APPLICATION OF SCHEMA MATCHING METHODS TO SEMANTIC WEB SERVICE DISCOVERY

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

FUNDA KARAGÖZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2006

Approval of the Graduate School of Natural and Applied Sciences

_____

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____

Assoc. Prof. Dr. Nihan K. Çiçekli
Supervisor

Examining Committee Members

Dr. Ayşenur Birtürk	(METU, CENG)	_____
Assoc. Prof. Dr. Nihan Kesim Çiçekli	(METU, CENG)	_____
Assoc. Dr. Ferda Nur Alpaslan	(METU, CENG)	_____
Assoc. Dr. Ali Hikmet Dogru	(METU, CENG)	_____
M.Sc. Yıldıray Kabak	(METU, SRDC)	_____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : **Funda Karagöz**

Signature: _____

ABSTRACT

APPLICATION OF SCHEMA MATCHING METHODS
TO
SEMANTIC WEB SERVICE DISCOVERY

Karagöz, Funda
M.Sc., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Nihan Kesim Çiçekli

September 2006, 114 pages

The Web turns out to be a collection of services that interoperate through the Internet. As the number of services increase, it is getting more and more diffucult for users to find, filter and integrate these services depending on their requirements. Automatic techniques are being developed to fulfill these tasks. The first step toward automatic composition is the discovery of services needed. UDDI which is one of the accepted web standards, provides a registry of web services. However representation capabilities of UDDI are insufficient to search for services on the basis of what they provide. Semantic web initiatives like OWL and OWL-S are promising for locating exact services based on their capabilities. In this thesis, a new semantic service discovery mechanism is implemented based on OWL-S service profiles. The service

profiles of an advertisement and a request are matched based on OWL ontologies describing them. In contrast to previous work on the subject, the ontologies of the advertisement and the request are not assumed to be same. In case they are different, schema matching algorithms are applied. Schema matching algorithms find the mappings between the given schema models. A hybrid combination of semantic, syntactic and structural schema matching algorithms are applied to match ontologies.

Keywords: Web Services, Semantic Web Service Discovery, Schema Matching, Automated Web Service Composition

ÖZ

ŞEMA EŞLEŞTİRME YÖNTEMLERİNİN ANLAMBİLİMSEL WEB HİZMETLERİNİN ARANMASINA UYGULANMASI

Karagöz, Funda

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Doç. Dr. Nihan Kesim Çiçekli

Eylül 2006, 114 sayfa

Web, Internet aracılığı ile birbirine bağlı olarak çalışan hizmetler derlemi olma yolunda ilerlemektedir. Web hizmetlerinin sayısı arttıkça, kullanıcılar için, ihtiyaçlarına cevap verecek servisleri bulmak, süzmek ve tümlemek gittikçe zorlaşmaktadır. Bu işlemleri yapmak üzere otomatik teknikler geliştirilmektedir. Servislerin otomatik olarak birleştirilmesinde ilk adım gerekli servislerin bulunmasıdır. Kabul görmüş Web standartlarından biri olan UDDI, Web hizmetlerinin yayınlanmasını ve bulunmasını sağlar. Ancak UDDI'nın gösterim olanakları hizmetlerin, ne sağladığına bağlı olarak aranabilmesi için yetersizdir. Anlambilimsel Web teşebbüslerinden OWL ve OWL-S servislerin sağladıklarına göre aranmasını vaadetmektedir. Bu tezde OWL-S hizmet belgilerine dayalı olarak yeni bir anlambilimsel hizmet arama mekanizması gerçekleştirilmiştir. Reklam edilen ve talep edilen servis belgileri, onları tanımlayan OWL ontolojilerine göre eşleştirilmiştir. Önceki

çalışmaların aksine reklam ve isteğin ontolojilerinin aynı olduğu varsayılmamış, ayrı oldukları durumda şema eşleme algoritmaları uygulanmıştır. Şema eşleme algoritmaları verilen şemalarda birbirine karşılık gelen elemanları bulmaktadır. Ontolojileri eşlemek için anlambilimsel, sözdizimsel ve yapısal şema eşleme algoritmalarının bir karışımı kullanılmıştır.

