide drag-and-drop capability so that users can visually drag their desired operations to the canvas and connect them. Figure 9.2 is the interface of the NetBeans BPEL designer. On the top right are components and structures that users can drag to the canvas to invoke an operation or connect into a structure. These convenient graphical designers enable users to visually create a process and reduce the need for hand-coding. However, which operations to choose and in which manner to connect them still remain hard for users.

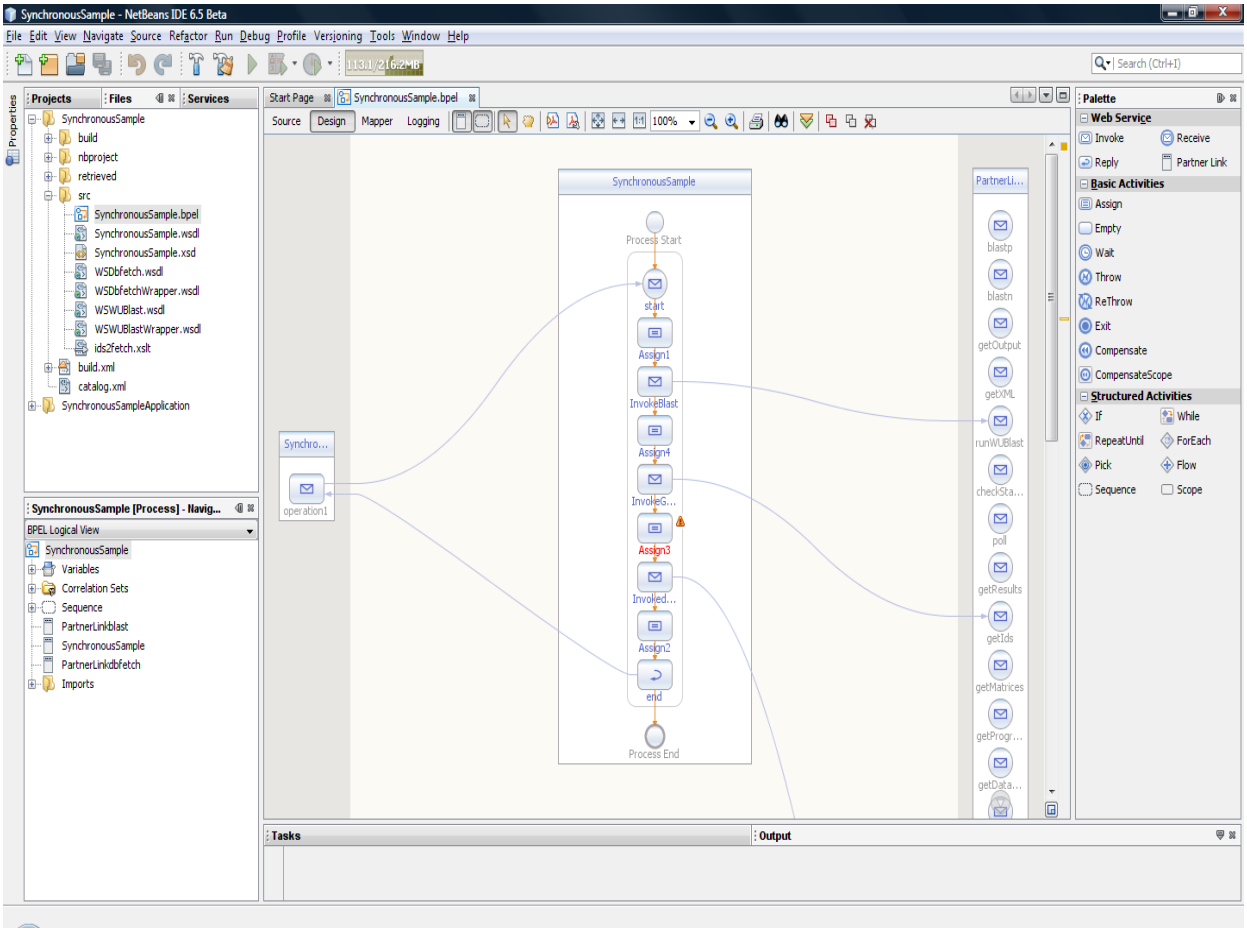

Figure 9.2. NetBeans BPEL designer

9.2. Automatic Approach

Automatic approaches compose Web service operations into a process automatically without users' intervention. In the automatic WSC, a user formulates a process specification (chiefly an initial state and goals), which will be processed and the WSC process will be generated automatically. Many techniques have been applied to automatic WSC including: Golog planning [12], Heuristic Search [68], Estimated-regression planning [13], rule-based mechanism [69], SWORD [14], Simple Hierarchical Ordered Planner 2 (SHOP2) [70], HTN planning [71], Constraint Satisfaction Problem (CSP) planning [15], theorem proving [72], planning by Model Checking [16, 17], finite state machine [73], template-based approach [74], QoS oriented planning [11], MDP planning [18], Colored Petri-net [75, 76], CSP-based Graphplan [19], etc.

Most automatic WSC approaches generally use techniques from the Artificial Intelligence (AI) area, i.e., planning algorithms. [11] uses the QoS properties to guide a heuristic search to generate a plan and an OWL-S process will be generated based on the plan. [13] applies an estimated-regression planning algorithm to WSC. It uses a backward analysis of the difficulty of a goal to guide a forward search through situation space to generate a plan using Planning Domain Definition Language (PDDL). The generated plan is then translated into a WSC process represented by DAML-S. Simple Hierarchical Ordered Planner 2 (SHOP2) [70] is a planning system based on ordered task decomposition. The inputs to their planner are specified in OWL-S that reflects the user requirement. SHOP2 is then used to build a plan, which afterwards is transferred back to an OWL-S description of a WSC process. SWORD [14] applies a rule-based planner for automatic Web service composition utilizing an Entity-Relation (ER) model. Web

services are represented in the form of a Horn rule. A plan will be generated automatically using a rule-based planner. The ASTRO [16, 17] tool set is a platform that supports automatic Web service composition based on Planning by Model Checking. Given a set of existing BPEL processes and a composition requirement, ASTRO translates it into an intermediate file that represents a Planning as Model Checking problem. With the intermediate file, a plan is generated and then translated back to BPEL format. [77] presents a Markov Decision Process (MDP) based approach for automatic Web service composition. A decision-theoretic planning, MDP is used to model the WSC. The solution of a MDP produces a policy to guide the generation of a WSC process. They focus on automatically establishing the process logic, the process mediation issue, but ignore the data mediation problems. The template-based approach [74] customizes an abstract workflow for specific requirements. Abstract activities in templates specify the features of the required services and concrete service can be ranked, selected based on abstract service descriptions, and used to generate executable workflows. The HTN-DL planner, an extension to the HTN planner, is used to handle process mediation, i.e., select templates, rank and select concrete services, and generate the WSC process specified in OWL-S. No data mediation issue has been discussed. All these planning based WSC tools tackle process mediation using planners, so the quality of the generated WSC process relies on the planners, the quality of the domain knowledge, and the problem specification. However, the complexity of the domain knowledge and the problem specification will lead to higher complexity to use the approach. Moreover, describing domain knowledge in sufficient completeness is a challenge to the user, and presenting the problem specification highly and accurately is still a challenge in the Semantic Web itself. The data mediation problems are ignored in most of the automatic approaches, e.g.,

they ignore semantic data heterogeneities and assume all the connected services would have perfectly matched outputs and inputs, which can make their solutions impractical.

Some other approaches do not use planning algorithms, such as [68], which utilizes a heuristic search on a dependency graph of all the candidate services to generate a WSC process specified by OWL-S. The heuristic search algorithm tackles process mediation automatically, but the authors assume every service only has one output, to avoid the data mediation problems. [73] models the WSC as a protocol synthesis problem. It uses a finite state machine (FSM) to model the behaviors of services and automatically generate the WSC process. The data mediation problems are ignored. The automatic approach in [75] tackles both data mediation and process mediation. It handles the process mediation by creating a Colored Petri-net (CPN) that represents the WSC process. Data types are defined as object-oriented classes to resolve data mediation problems, however, no ontology is used to capture the semantic heterogeneities.