Anahtar Kelimeler: Web Hizmetleri, Anlambilimsel Web Hizmeti Arama, Şema Eşleştirme, Otomatik Web Hizmeti Birleştirme

ACKNOWLEDGEMENTS

Although only my name appears in the title, this is completely a team work. Some members of the team spent their time to enable me to work, some forwent the time which I had to spent for them. I am indebted to all the members of my family which constitude the team, especially, to my parents who were with me whenever I need, not only throughout this work, but also throughout all my life.

I am also indebted to my supervisor Assoc. Prof. Dr. Nihan Kesim Çiçekli for her guidance, patience and compassion. I could not succeed without her support and motivation.

I would like to thank Prof. Dr. Asuman Doğaç for providing me the test data. Finally, I also would like to thank to all my friends, especially to Aynur Badur and Asiye Saygı, for their encouragement throughout this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Problem Domain

Web is now more than a collection of pages. Web services promise to change Web from a database of static documents to an e-business marketplace[62], where partners colloborate and consumers are provided with the information and services they require. However, as the usage of Web services become widespread, new challenges arise. Web service composition problem, which is about integrating a number of services in order to produce the desired ouputs and effects, is one of them. Composition is needed when user request can not be satisfied with a single service.

Web service composition problem can be decomposed into three sub-problems: composition planning, Web service discovery and composition execution. The common opinion about Web service composition is that it is getting more and more diffucult to be carried out manually [2, 64, 43, 69, 63]. One of the reasons of this, is the upward trend in the number of services available on Web. Finding the exact services will be very time consuming. The second is the dynamic nature of the Web. The Web services can be out of service or can be updated or new services can be created better fulfilling the desired task. The composer must always be aware of the changes at run time and make decisions based on up to date information. Therefore automated composition techniques which exploit AI or similar technologies have been developed [2, 43, 69, 54, 42, 40].

These proposals mostly focus on the planning phase. The Web service discovery phase is often neglected or an existing method is adopted. However, automated Web service discovery is the first step in the automation of the composition. Three key requirements of the Web service discovery are: a common way to describe the capabilities of the provided and requested services, a registry to advertise the services and discovery methods to find the services that best satisfy the user request. Currently, the Web Services Description Language (WSDL) [10], which is for the specification and Universal Description, Discovery and Integration (UDDI) [11], which is for advertising and discovery, are industry standards for Web services. These standards, however, are mostly syntactic, poor in capturing semantics of the services and their representation capabilities are insufficient for powerful automation. In order to automate tasks, richer semantic specifications are needed.

Semantic Web is under way, to turn Web into a medium where data can be shared, understood and processed by automated tools. This requires converting HTML encoded information into a machine processible form, namely Resource Description Framework (RDF) [37] and put semantics on it via an ontology description language, namely Ontology Web Language (OWL) [41]. Ontology Web Language for Services(OWL-S) [38], which is an OWL ontology, is an extension of Semantic Web for web services. OWL-S, which is capable of addressing the deficiency of WSDL in automation process, provides a rich set of constructs for advertising and modelling services. On the other hand it exploits WSDL constructs for accessing services. Web Service Modelling Ontology (WSMO) [58] is another initiative which exploits semantic Web constructs for describing Web services.

Semantic Web service discovery methods operate on Web services whose capabilities are semantically described. Two ways of sematically describing the capabilities of a service are: using ontologies to describe the functionality of the service and using ontologies to describe the state transformation produced by the service. The semantic discovery methods proposed in [33, 18, 30] exploit the first way. The OWL-S MatchMaker in [52, 53, 62] adopts the second way. It exploits a

subset of the tranformations produced by the service. The overall similarity of two descriptions depends on the similarity of the their input/output parameters. A discrete scale is defined to rank the similarity between two parameters. The scale depends on the semantic relation between the concepts that describe the parameters. The discovery methods in [60] and [65] use both methods during discovery.