9.3. Semi-automatic Approach

The semi-automatic approach allows some degree of human interaction with the computer during WSC, while still providing some automation to ease the users' work. It aims to reduce the disadvantages of the manual and automatic approach, but still keep the advantages of them by combining human and machine capabilities. Many researchers have been working on semi-automatic WSC approaches [78-92].

This dissertation proposes a semi-automatic approach for WSC including both data mediation and process mediation issues. Our data mediation (path-based) performs a semantic structural type checking to handle the heterogeneities between inputs and outputs of service operations. Some other semi-automatic approaches also can tackle the data mediation problems: [78] uses OWL-S to describe Web services, more

specifically, OWL-S profile sub-ontology, so the data mediation problems are tackled by ontology mapping between the sub-ontologies. However, ontology mapping is still an open problem in the semantic Web and users may need more effort to create the sub-ontologies for all the services compared to our approach, which only adds several annotations in the standard WSDL documents. [79] is an OWL-S based approach, which focuses on verifying a given WSC process. It checks the WSC process based on the semantic compatibility between outputs and inputs of the operations in the process, i.e., verifying based on data mediation. Its matching at the element-level only considers identical, generalization and specialization matching, which correspond to our exact, subsumes and subsumed-by matching. We further compare the properties of the semantic concepts with their ranges, cardinalities, etc. Moreover, its data mediation approach ignores the structure of the schema of inputs and outputs and does not perform any type checking. [80] proposes a data mediation approach that is based on matching schema of input and output messages, but it only compares the syntax of message schema without any semantics, so it is not able to resolve the semantic heterogeneities between outputs and inputs of service operations. None of these semi-automatic approaches tackle process mediation problems.

Our process mediation approach ranks all the candidate service operations based on the data mediation, functionality, precondition and effect, and makes suggestions to the user. [81] also tackles process mediation by suggesting services to the user, however, its suggestion algorithm is based on information provided from social network analysis, which implicitly builds from the interactions between users and services, and the different services compositions operated by the user's social network members as well as the global social network. [82] selects services based on QoS parameters, so it handles process mediation based on non-functional requirements. [84] presents a goal decomposition approach to generate

the WSC process. The user provides a goal, which is reduced by repeated partial matching of services until all requirements are satisfied or there are no more services to consider. As a result, the generated process will be presented to the requestor to confirm if it is acceptable. In contrast to our approach, it suggests the whole process to the user rather than single services, which is less flexible since users cannot make decisions on the service level. The approach described by Xu et al. [85] helps users select appropriate services by analyzing dynamic semantic association between services. The dynamic semantics is the semantics of values of IO when the service is executed. The main idea of this approach is to match dynamic semantics of IO as a basic step to select some services, and then add more services based on the inheritance relationships between functionalities of services. Afterwards, all the selected services will be ranked based on the semantic distance of their functionalities in ontology. Michael et al. [86] ranks and suggests suitable Web services to the user by using a lazy breadth-first search over an implicit graph of the service space base on the information presented in the Moby[49] service ontology. Its suggestion algorithm does not consider the parameters that are used in our data mediation, e.g., structure of the message schema or XSD type compatibility, as well as preconditions and effects. [87] uses backward chaining along with filtering to help the user select the next service/operation to add to their composition. Kim et al. [88] developed a novel tool for finding errors or deficiencies in WSC designs and providing written suggestions for how to fix the problems. This tool focuses on finding deficiencies and does not consider data mediation issues in detail. PASSAT [89] promotes an interactive planning to resolve the process mediation problems. [90, 91] use Case-Based Reasoning (CBR) to compose WSC processes semi-automatically. All these approaches listed above do not provide any mechanism to tackle

[49] BioMoby: http://www.biomoby.org/

data mediation problems.

IRS-III [92] is based on WSMO, which deals with both data mediation and process mediation semi-automatically. Different from our data mediation approach, its data mediation is solved by users manually selecting predefined mediators in the system, which is presented as a sub-ontology defined by WSMO aiming to automatically mediate heterogeneities. It might be impossible to predefine all the possibly needed mediators, which can make the approach less practical to use. Its process mediation approach lets the user manually select predefined goals and structures through a GUI, and then the system selects appropriate services based on those selected goals. It is convenient that the user does not have to create the goal, but it might be impossible for a system to predefine all the possible goals needed by all designers, which may limit the usage of this approach.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

Along with the increasing number of Web services implemented and available on the Web, reusing and composing existing Web services together to complete a new task is becoming increasingly important. However, for general users, Web service composition is not an easy task. Focusing on this problem, this dissertation presents an effort to aid the user's work and resolve challenges during WSC. The rest of this chapter will describe the conclusions and contributions of this work and then discuss some future research directions for this work.

10.1. Conclusions

This work seeks to aid users in composing Web services into a process. Three data mediation algorithms, leaf-based, structure-based and path-based data mediation, are developed to address data dependency and heterogeneity during the WSC and to support service suggestion. The path-based and structure-based data mediation algorithms take into consideration the structure of a Web service operation's input/output, the former via the path comparisons in the structure-level matching and the latter using a sub-tree homeomorphism algorithm. The three data mediation algorithms can work with various types and levels of semantic annotations of SAWSDL documents. All three data mediation algorithms consider the global inputs and the outputs from all preceding operations, which allow them to handle

more general cases. An evaluation is presented to compare three data mediation algorithms, which shows that path-based data mediation has a higher degree of overlap compared to the other two. The data mediation engine is implemented as an independent external engine, which can be attached to a WSC designer to solve data heterogeneities between messages. The data mediation engine can automatically detect the different types of inputs, i.e., required, optional, and unknown type input. Different weights are assigned to calculate the matching score, so that the required input has the highest priority, unknown type is the next, and then the optional input.

A formal graph model (called IODAG) for the input/output of a Web service operation is defined. The IODAG models the metadata of input/output for a service operation as a labeled DAG, which represents the structure of the input/output and serves as the basis for our data mediation algorithms.

Type theory has been studied and the data mediation problem is formalized as a semantic structural subtype compatibility problem. The typed representations for path-based and leaf-based data mediation based on type checking are given. The input/output of a Web service operation is formalized as a structural type, which combines the structure, semantic and syntactic type together. Subtyping rules are defined for the type checking to indicate the type safety of the matching. The type checking is performed at both elemental and structural levels. The data mediation results include both a matching score and a type checking result for every pair of matched input and output. Type incompatibility will be detected even if the pair of output and input semantically matches. For example, in our scenario, the output of the operation "*getIds*" is an array of string, "*string*[]," and the input of operation "*fetchBatch*" is a string. Even though semantically they both represent sequence ids, the type checking will detect it as unsafe, and a converter might be needed to convert the data.

A service ranking schema and suggestion algorithm is developed to address the process mediation issue for WSC. It ranks all the candidate Web service operations and suggests them to the user during composition. The suggestion score is calculated based on some important aspects of WSC, i.e., data mediation, precondition/effects and service functionality. The service suggestion algorithm can work with various types and levels of semantic annotation, such as with or without functionality annotations, precondition/effects and annotations for the input and output. An evaluation has been performed to compare the effectiveness of different annotation cases, which shows that complicated annotation cases result in more accurate service suggestions, but need more effort to create; less complicated annotations need less effort to create, but result in less accurate service suggestions. The cases with annotations on input/output and functionality are good to use because they require less effort to create compared to the cases with precondition/efforts, and still result in sufficiently accurate service suggestions. This service suggestion approach is implemented as an independent external engine, which can be attached to a WSC designer to aid users' work during design time. The service suggestion engine can provide forward, backward and bi-directional suggestions, which suggest a Web service operation after, before, or in the middle of the current WSC process.

This work extends the METEOR-S [51] service model and utilizes the precondition/effects to guide Web service composition using our service suggestion algorithm. A Horn logic language (Prolog), which is more expressive than propositional logic, is used to describe preconditions and effects. The Prolog KB is used to represent the states for a WSC process, which is a collection of Prolog facts and rules. A Horn logic KBMS is implemented on top of the Prolog engine, SWI-Prolog, which is able to query the KB for the entailment of the given preconditions and update the KB based on the given effects.

We studied and compared many string metrics to find an appropriate syntactic similarity measure algorithm that is used for our similarity measure, i.e., the Jaro-Winkler string metric. Several concept similarity measure algorithms are studied, customized and extended for use in our data mediation and service suggestion algorithms, such as Verma's [22] and Garlapati's [23] concept similarity measures, which were originally used for Web service discovery.

10.2. Future Work

Although the algorithms and approaches for WSC presented in this dissertation are based on our numerous studies and research, some aspects remain that can be extended or improved in the future.

For forward service suggestions, a planner can be used to complete the rest of the process to estimate the distance from the state after the execution of the candidate operation to the goal state. This distance can be taken into consideration to compute the precondition/effects score to impact the service suggestion. For bi-directional service suggestions, a planner can be used to generate a chain of Web service operations when one Web service operation is not sufficient. The cost and the gain from using the planner is still an issue to be studied.

The current approach suggests the connections between service operations based on the data dependency captured during data mediation, which can be either sequence or parallel. In the future we plan to suggest more complicated control structures, such as "if conditions" and "loops."

The current service suggestion algorithm considers only functional requirements, such as functionality, precondition, effects, input/output, etc. In the future, some non-functional requirements can be taken into account for service suggestions, e.g., QoS parameters, user's preference, etc.

We plan to extend and apply this approach to RESTful Web services. One possible solution is to extend the current approach to support WSDL 2.0, which can specify both SOAP and RESTful Web services' interfaces. Currently, most SOAP Web services are implemented with WSDL 1.1, and there are very few WSDL 2.0 based Web services, so that solution will only be practical if in the future many SOAP and RESTful Web services are described using WSDL2.0. Another alternative is Web Application Description Language (WADL), which is used to describe RESTful Web services. Semantically Annotated Web Application Description Language (SAWADL) can be used to add semantic annotations to WADL.

RIF is recommended by W3C as a standard rule language for the semantic Web. Our current implementation does not use RIF to specify preconditions/effects because there is no mature RIF engine available at the present time. In the future, we plan to switch to RIF when a mature RIF engine is available.

Path alignment can be used for path-based data mediation. As discussed in Section 4.5, aligning two paths at different nodes will result in different matching scores between two paths, which implies a better match between the two paths. However, since leaf nodes hold the value of the input/output, if two paths are not aligned at their leaf nodes, which means a leaf node matches to a non-leaf node, a difficult problem will be how to create a mapping between the leaf node and all the leaf nodes under the non-leaf node to transfer the values from the output to the input. Another improvement for the path-based data mediation can be allowing gaps when comparing two paths. Currently, two paths are compared along with each pair of nodes without gaps. Allowing gaps may give a more accurate matching between the two paths, but we have to watch the cost to see whether the accuracy gained is worth the extra cost, and the

time cost also has to be acceptable for making suggestions to the users.

In addition to the graph homeomorphism algorithm, we also plan to study the graph homomorphism algorithm and apply it to data mediation. The graph homomorphism differs from graph homeomorphism in that it provides no subdividing or smoothing operations and keeps the adjacency relationships between nodes.

Some machine learning algorithms can be used to set the weights in our service suggestion and data mediation algorithms. We can set the target function ($f(\{w_1, \ldots, w_i, \ldots, w_n\}) = x$) as a mapping from all the weights to a value (x), which represents the performance of the result, e.g., degree of overlap of top-5 rankings used in our evaluation. Some open source software, e.g., WEKA (http://www.cs.waikato.ac.nz/ml/weka/) that includes many popular machine learning algorithms such as decision trees, neural networks, etc. can be used. For example, we can run our system with different weight combinations and get different values of x (e.g., degree of overlap of top-5 rankings). A dataset can then be created and fed to WEKA to generate a decision tree or neural network, from which we can get the weight combinations that should result in the best performance (x).

Another improvement we plan to do is to annotate Web services with properties defined in an ontology. Currently, all the concepts used to annotate the Web services are classes. In the future, we might consider using properties to annotate Web services. This requires some extensions to our similarity measure, such as measuring the similarity between class and property in ontology.

We plan to attach our suggestion engine to Galaxy (http://galaxy.psu.edu/), a bioinformatics integration and workflow system, which provides a Web-based GUI designer to compose workflow. As a semi-automatic approach, users should be able to request service suggestions only when they want to.

Therefore, a button or menu item can be added to the workflow design canvas and whenever the user clicks it the service suggestion engine will be fired. The information required by the service suggestion engine has to be collected according to the workflow on the canvas when the user requests suggestion: The topological order of the services (operation names and URI of WSDLs) in the current workflow will be provided to the suggestion engine. The operation names and URIs of the related WSDL documents for the current workflow can be retrieved from the information of every box (each box represents a service) on the canvas. The candidate services consist of those Web service operations that have been added to the Galaxy server. The operation names and URIs of the related WSDL documents for the candidate services were stored previously whenever the services were added to Galaxy. If a service is annotated using SAWSDL, the URI for the OWL document was also stored when the service was added to Galaxy. On the canvas, we also need to add a text field or a upload button, so the user can type in or upload the logic file (e.g., a fragment of Prolog) for the initial state (optional). A drop-down-list, including all the functionality concepts in the OWL ontology referenced from SASWDL documents stored in Galaxy, can be added to the canvas to let the user select the desired functionality (optional). All the information will be fed to the suggestion engine and a small pop-up window (e.g., using <iframe>) will display the suggested services.

BIBLIOGRAPHY

[1] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell, "SAWSDL: Semantic Annotations for WSDL and XML Schema," *IEEE Internet Computing*, vol. 11, pp. 60-67, November/December, 2007.

[2] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, "Web Services Architecture," http://www.w3.org/TR/ws-arch/, February, 2004

[3] T. Gruber, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition*, vol. 5, pp. 199-199, 1993.

[4] D. Fensel, F. van Harmelen, I. Horrocks, D. L. McGuinness, and P. F. Patel-Schneider, "OIL: An Ontology Infrastructure for the Semantic Web," *IEEE Intelligent Systems*, vol. 16, pp. 38-45, March-April, 2001.

[5] F. Baader, S. Brandt, and C. Lutz, "Pushing the El Envelope," in *19th Joint International Conference on Artificial Intelligence (IJCAI)*, Edinburgh, Scotland, UK, pp. 364-369, July-August, 2005.

[6] C. Feier, D. Roman, A. Polleres, J. Domingue, M. Stollberg, and D. Fensel, "Towards Intelligent Web Services: Web Service Modeling Ontology (WSMO)," in *International Conferemce on Intelligent Computing (ICIC)*, Hefei, China, pp. 23-26, August, 2005.

[7] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara, "Bringing Semantics to Web

Services: The OWL-S Approach," in *First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, San Diego, California, USA, pp. 26-42, July, 2004.

[8] R. Akkiraju, J. Farell, J. A. Miller, M. Nagarajan, A. Sheth, and K. Verma, "Web Service Semantics - WSDL-S," in *W3C Workshop on Frameworks for Semantics in Web Service (W3CW)*, Innsbruck, Austria, pp. 1-5, June 2005.

[9] M. Nagarajan, K. Verma, A. P. Sheth, and J. A. Miller, "Ontology Driven Data Mediation in Web Services," *International Journal of Web Services Research (JWSR) ,USA*, vol. 4, pp. 104-126, Dec 2007.

[10] Z. Wu, A. Ranabahu, K. Gomadam, A. P. Sheth, and J. A. Miller, "Automatic Composition of Semantic Web Services Using Process Mediation," in *9th International Conference on Enterprise Information Systems (ICEIS)*, Funchal, Portugal pp. 453-461, Jun. 2007.

[11] J. M. Ko, C. O. Kim, and I.-H. Kwon, "Quality-of-Service Oriented Web Service Composition Algorithm and Planning Architecture," *Journal of Systems and Software*, vol. 81, pp. 2079-2090, November 2008.

[12] S. Sohrabi, N. Prokoshyna, and S. McIlraith, "Web Service Composition Via the Customization of Golog Programs with User Preferences," *Conceptual Modeling: Foundations and Applications*, vol. 5600, pp. 319-334, 2009.

[13] D. McDermott, "Estimated-Regression Planning for Interactions with Web Services," in *6th International Conference on AI Planning and Scheduling*, Toulouse, France, pp. 204-211, April 2002.

[14] S. R. Ponnekanti and A. Fox, "SWORD: A Developer Toolkit for Web Service Composition," in

11th World Wide Web Conference, Honolulu, HI, USA, May 2002.