1.2. Schema Matching Problem

With the advances in information technology the user is now capable of accessing different data sources and desires the integration of them for easy processing. Schema matching is the first step of integrating different data sources. Since these data sources are designed independently, most often to be self contained, there might be different kinds of heterogeneities between them like naming conflicts, representation conflicts and structural conflicts. Schema matching is the process of finding a set of *direct matches* each of which binds a source schema element to a target schema element, if they are semantically equivalent despite the heterogeneities between them. Manually finding these matches is time consuming and error-prone.

Many methods are proposed for automating schema matching process. These can be classified as individual and combined matchers. Individual matchers use only one criterion to find matches. These criterion may either depend on the schema or the content of the data source. Schema based solutions can be classified into two: element level or structural matchers, where the former matchers compare the schema elements syntactically or semantically and the latter matchers compare the structures in a heuristic, formal or constraint-based fashion. Individual matchers most often exploit external resources like thesaurus, domain ontologies or domain specific rules.

Combined matchers exploit different number of individual matchers and combine their results in a fixed or dynamic fashion.

Exploiting only one criterion during matching is not sufficient for achieving satisfying results. In most of the recent studies on schema matching, hybrid or

composite matchers are used and individual matchers are exploited as building structures. Cupid [34], S-Match system [20], iMAP[14] and GLUE[16] are examples of hybrid matchers where as COMA [15], COMA++ [1], Protoplasm [3] and H-Match system [7] are composite matchers.

Ontology matching is a specialized version of schema matching where input schemas are two ontologies which describe similar or overlapping domains, but using different languages, conceptualizations, modelling styles and terms. Most of the schema matching systems are capable of matching ontologies, besides relational and XML schemas.

1.3. Scope of the Thesis

Semantic discovery methods, introduced in Section 1.1, exploit ontologies for providing shared understanding. The problem with their usage is that they assume the advertisements and requests share the same ontologies. OWL-S MatchMaker, for instance, determines the match degree between two inputs or two outputs depending on the subsumption relation between the concepts that describe them. However, due to the de-centralized nature of the Web, most probably there will be an explosion in the number of ontologies, most of them describing similar or overlapping domains, but using different languages, conceptualizations, modelling styles and terms. Consider the Activity ontology and Recreation ontology in Figure 1.1. Suppose there exists an advertisement with output concept “Sports” in repository and a discovery request comes with output concept “SportsContest”. MatchMaker can not match them since no explicit semantic relation is defined between them or can be inferred. However, they model the same real world entities and most probably the advertisement satisfies the request.

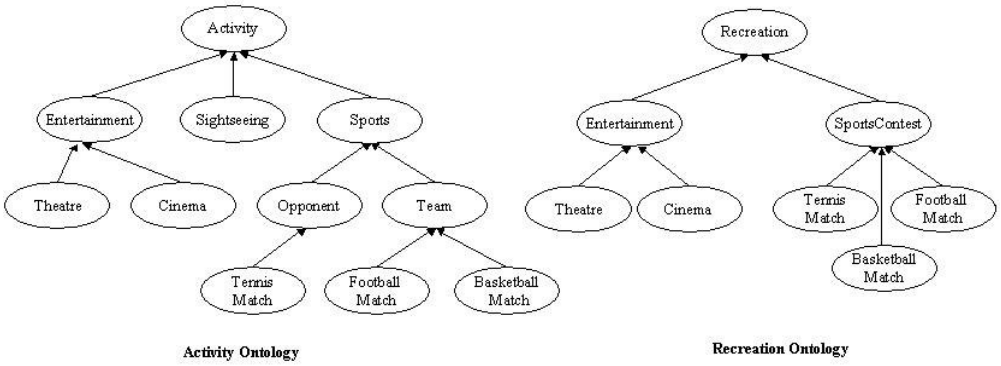

Figure 1. 1 Two ontlogies modelling similar real world entities

The aim of this thesis is to extend semantic discovery methods in a way to handle the stated ontology differences. This increases the number of related items retrieved during discovery.