[15] D. Maruyama, I. Paik, and M. Shinozawa, "A Flexible and Dynamic Csp Solver for Web Service Composition in the Semantic Web Environment," in *Sixth IEEE International Conference on Computer and Information Technology*, Seoul, Korea, p. 43. September 2006.

[16] M. Trainotti, M. Pistore, F. Barbon, P. Bertoli, A. Marconi, P. Traverso, and G. Zacco, "ASTRO: Supporting Web Service Development by Automated Composition, Monitoring and Verfication," in *Proceedings of the 16th International Conference on Automated Planning and Scheduling (Software Demonstration)*, The English Lake District, Cumbria, U.K. , pp. 28-31, June 2006.

[17] A. Marconi, M. Pistore, and P. Traverso, "Automated Composition of Web Services: The Astro Approach," *IEEE Data Engineering Bulletin*, vol. 31, pp. 23-26, 2008.

[18] C. Kun, X. Jiuyun, and S. Reiff-Marganiec, "Markov-HTN Planning Approach to Enhance Flexibility of Automatic Web Service Composition," in *IEEE International Conference on Web Services (ICWS)*, Los Angeles, CA, USA, pp. 9-16, July 2009.

[19] B. Yang and Z. Qin, "Semantic Web Service Composition Using Graphplan," in *4th IEEE Conference on Industrial Electronics and Applications*, Xi'an, China, pp. 459-63, 25-27 May 2009

[20] A. ten Teije, F. van Harmelen, and B. Wielinga, "Configuration of Web Services as Parametric Design," *Engineering Knowledge in the Age of the SemanticWeb*, vol. 3257, pp. 321-336, 2004.

[21] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, vol. 215, pp. 403-410, October 1990.

[22] K. Verma, A. Sheth, S. Oundhakar, K. Sivashanmugam, and J. A. Miller, "Allowing the Use of

Multiple Ontologies for Discovery of Web Services in Federated Registry Environment," Department of Computer Science, University of Georgia, Athens, Georgia. Technical Report #UGA-CS-LSDIS-TR-07-011, pp. 1-27, February 2007.

[23] S. Garlapati, "A Comparison of SAWSDL Based Semantic Web Service Discovery Algorithms," in *Department of Computer Science* Athens, GA, U.S.: University of Georgia, Master's Thesis, August 2010.

[24] M. Klusch, B. Fries, and K. Sycara, "Automated Semantic Web Service Discovery with OWLS-MX," in *5th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, Hakodate, Japan: ACM Press, pp. 915-922, April 2006.

[25] G. Kondrak, "N-Gram Similarity and Distance," in *Twelfth International Conference on String Processing and Information Retrieval (SPIRE)*, Buenos Aires, Argentina, pp. 115-126, November 2005.

[26] W. E. Winkler, "The State of Record Linkage and Current Research Problems," Technical Report, Statistical Research Division, U.S. Census Bureau., 1999.

[27] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*: Prentice Hall, 2001.

[28] S. Thompson, *Type Theory and Functional Programming*. Redwood City, CA: Addison Wesley Longman Publishing Co., Inc., 1991.

[29] R. Moten, "The Need for Type Theory in Semantic Web Services," in *14th International World Wide Web Conference (WWW)*, Chiba, Japan, May 2005.

[30] S. Jones, D. Vytiniotis, S. Weirich, and M. Shields, "Practical Type Inference for Arbitrary-Rank

Types," *Journal of Functional Programming,* vol. 17, pp. 1-82, January 2007.

[31] J. Mitchell, "Type Inference with Simple Subtypes," *Journal of Functional Programming,* vol. 1, pp. 245-285, July 1991.

[32] M. Sulzmann, "A General Type Inference Framework for Hindley/Milner Style Systems," in *5th International Symposium on Functional and Logic Programming,* Tokyo, Japan, pp. 248-263, March 2001.

[33] K. Hristova, T. Rothamel, Y. A. Liu, and S. D. Stoller, "Efficient Type Inference for Secure Information Flow," in *Workshop on Programming Languages and Analysis for Security,* Ottawa, Ontario, Canada, pp. 85-94, June 2006.

[34] T. Milo, D. Suciu, and V. Vianu, "Typechecking for XML Transformers," *Journal of Computer and System Sciences,* vol. 66, pp. 66-97, February 2003.

[35] T. Milo and D. Suciu, "Type Inference for Queries on Semistructured Data," in *Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Philadelphia, Pennsylvania, United States, pp. 215-226, May-June 1999.

[36] V. Simonet, "Type Inference with Structural Subtyping: A Faithful Formalization of an Efficient Constraint Solver," *Programming Languages and Systems,* vol. 2895, pp. 283-302, November 2003.

[37] R. Hindley, "The Principal Type-Scheme of an Object in Combinatory Logic," *Transactions of the American Mathematical Society,* vol. 146, pp. 29-60, December 1969.

[38] R. A. Milner, "Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences,* vol. 17, pp. 348-375, 1978.

[39] Y. Leontiev, M. Özsu, and D. Szafron, "On Type Systems for Object-Oriented Database Programming Languages," *ACM Computing Surveys (CSUR)*, vol. 34, pp. 409-449, December 2002.

[40] C. Russo, "Types for Modules," *Electronic Notes in Theoretical Computer Science*, vol. 60, pp. 3-421, April 2004.

[41] H. Cai, S. Eisenbach, A. Shafarenko, and C. Grelck, "Extending the S-Net Type System," in *Æther-Morpheus Workshop from Reconfigurable to Self-Adaptive Computing (AMWAS)*, Paris, France, October 2007.

[42] O. Agesen, "Concrete Type Inference: Delivering Object-Oriented Applications," in *Department of Computer Science* Stanford, CA, USA: Stanford University, Ph.D. Dissertation, December 1995.

[43] J. Munkres, "Algorithms for the Assignment and Transportation Problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, pp. 32-38, March 1957.

[44] Z. Wang, J. A. Miller, J. C. Kissinger, R. Wang, D. Brewer, and C. Aurrecoechea, "WS-BioZard: A Wizard for Composing Bioinformatics Web Services," in *IEEE International Workshop on Scientific Workflows (SWF'08), in conjunction with IEEE International Conference on Services Computing (SCC'08)*, Honolulu, Hawaii, 2008, pp. 437-444, Jul. 2008.

[45] P. V. Biron, K. Permanente, and A. Malhotra, "XML Schema Datatypes," http://www.w3.org/TR/xmlschema-2, October 2004.

[46] J. L. Gross and J. Yellen, *Graph Theory and Its Applications, Discrete Mathematics and Its Applications (2nd Ed.)*. Boca Raton, FL: Chapman & Hall/CRC, 2006.

[47] S. Fortune, J. Hopcroft, and J. Wyllie, "The Directed Subgraph Homeomorphism Problem," *Theoretical Computer Science,* vol. 10, pp. 111-121, February 1980.

[48] R. Y. Pinter, O. Rokhlenko, D. Tsur, and M. Ziv-Ukelson, "Approximate Labelled Subtree Homeomorphism," *Journal of Discrete Algorithms,* vol. 6, pp. 480-496, September 2008.

[49] M. Fredman and R. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the ACM (JACM),* vol. 34, pp. 596-615, July 1987.

[50] C. Bourke, R. Tewari, and N. V. Vinodchandran, "Directed Planar Reachability Is in Unambiguous Log-Space," *ACM Transaction on Computational Theory,* vol. 1, pp. 1-17, February 2009.

[51] R. Aggarwal, K. Verma, J. A. Miller, and W. Milnor, "Constraint Driven Web Service Composition in METEOR-S," in *IEEE International Conference on Services Computing, 2004. (SCC),* Shanghai, China, pp. 23 - 30, September 2004.

[52] V. Levenshtein, "Binary Codes Capable of Correcting Spurious Insertions and Deletions of Ones," *Problems of Information Transmission,* vol. 1, pp. 8-17, January-March 1965.

[53] W. Cohen, P. Ravikumar, and S. Fienberg, "A Comparison of String Distance Metrics for Name-Matching Tasks," in *International Joint Conference on Artificial Intelligence (IJCAI),* Acapulco, Mexico, pp. 73-78, August 2003

[54] J. Piskorski, M. Sydow, and K. Wieloch, "Comparison of String Distance Metrics for Lemmatisation of Named Entities in Polish," *Human Language Technology. Challenges of the Information Society,* pp. 413-427, August 2009.

[55] D. Lin, "An Information-Theoretic Definition of Similarity," in *15th International Conference on*

Machine Learning, Madison, Wisconsin, USA, pp. 296–304, July 1998.

[56] R. Rada, H. Mili, E. Bicknell, and M. Blettner, "Development and Application of a Metric on Semantic Nets," *IEEE Transactions on Systems Management and Cybernetics,* vol. 1, pp. 17-30, January 1989.

[57] P. Resnik, "Semantic Similarity in a Taxonomy: An Information-Based Measure and Its Application to Problems of Ambiguity in Natural Language," *Journal of Artificial Intelligence Research,* vol. 11, pp. 95-130, 1999.

[58] J. H. Lee, M. H. Kim, and Y. J. Lee, "Information Retrieval Based on Conceptual Distance in Is-a Hierarchies," *Journal of Documentation,* vol. 49, pp. 188-207, June 1993.

[59] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara, "Semantic Matching of Web Services Capabilities," in *1st International Semantic Web Conference (ISWC)*, Las Vegas, Nevada, pp. 333-347, June 2003.

[60] Gonzales-Castillo, D. Trastour, and C. Bartolini, "Description Logics for Matchmaking of Services," in *Workshop on Application of Description Logics*, Vienna, Austria, September 2001

[61] M. Klusch, P. Kapahnke, and I. Zinnikus, "Hybrid Adaptive Web Service Selection with SAWSDL-Mx and WSDL Analyzer," in *6th Annual European Semantic Web Conference (ESWC 2009)*, Heraklion, Greece, pp. 550-564, May-June 2009.

[62] J. Cardoso, J. A. Miller, and S. Emaini, "Web Services Discovery Utilizing Semantically Annotated WSDL," *Reasoning Web 2008, Lecture Notes in Computer Science (LNCS),* vol. 5224, pp. 240-268, September 2008.

[63] Y. Charif-Djebbar and N. Sabouret, "Dynamic Web Service Selection and Composition: An

Approach Based on Agent Dialogues," in *International Conference on Service-Oriented Computing (ICSOC)*, pp. 515-521, December 2006.

[64] S. Dustdar and W. Schreiner, "A Survey on Web Service Composition," *International Journal of Web and Grid Services*, vol. 1, pp. 1-30, August 2005.

[65] R. Cheng, S. Su, F. Yang, and Y. Li, "Using Case-Based Reasoning to Support Web Service Composition," in *International Conference on Computational Science (ICCS)*, University of Reading, UK, pp. 87-94, May 2006.

[66] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaegar, M. Jones, E. Lee, J. Tao, and Y. Zhao, "Scientific Workflow Management and the Kepler System," *Concurrency and Computation*, vol. 18, pp. 1039-1065, August 2006.

[67] C. Lushbough, M. Bergman, C. Lawrence, D. Jennewein, and V. Brendel, "BioExtract Server - an Integrated Workflow-Enabling System to Access and Analyze Heterogeneous, Distributed Biomolecular Data," *IEEE/ACM Transactions on Computational Biology and Bioinformatics* vol. 7, pp. 12-24, January 2010.

[68] N. Ukey, R. Niyogi, A. Milani, and K. Singh, "A Bidirectional Heuristic Search Technique for Web Service Composition," in *11th International Conference on Computational Science and Its Applications (ICCSA)*, Santander, Spain, pp. 309-320, June 2010.

[69] Y. Yujie and C. Haopeng, "A Rule-Based Web Service Composition Approach," in *Sixth International Conference on Autonomic and Autonomous Systems (ICAS)*, Cancun, Mexico, pp. 150-155, March 2010.

[70] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "HTN Planning for Web Service Composition

Using Shop2," *Journal of Web Semantics,* vol. 1, pp. 377-396, October 2004.

[71] J. Zhang, S. Zhang, J. Cao, and Y. Mou, "Improved HTN Planning Approach for Service Composition," in *IEEE International Conference on Service Computing*, Shanghai, China, pp. 609-612, September 2004.

[72] J. Rao, P. Küngas, and M. Matskin, "Composition of Semantic Web Services Using Linear Logic Theorem Proving," *Information Systems,* vol. 31, pp. 340-360, June-July 2006.

[73] R. Ragab Hassen, L. Nourine, and F. Toumani, "Protocol-Based Web Service Composition," in *6th International Conference on Service Oriented Computing (ICSOC)*, Sydney, Austrialia, pp. 38-53, December 2008.

[74] E. Sirin, B. Parsia, and J. Hendler, "Template-Based Composition of Semantic Web Services," in *AAAI Fall Symposium on Agents and the Semantic Web*, Virginia, USA, pp. 85-92, November 2005.

[75] Z. Zhang, F. Hong, and H. Xiao, "A Colored Petri Net-Based Model for Web Service Composition," *Journal of Shanghai University (English Edition),* vol. 12, pp. 323-329, January 2008.

[76] X. Yi and K. Kochut, "A CP-Nets-Based Design and Verification Framework for Web Services Composition," in *IEEE International Conference on Web Services (ICWS)*, San Diego, California, pp. 756-760, July 2004.

[77] P. Doshi, R. Goodwin, R. Akkiraju, and K. Verma, "Dynamic Workflow Composition: Using Markov Decision Processes," *International Journal of Web Service Research,* vol. 2, pp. 1-17, 2005.

[78] S. Izza, L. Vincent, and P. Burlat, "Exploiting Semantic Web Services in Achieving Flexible Application Integration in the Microelectronics Field," *Computers in industry,* vol. 59, no. 7, pp. 722-740, September 2008.

[79] N. Lebreton, C. Blanchet, D. Claro, J. Chabalier, A. Burgun, and O. Dameron, "Verification of Parameters Semantic Compatibility for Semi-Automatic Web Service Composition: A Generic Case Study," in *International Conference on Information Integration and Web-based Applications & Services (iiWAS),* Paris, France, November 2010.

[80] A. Gao, D. Yang, and S. Tang, "Web Service Composition Based on Message Schema Analysis," *Advances in Databases: Concepts, Systems and Applications,* vol. 4443, pp. 918-923, August 2007.

[81] A. Maaradji, H. Hacid, J. Daigremont, and N. Crespi, "Social Composer: A Social-Aware Mashup Creation Environment," in *ACM Conference on Computer Supported Cooperative Work (CSCW),* Savannah, Georgia, USA, pp. 549-550, February 2010.

[82] X. Fan, C. Jiang, and X. Fang, "An Efficient Approach to Web Service Selection," in *International Conference on Web Information Systems and Mining (WISM),* Amsterdam, Netherlands, pp. 271-280, June 2009.

[83] I. B. Arpinar, R. Zhang, B. Aleman-Meza, and A. Maduko, "Ontology-Driven Web Services Composition Platform," *Information Systems and E-Business Management,* vol. 3, pp. 175-199, June 2005.

[84] M. Naeem, R. Heckel, and F. Orejas, "Semi-Automated Service Composition Using Visual Contracts," in *International Conference on Frontiers of Information Technology (FIT),*

Abbottabad, Pakistan, pp. 1-6, December 2009.

[85] M. Xu, J. Chen, Y. Peng, X. Mei, and C. Liu, "A Dynamic Semantic Association-Based Web Service Composition Method," in *IEEE/WIC/ACM International Conference on Web Intelligence*, Hong Kong, pp. 666-672, December 2006.

[86] D. Michael, P. Rachel, and W. Mark, "Semi-Automatic Web Service Composition for the Life Sciences Using the Biomoby Semantic Web Framework," *Journal of Biomedical Informatics*, vol. 41, pp. 837-847, Oct 2008.

[87] E. Sirin, B. Parsia, and J. Hendler, "Filtering and Selecting Semantic Web Services with Interactive Composition Techniques," *IEEE Intelligent Systems*, vol. 19, pp. 42-49, July-August 2004.

[88] J. Kim, M. Spraragen, and Y. Gil, "An Intelligent Assistant for Interactive Workflow Composition," in *Ninth international conference on Intelligent User Interface (IUI)*, Funchal, Madeira Island, Portugal, pp. 125-131, January 2004.