The OWL-S Matchmaker's matching approach is selected for extension since it is exploited by several automated Web service composition systems and provides integration with UDDI which is a widely accepted industry standard. The thesis extends this approach in two ways:

- The similarity scale is extented to rank the similarity of not only the concepts which have semantic relation, but also the concepts which have semantic similarity between them.
- A schema matching component is added to compare two concepts and calculate semantic similarity between them using the schema matching algorithms. Different individual and hybrid matchers are adapted to the problem domain and implemented to compare different aspects of the concepts.

The new matching approach is named as **S**emantic **S**ervice Discovery with **S**chema **M**atcher (SSSM).

1.4. Organization Of the Thesis

The organization of the thesis is as follows: In Chapter 2, the problem domain is described in detail. The vision of Semantic Web and its components are explained first. Following the definition of Web services, Web service composition problem and its sub problems, Web service discovery and composition planning are presented. Afterwards, the proposals for the automated Web service composition problem and the contribution of semantic Web to them are discussed. In Chapter 3, schema matching problem and the proposals to the solution of the problem are discussed. Chapter 4, introduces the new matching approach, SSSM, which exploits schema matching algorithms to improve Web service discovery results. In Chapter 5, the results of the tests for system evaluation are discussed. Finally, Chapter 6 presents the conclusions and future work.

CHAPTER 2

BACKGROUND INFORMATION ON WEB SERVICES

Web services, due to their promising nature, gained much attention from both industry and academia. Composition of services is one of the major topics of the domain. The composition problem is composed of three sub-problems: service discovery, planning, and execution. Service discovery requires languages to describe services and methods operating on these descriptions. WSDL and OWL-S are proposals for describing services. OWL-S is an application of Semantic Web technologies in Web Services domain. Many discovery methods are proposed operating on these languages. UDDI, which is a widely accepted industry standard for discovery, and semantic discovery methods deserve more attention among the others. The work on planning problem proposes manual, semi-automated and automated solutions. Automated solutions exploit AI technologies.

This chapter is structured as follows: Section 2.1 gives detailed description about Semantic Web and its components. Section 2.2 explains Web services and parties involved in Web service processes. Section 2.3 defines Web service discovery and presents current proposals on describing and discovering services. Finally, Section 2.4 focuses on the composition problem and solution proposals for planning and execution phases.

2.1. Semantic Web

The Web can reach its full potential only if it becomes a place where data can be shared and processed by machines as well as by people [23]. One of the major obstacles to this is the structure of the current Web, which is primarily based on documents written in Hyper Text Markup Language (HTML). HTML is a language that is useful for describing, with an emphasis on visual presentation. It has no ability to structure the documents and has no capability to provide the machine with the semantics of the contents. For example as far as the machine is concerned, the sentence "Turkey' s capital is Ankara." is just a plain text. It has neither a particular structure nor a meaning for the machine.

The Semantic Web, which promises to address these shortcomings, is not a separate Web, but an extension of the current one. In Semantic Web, information is given well-defined meaning, better enabling computers and people to work in cooperation. The answer of the question "how to give meaning?" is "by using *ontologies*". An ontolology defines the terms used to describe and represent an area of knowledge using classes, properties of the classes and relationships between classes.

The components of the semantic web are Resource Description Framework (RDF), RDF Schema language and the Web Ontology Language (OWL). All these components are built on the foundations of Universal Resource Identifiers (URIs), Extensible Markup Language (XML) and XML namespaces.