[89] K. L. Myers, W. M. Tyson, M. J. Wolverton, P. A. Jarvis, T. J. Lee, and M. desJardins, "PASSAT: A User-Centric Planning Framework," in *Third International NASA Workshop on Planning and Scheduling for Space*, Houston, TX, USA, October 2002.

[90] E. Chinthaka, J. Ekanayake, D. Leake, and B. Plale, "CBR Based Workflow Composition Assistant," in *Congress on Services*, Los Angeles, California, USA pp. 352-355, July 2009.

[91] S. Lajmi, C. Ghedira, and K. Ghedira, "CBR Method for Web Service Composition," *Advanced Internet Based Systems and Applications. Second International Conference on Signal-Image Technology and Internet-Based Systems (SITIS 2006, Hammamet, Tunisia), Revised Selected*

Papers, pp. 314-26, 2009.

[92] F. Hakimpour, D. Sell, L. Cabral, J. Domingue, and E. Motta, "Semantic Web Service Composition in IRS-III: The Structured Approach," in *7th International IEEE Conference on E-Commerce Technology (CEC)*, Technische Universität München, Germany, pp. 484-487, July 2005.

APPENDIX A

SAWSDL ANNOTATIONS FOR WSDL2.0

Table A.1. Allowable SAWSDL annotations for WSDL 2.0

Annotation / Tag	modelReference	lifting SchemaMapping	lowering SchemaMapping
<interface>	Yes	No	No
<operation>	Yes	No	No
<element>	Yes	Yes	Yes
<complexType>	Yes	Yes	Yes
<simpleType>	Yes	Yes	Yes
<attribute>	Yes	No	No
<fault>	Yes	No	No

APPENDIX B

RANKINGS OF CANDIDATE WEB SERVICES

Table B.1 Rankings from the first evaluator

Web service	Operation	1	2	3	4	5	6
WU-BLAST http://www.ebi.ac.uk/Tools/webservices/wsdl/WSWUBlast.wsdl	1. blastp	25	21	25	6	45	46
	2. blastn	26	22	26	7	46	47
	3. getOutput	2	34	33	31	36	36
	4. getXML	3	35	34	32	37	37
	5. runWUBlast	23	19	23	4	43	44
	6. getIds	1	40	35	33	38	39
	7. checkStatus	4	36	37	34	39	40
	8. poll	6	37	38	35	40	41
	9. getResults	5	38	36	36	41	42
	10. getMatrices	8	8	7	11	5	3
	11. getPrograms	9	9	8	12	6	4
	12. getDatabases	10	10	9	13	7	5
	13. getSort	11	11	10	14	8	6
	14. getStats	12	12	11	15	9	7
	15. getXmlFormats	13	13	12	16	10	8
	16. getSensitivity	14	14	13	17	11	9
	17. getFilters	15	15	14	18	12	10
	18. polljob	7	39	39	37	42	43
	19. doWUBlast	24	20	24	5	44	45
WSDbfetch http://www.ebi.ac.uk/ws/services/WSDbfetchDoclit?wsdl	20. fetchBatch	32	3	1	49	47	48
	21. fetchData	33	2	2	50	48	49
	22. getDatabaseInfo	34	24	18	26	22	20
	23. getDatabaseInfoList	16	4	6	19	13	11
	24. getDbFormats	35	25	19	27	23	21
	25. getFormatInfo	36	26	20	28	24	22
	26. getFormatStyles	37	27	21	29	25	23
	27. getStyleInfo	38	28	22	30	26	24

	28. getSupportedDBs	17	5	3	20	14	12
	29. getSupportedFormats	18	6	4	21	15	13
	30. getSupportedStyles	19	7	5	22	16	14
ClustalW2 http://www.ebi.ac.uk/Tools/services/soap/clustalw2?wsdl	31. getParameters	20	16	15	8	4	2
	32. getParameterDetails	39	29	28	23	19	17
	33. run	40	32	31	1	27	33
	34. getStatus	41	42	40	40	2	25
	35. getResultTypes	42	42	41	38	3	26
	36. getResult	43	43	42	39	1	27
WSConverter WSConverter.wsdl	37. array2string	44	1	49	47	49	50
	38. base64toString	45	44	50	48	50	1
tcoffee http://www.ebi.ac.uk/Tools/services/soap/tcoffee?wsdl	39. getParameters	21	17	16	9	17	15
	40. getParameterDetails	46	30	29	24	20	18
	41. run	47	33	32	2	31	34
	42. getStatus	48	45	43	43	28	27
	43. getResultTypes	49	46	44	41	29	28
	44. getResult	50	47	45	42	30	29
ncbiblast http://www.ebi.ac.uk/Tools/services/soap/ncbiblast?wsdl	45. getParameters	22	18	17	10	18	16
	46. getParameterDetails	31	31	30	25	21	19
	47. run	27	23	27	3	35	35
	48. getStatus	28	48	46	46	32	30
	49. getResultTypes	29	49	47	44	33	31
	50. getResult	30	50	48	45	34	32

Table B.2 Rankings from the second evaluator

Web service	Operation	1	2	3	4	5	6
WU-BLAST http://www.ebi.ac.uk/Tools/webservices/wsdl/WSWUBlast.wsdl	1. blastp	19	22	42	49	16	34
	2. blastn	18	21	43	48	17	35
	3. getOutput	17	20	44	46	15	30
	4. getXML	16	3	45	47	15	29
	5. runWUBlast	49	49	49	45	13	36
	6. getIds	1	48	41	44	18	28
	7. checkStatus	2	23	47	29	12	33
	8. poll	4	47	32	30	45	2
	9. getResults	3	2	46	26	11	18
	10. getMatrices	48	46	38	27	47	26
	11. getPrograms	5	8	39	28	48	25

	12. getDatabases	6	13	40	18	49	24
	13. getSort	7	4	33	19	44	19
	14. getStats	8	5	34	20	43	20
	15. getXmlFormats	9	7	35	21	42	23
	16. getSensitivity	10	6	36	22	41	22
	17. getFilters	11	24	37	25	46	21
	18. polljob	15	42	50	24	10	27
	19. doWUBlast	50	50	48	23	50	37
WSDbfetch http://www.ebi.ac.uk/ws/services/WSDbfetchDoclit?wsdl	20. fetchBatch	21	9	1	7	19	31
	21. fetchData	22	10	2	8	20	32
	22. getDatabaseInfo	23	12	3	12	22	9
	23. getDatabaseInfoList	24	11	5	11	21	10
	24. getDbFormats	26	37	6	9	25	15
	25. getFormatInfo	29	35	16	13	23	11
	26. getFormatStyles	28	40	17	15	27	13
	27. getStyleInfo	30	36	20	14	24	12
	28. getSupportedDBs	25	39	14	16	26	17
	29. getSupportedFormats	27	38	18	10	28	16
	30. getSupportedStyles	31	41	19	17	29	14
ClustalW2 http://www.ebi.ac.uk/Tools/services/soap/clustalw2?wsdl	31. getParameters	32	14	7	21	30	3
	32. getParameterDetails	33	17	8	3	31	4
	33. run	12	35	21	1	36	38
	34. getStatus	38	25	22	33	7	41
	35. getResultTypes	44	26	27	36	2	43
	36. getResult	41	29	13	37	1	47
WSConverter WSConverter.wsdl	37. array2string	20	1	30	42	39	50
	38. base64toString	47	34	31	43	40	1
tcoffee http://www.ebi.ac.uk/Tools/services/soap/tcoffee?wsdl	39. getParameters	34	15	9	4	32	5
	40. getParameterDetails	35	18	10	5	33	6
	41. run	13	33	23	31	37	39
	42. getStatus	39	26	24	34	9	42
	43. getResultTypes	45	30	28	38	4	45
	44. getResult	42	27	4	39	3	48
ncbiblast http://www.ebi.ac.uk/Tools/services/soap/ncbiblast?wsdl	45. getParameters	36	16	11	6	34	7
	46. getParameterDetails	37	19	12	7	35	8
	47. run	14	32	25	32	38	40
	48. getStatus	40	27	26	35	8	43
	49. getResultTypes	46	31	29	40	6	46
	50. getResult	43	28	15	41	5	49