2.1.1. RDF

The Resource Description Framework (RDF) is a language for representing information about resources in the World Wide Web (WWW) [37]. RDF defines a *resource* as any object that is uniquely identifiable by a Uniform Resource Identifier (URI) [45]. Resources have properties and properties have values. In RDF, everything about a resource is a statement, in the form of subject, predicate, object

expression which is called a triple. Subject is the resource, predicate is a property of the resource and object is the value of that property. The object can either be a literal or another resource. The description of the sample sentence in RDF is given in figure 2.1.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<rdf:RDF
  xmlns:rdf=     " http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:prp=     " http://www.thesis.org/example-schema/">

<rdf:Description rdf:about="http://www.thesis.org/example/Turkey">
            <rdf:type rdf:resource="http://www.thesis.org/example/Country"/>
            <prp:hasCapital>Ankara</prp:hasCapital>
</rdf:Description>
```

Figure 2. 1 RDF representation of the sample sentence

2.1.2. RDF Schema

RDF Schema (RDFS)[5] puts some semantics on RDF. It is a vocabulary for describing properties and classes of RDF resources, with a semantics for generalization hierarchies of such properties and classes. It provides mechanisms for describing groups of related resources and the relationships between these resources.

These make RDFS a simple ontology language, however, in order to provide machines to perform useful reasoning on these documents, richer semantics are needed.

2.1.3. OWL

OWL [41], as the first layer on RDF in Semantic Web stack, has been designed to meet the need for a Web Ontology Language that can formally describe the meaning of the terminology used in web documents.

Three species of OWL are *OWL Lite*, *OWL DL* and *OWL full*. OWL Lite provides constructs for classification hierarchy and simple constraints. OWL DL, on the other hand supports all OWL constructs with some restrictions to provide maximum expresiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL Full provides maximum expresiveness without restrictions with no computational guarantees.

OWL Features

In order to illustrate the usage of OWL features, some examples drawn from a travel ontology [66] are used. Part of the concept hierarchy from the ontology is given in figure 2.2. In OWL ontologies, besides OWL features, some features from RDF and RDFS are also exploited. These features are prefixed by rdf and rdfs, respectively.

In an OWL ontology, concepts can be declared using *"owl:Class"* and can be organized in a specification hierarchy using *"rdfs:subClassOf*".Synonymous classes can be stated using *"owl:equivalentClass*", and classes may be stated to be disjoint from each other using *"owl:disjointWith*". The following OWL statements define class "Hotel" as a subclass of "Accommodation", with no common instances with "BedAndBreakfast" and "Campground".

```
<owl:Class rdf:ID="Hotel">
    <owl:disjointWith>
        <owl:Class rdf:about="#BedAndBreakfast"/>
    </owl:disjointWith>
```

<owl:disjointWith>

<owl:Class rdf:about="#Campground"/>

</owl:disjointWith>

<rdfs:subClassOf rdf:resource="#Accommodation"/>

</owl:Class>

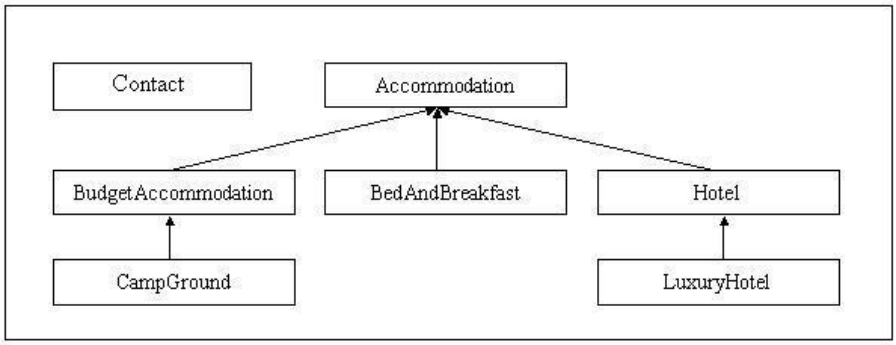

Figure 2.2 Part of the concept hierarchy of travel ontology

An instance of a class can be declared using *"owl:Individual"*. Two individuals can be stated to be same using *"owl:sameAs"* or different using *"owl:differentFrom"*. It is also possible to state that a number of individuals are mutually distinct in one "*owl:AllDifferent"* statement. The following OWL statement defines "FourSeasons" as a LuxuryHotel.

<LuxuryHotel rdf:ID="FourSeasons"/>

For declaring properties, *"owl:ObjectProperty"* is used for those that can have only instances of classes as their values and *"owl:DatatypeProperty"* is used for those that can have only instances of datatypes as their values. Property hierarchies

can be constructed using “rdfs:subPropertyOf” and synonymous properties can be stated using *“owl:equivalentProperty”*. To restrict the indivuduals which a property can be applied to and may have as its value, *“rdfs:domain”* and *“rdfs:range”* are used, respectively. The following OWL statement defines a proper-ty named “hasRating” which can be applied to individuals of class “Accomodation” and can have values only from individuals of “AccommodationRating”.