Table B.3 Rankings from the third evaluator

Web service	Operation	1	2	3	4	5	6
WU-BLAST http://www.ebi.ac.uk/Tools/webservices/wsdl/WSWUBlast.wsdl	1. blastp	20	29	49	31	48	32
	2. blastn	19	34	50	32	49	33
	3. getOutput	5	35	23	30	24	18
	4. getXML	6	36	24	29	23	17
	5. runWUBlast	7	30	19	25	19	14
	6. getIds	1	31	21	26	20	15
	7. checkStatus	8	33	36	28	22	19
	8. poll	4	32	20	27	21	20
	9. getResults	2	37	25	33	25	21
	10. getMatrices	9	38	26	34	26	22
	11. getPrograms	10	39	27	35	27	23
	12. getDatabases	11	40	28	36	28	24
	13. getSort	12	41	29	37	29	25
	14. getStats	13	42	30	38	30	26
	15. getXmlFormats	14	43	31	39	31	27
	16. getSensitivity	15	44	32	40	32	28
	17. getFilters	16	45	33	41	33	29
	18. polljob	17	46	34	42	34	30
	19. doWUBlast	18	47	35	43	35	31
WSDbfetch http://www.ebi.ac.uk/ws/services/WSDbfetchDoclit?wsdl	20. fetchBatch	21	2	1	44	37	34
	21. fetchData	22	3	8	1	38	35
	22. getDatabaseInfo	23	4	9	2	39	36
	23. getDatabaseInfoList	24	15	10	3	40	37
	24. getDbFormats	25	16	11	4	41	38
	25. getFormatInfo	25	17	12	5	42	39
	26. getFormatStyles	27	18	13	6	43	40
	27. getStyleInfo	28	19	14	7	44	41
	28. getSupportedDBs	29	20	15	8	45	42
	29. getSupportedFormats	30	21	16	9	46	43
	30. getSupportedStyles	31	22	17	10	47	44
ClustalW2 http://www.ebi.ac.uk/Tools/services/soap/clustalw2?wsdl	31. getParameters	35	6	5	45	5	9
	32. getParameterDetails	36	7	7	14	4	10
	33. run	32	5	2	11	50	8
	34. getStatus	37	8	6	16	3	11
	35. getResultTypes	38	9	7	15	2	12

	36. getResult	33	10	3	12	1	13
WSConverter WSConverter.wsdl	37. array2string	3	1	22	27	21	16
	38. base64toString	34	11	4	13	6	1
tcoffee http://www.ebi.ac.uk/Tools/services/soap/tcoffee?wsdl	39. getParameters	41	48	39	21	9	3
	40. getParameterDetails	42	49	40	22	10	4
	41. run	39	12	37	17	7	2
	42. getStatus	43	15	41	18	11	5
	43. getResultTypes	44	14	42	19	12	6
	44. getResult	40	13	38	20	8	7
ncbiblast http://www.ebi.ac.uk/Tools/services/soap/ncbiblast?wsdl	45. getParameters	47	23	45	46	18	45
	46. getParameterDetails	48	24	46	47	17	46
	47. run	45	25	43	23	13	47
	48. getStatus	49	26	47	25	15	48
	49. getResultTypes	50	27	48	26	16	49
	50. getResult	46	28	44	24	14	50

APPENDIX C

STATISTICAL DATA FROM EVALUATIONS

Table C.1. Degree of overlap of request 1 for evaluation I

	human1	human2	human3	leaf	struct	path	random
Top 5	0.8	0.8	0.8	0.4	0.6	0.2	0
Top 10	0.8	0.8	0.7	0.5	0.7	0.5	0
Top 15	0.933333	0.8	0.8	0.466667	0.6	0.466667	0.066667
Top 20	0.85	0.85	0.9	0.45	0.7	0.35	0.2

Table C.2. Degree of overlap of request 2 for evaluation I

	human1	human2	human3	leaf	struct	path	random
Top 5	0.8	0.2	0.6	0	0	0.2	0
Top 10	0.5	0.5	0.6	0	0.3	0.4	0
Top 15	0.733333	0.8	0.466667	0.2	0.533333	0.533333	0.066667
Top 20	0.75	0.75	0.45	0.25	0.5	0.6	0.2

Table C.3. Degree of overlap of request 3 for evaluation I

	human1	human2	human3	leaf	struct	path	random
Top 5	0.4	0.8	0.4	0	0.4	0.4	0
Top 10	0.6	0.7	0.6	0.2	0.2	0.2	0
Top 15	0.466667	0.6	0.866667	0.266667	0.133333	0.4	0.066667
Top 20	0.7	0.8	0.8	0.4	0.45	0.45	0.2

Table C.4. Degree of overlap of request 4 for evaluation I

	human1	human2	human3	leaf	struct	path	random
Top 5	0.2	0.4	0.6	0.2	0	0.2	0
Top 10	0.2	0.5	0.7	0.3	0.2	0.2	0
Top 15	0.266667	0.733333	0.733333	0.4	0.2	0.2	0.066667
Top 20	0.5	0.85	0.65	0.4	0.25	0.25	0.2

Table C.5. Degree of overlap of request 5 for evaluation I

	human1	human2	human3	leaf	struct	path	random
Top 5	0.8	0.6	0.8	0.2	0.2	0.4	0
Top 10	0.4	0.8	0.6	0.5	0.5	0.5	0
Top 15	0.266667	0.666667	0.8	0.466667	0.466667	0.466667	0.066667
Top 20	0.45	0.65	0.8	0.45	0.5	0.5	0.2

Table C.6. Degree of overlap of request 6 for evaluation I

	human1	human2	human3	leaf	struct	path	random
Top 5	0.4	0.8	0.6	0.4	0.2	0.2	0
Top 10	0.7	0.5	0.5	0.4	0.3	0.3	0
Top 15	0.8	0.466667	0.333333	0.266667	0.266667	0.533333	0.066667
Top 20	0.9	0.7	0.3	0.2	0.3	0.6	0.2

Table C.7. Average degree of overlap of all six requests for evaluation I

	human1	human2	human3	leaf	struct	path	random
Top 5	0.566667	0.6	0.633333	0.2	0.233333	0.266667	0
Top 10	0.533333	0.633333	0.616667	0.316667	0.366667	0.35	0
Top 15	0.577778	0.677778	0.666667	0.344444	0.366667	0.433333	0.066667
Top 20	0.691667	0.766667	0.65	0.358333	0.45	0.458333	0.2

Table C.8. Degree of overlap of request 1 for evaluation II

	Top5	Top10	Top15	Top20
human1	0.8	0.8	0.933333	0.85
human2	0.8	0.8	0.8	0.85
human3	0.8	0.7	0.8	0.9
case0	0.2	0.6	0.8	0.85
case1	0.2	0.5	0.4	0.4
case2	0.2	0.5	0.466667	0.35
case3	0.8	0.5	0.466667	0.4
case4	0.2	0.6	0.466667	0.35
case5	0.2	0.5	0.466667	0.35
case6	0.4	0.7	0.8	0.8
case7	0.8	0.6	0.466667	0.65
case8	0.8	0.5	0.466667	0.6
case9	0.8	0.9	0.8	0.75
case10	0.8	0.6	0.466667	0.65
case11	0.8	0.5	0.466667	0.5

Table C.9. Degree of overlap of request 2 for evaluation II

	Top5	Top10	Top15	Top20
human1	0.8	0.5	0.733333	0.75
human2	0.2	0.5	0.8	0.75
human3	0.6	0.6	0.466667	0.45
case0	0	0	0.333333	0.45
case1	0	0.3	0.666667	0.6
case2	0	0.3	0.6	0.55
case3	0.2	0.1	0.066667	0.3
case4	0.2	0.3	0.666667	0.6
case5	0.2	0.3	0.533333	0.6
case6	0	0.1	0.4	0.6
case7	0	0.3	0.666667	0.65
case8	0	0.3	0.6	0.6
case9	0.2	0.3	0.6	0.7
case10	0.2	0.3	0.666667	0.7
case11	0.2	0.3	0.6	0.65