```
<owl:ObjectProperty rdf:ID="hasRating">
        <rdfs:range rdf:resource="#AccommodationRating"/>
        <rdfs:domain rdf:resource="#Accommodation"/>
</owl:ObjectProperty>
```

A property may be stated to be *transitive (owl:TransitiveProperty)*, *symmetric (owl:SymmetricProperty)* and *functional (owl:FunctionalProperty*). If a property is a FunctionalProperty, then it has no more than one value for each individual. One property may be stated to be the inverse of another property using *“owl:inverseOf”*. The following OWL statement defines a datatype property “hasEMail” which is a functional property, can be applied to individuals of “Contact” and values of this property can only be of type String.

```
<owl:DatatypeProperty rdf:ID="hasEMail">
        <rdfs:domain rdf:resource="#Contact"/>
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
        <rdf:type
                rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/
        >
</owl:DatatypeProperty>
```

Restrictions may be placed on how properties can be used by instances of a class. These can be type or cardinality restrictions which are used within the context of an *"owl:Restriction*". Type restrictions *(owl:allValuesFrom, owl:someValues-From, owl:hasValue)* limit which values can be used, while cardinality restrictions*("owl:minCardinality", "owl:maxCardinality", "owl:cardinality")* limit how many values can be used. The following OWL statement declares "Campground" as a class whose individuals can have only "OneStarRating" as the value of their "hasRating" property.

```
<owl:Class rdf:ID="Campground">
        <rdfs:subClassOf>
                <owl:Restriction>
                        <owl:hasValue rdf:resource="#OneStarRating"/>
                        <owl:onProperty>
                                <owl:ObjectProperty rdf:about="#hasRating"/>
                         </owl:onProperty>
                </owl:Restriction>
        </rdfs:subClassOf>
 </owl:Class>
```

It is possible to form classes using basic set operations, namely union, intersection and complement using *"owl:unionOf", "owl:intersectionOf*", and *"owl:complementOf*", respectively. The following OWL statements define "FamilyDestination" as a destination with at least one accommodation.

```
<owl:Class rdf:ID="FamilyDestination">
   <owl:equivalentClass>
      <owl:Class>
         <owl:intersectionOf rdf:parseType="Collection">
```

```
        <owl:Class rdf:about="#Destination"/>
            <owl:Restriction>
                <owl:onProperty>
                    <owl:ObjectProperty rdf:about="#hasAccommodation"/>
                </owl:onProperty>
                <owl:minCardinality rdf:datatype=
                    "http://www.w3.org/2001/XMLSchema#int">1</owl:minCardin
                ality>
            </owl:Restriction>
            …
        </owl:intersectionOf>
        </owl:Class>
    </owl:equivalentClass>
</owl:Class>
```

Additionally, Classes can be described by enumeration of the individuals that make up the class using “owl:oneOf”. The following OWL statement defines “AccommodationRating” which consist of exactly three individuals.