Table C.10. Degree of overlap of request 3 for evaluation II

	Top5	Top10	Top15	Top20
human1	0.4	0.6	0.466667	0.7
human2	0.8	0.7	0.6	0.8
human3	0.4	0.6	0.866667	0.8
case0	0	0	0	0.15
case1	0.2	0.1	0.4	0.55
case2	0	0.1	0.133333	0.3
case3	0.4	0.2	0.2	0.3
case4	0.4	0.2	0.4	0.55
case5	0.4	0.3	0.333333	0.35
case6	0	0	0	0.35
case7	0	0.3	0.333333	0.45
case8	0.2	0.3	0.266667	0.45
case9	0.2	0.1	0.266667	0.55
case10	0.2	0.3	0.333333	0.45
case11	0.4	0.3	0.266667	0.45

Table C.11. Degree of overlap of request 4 for evaluation II

	Top5	Top10	Top15	Top20
human1	0.2	0.2	0.266667	0.5
human2	0.4	0.5	0.733333	0.85
human3	0.6	0.7	0.733333	0.65
case0	0	0	0	0.15
case1	0	0	0.2	0.5
case2	0.2	0.2	0.266667	0.35
case3	0.2	0.1	0.2	0.5
case4	0	0	0.133333	0.5
case5	0.2	0.2	0.266667	0.35
case6	0	0	0	0.3
case7	0	0.1	0.266667	0.4
case8	0.4	0.2	0.133333	0.3
case9	0.2	0.1	0.266667	0.5
case10	0	0.1	0.2	0.4
case11	0.4	0.2	0.133333	0.35

Table C.12. Degree of overlap of request 5 for evaluation II

	Top5	Top10	Top15	Top20
human1	0.8	0.4	0.266667	0.45
human2	0.6	0.8	0.666667	0.65
human3	0.8	0.6	0.8	0.8
case0	0	0	0.066667	0.2
case1	0.2	0.4	0.466667	0.45
case2	0.2	0.5	0.533333	0.6
case3	0	0.4	0.6	0.6
case4	0.2	0.4	0.533333	0.6
case5	0.2	0.4	0.466667	0.6
case6	0	0	0.066667	0.25
case7	0.2	0.2	0.266667	0.5
case8	0.2	0.2	0.266667	0.55
case9	0.2	0.3	0.266667	0.3
case10	0.2	0.3	0.266667	0.5
case11	0.2	0.3	0.266667	0.55

Table C.13. Degree of overlap of request 6 for evaluation II

	Top5	Top10	Top15	Top20
human1	0.4	0.7	0.8	0.9
human2	0.8	0.5	0.466667	0.7
human3	0.6	0.5	0.333333	0.3
case0	0	0.1	0.466667	0.45
case1	0.2	0.5	0.6	0.65
case2	0.2	0.3	0.466667	0.6
case3	0.2	0.1	0.133333	0.35
case4	0.2	0.5	0.6	0.7
case5	0.2	0.3	0.466667	0.55
case6	0	0.3	0.533333	0.55
case7	0.2	0.5	0.733333	0.75
case8	0.2	0.5	0.6	0.55
case9	0.2	0.3	0.533333	0.6
case10	0.2	0.5	0.733333	0.75
case11	0.2	0.5	0.6	0.5

Table C.14. Average degree of overlap of all six requests for evaluation II

	Top5	Top10	Top15	Top20	Avg. top5&10	Rank avg. top5&10
human1	0.566667	0.533333	0.577778	0.691667	0.55	
human2	0.6	0.633333	0.677778	0.766667	0.616667	
human3	0.633333	0.616667	0.666667	0.65	0.625	
case0	0.033333	0.116667	0.277778	0.375	0.075	12
case1	0.133333	0.3	0.455556	0.525	0.216667	10
case2	0.133333	0.316667	0.411111	0.458333	0.225	9
case3	0.3	0.233333	0.277778	0.408333	0.266667	6
case4	0.2	0.333333	0.466667	0.55	0.266667	6
case5	0.233333	0.333333	0.422222	0.466667	0.283333	5
case6	0.066667	0.183333	0.3	0.475	0.125	11
case7	0.2	0.333333	0.455556	0.566667	0.266667	6
case8	0.3	0.333333	0.388889	0.508333	0.316667	2
case9	0.3	0.333333	0.455556	0.566667	0.316667	2
case10	0.266667	0.35	0.444444	0.575	0.308334	4
case11	0.366667	0.35	0.388889	0.5	0.358334	1

Table C.15. Degree of overlap of request 1 for evaluation III

	human1	human2	human3	forward	bi-directional	random
Top5	0.8	0.8	0.8	0.2	0.2	0
Top10	0.8	0.8	0.7	0.5	0.4	0
Top15	0.933333	0.8	0.8	0.466667	0.466667	0.066667
Top20	0.85	0.85	0.9	0.35	0.35	0.2

Table C.16. Degree of overlap of request 2 for evaluation III

	human1	human2	human3	forward	bi-directional	random
Top5	0.8	0.2	0.6	0.2	0.2	0
Top10	0.5	0.5	0.6	0.3	0.3	0
Top15	0.733333	0.8	0.466667	0.533333	0.666667	0.066667
Top20	0.75	0.75	0.45	0.6	0.55	0.2

Table C.17. Degree of overlap of request 3 for evaluation III

	human1	human2	human3	forward	bi-directional	random
Top5	0.4	0.8	0.4	0.4	0.4	0
Top10	0.6	0.7	0.6	0.3	0.3	0
Top15	0.466667	0.6	0.866667	0.333333	0.266667	0.066667
Top20	0.7	0.8	0.8	0.35	0.4	0.2

Table C.18. Degree of overlap of request 4 for evaluation III

	human1	human2	human3	forward	bi-directional	random
Top5	0.2	0.4	0.6	0.2	0.2	0
Top10	0.2	0.5	0.7	0.2	0.2	0
Top15	0.266667	0.733333	0.733333	0.266667	0.333333	0.066667
Top20	0.5	0.85	0.65	0.35	0.35	0.2

Table C.19. Degree of overlap of request 5 for evaluation III

	human1	human2	human3	forward	bi-directional	random
Top5	0.8	0.6	0.8	0.2	0	0
Top10	0.4	0.8	0.6	0.4	0.4	0
Top15	0.266667	0.666667	0.8	0.466667	0.6	0.066667
Top20	0.45	0.65	0.8	0.6	0.7	0.2

Table C.20. Average degree of overlap of all requests for evaluation III

	human1	human2	human3	forward	bi-directional	random
Top5	0.6	0.56	0.64	0.24	0.2	0
Top10	0.5	0.66	0.64	0.34	0.32	0
Top15	0.533333	0.72	0.733333	0.413333	0.466667	0.066667
Top20	0.65	0.78	0.72	0.45	0.47	0.2

Table C.21. Time (sec) comparison of three data mediation algorithms

	leaf	structure	path
req1	21.092	44.336	136.376
req2	30.851	78.758	155.756
req3	30.42	202.691	210.086
req4	31.014	213.55	213.751
req5	19.922	122.149	180.889
req6	30	210.835	217.247
Avg.	27.2165	145.3865	185.6842

Table C.22. Time (sec) comparison of different annotation cases

	req1	req2	req3	req4	req5	req6	Avg.
case0	0.016	0	0	0.016	0	0	0.0064
case1	0.844	0.937	0.733	0.905	0.906	0.89	0.869167
case2	136.376	155.756	210.086	213.751	180.889	217.247	185.6842
case3	7.143	7.846	7.785	8.252	6.974	7.769	7.628167
case4	38.876	44.273	41.138	43.977	42.963	43.213	42.40667
case5	136.376	155.756	210.086	213.751	180.889	217.247	185.6842
case6	1.671	1.186	1.359	1.625	1.858	2.157	1.642667
case7	1.547	2.251	2.61	2.375	2.578	2.921	2.380333
case8	136.637	162.857	213.202	221.172	183.457	220.825	189.6917
case9	8.049	9.585	9.647	10.028	9.028	10.638	9.495833
case10	40.174	43.745	42.007	44.487	40.706	44.043	42.527
case11	161.365	190.981	242.862	254.291	208.74	251.548	218.2978

Table C.23. Time (sec) comparison of forward and bi-directional suggestion algorithms

	forward	bi-directional
req1	136.376	297.768
req2	155.756	653.924
req3	210.086	7359.175
req4	213.751	1175.48
req5	180.889	545.954
Avg.	179.3716	2006.46