```
<owl:Class rdf:ID="AccommodationRating">
    <owl:equivalentClass>
        <owl:Class>
            <owl:oneOf rdf:parseType="Collection">
                <AccommodationRating
            rdf:about="#OneStarRating"/>
                <AccommodationRating
            rdf:about="#TwoStarRating"/>
                <AccommodationRating
            rdf:about="#ThreeStarRating"/>
```

```
                </owl:oneOf>
            </owl:Class>
        </owl:equivalentClass>
</owl:Class>
```

Detailed explanation about the features of OWL can be found in [41].

2.2. Web Services

Web services are defined as modular, self-describing, self-contained applications that are accessible over the Internet [60]. They are provided by *Service Providers.* They perform an action on behalf of the *Service Requester* depending on the inputs given by the requester and produces an output. Although it seems that only two parties are involved in this interaction, there is a need for a third party, since it is difficult for a requester to find a suitable service in a medium like Web. The third party is the service agency which provides registration and discovery of services. The interaction between the parties are given in Figure 2.3.

In the first phase, the provider developes the service, describes it using a common way and registers it to a service agency. In the second phase the requester sends a query, which describes the needed service, to the agency. Using discovery methods the agency finds services conforming to user request and sends them in the third phase. Requester selects one among them and the interaction between requester and provider starts.

The provided service can be *simple* (or *primitive*) like a service that returns a postal code or returns the latitute-longitude of a given an address. Simple services does not require much interaction. Requester provides the inputs and depending on the inputs given, provider returns the output. Alternatively, the service can be *complex*, composed of primitive processes which requires more conversation. Besides providing information, the user can make choices and the primitive services may be executed conditionally depending on the choices of the user .

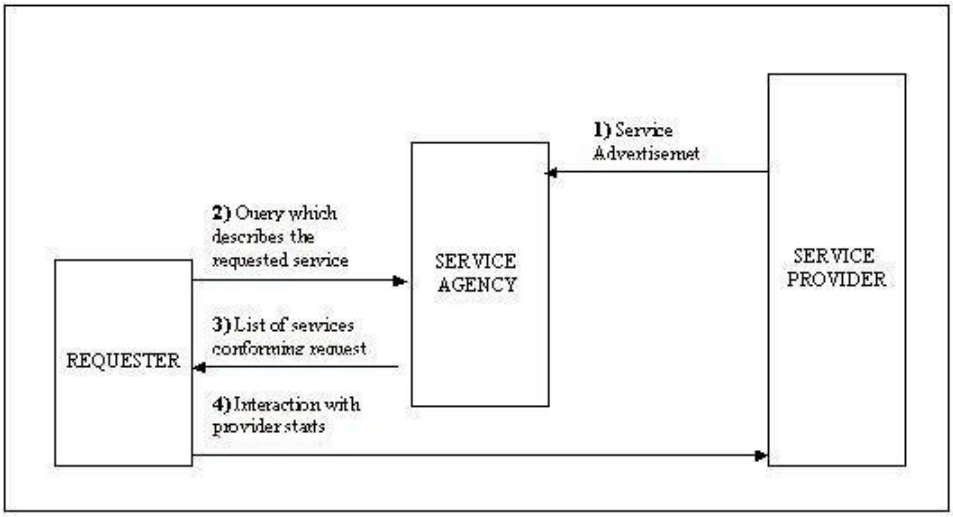

Figure 2. 3 Interaction between parties involved in the usage of a Web service

2.3. Web Service Discovery

A service must be discovered before it can be used [6]. However in a medium like Web, services are distributed and heterogenous. They are offered by different providers, developed using different technologies and executed under different operating systems. Moreover similar services might have different capabilities. How these services can be found and which one best conforms to user request are questions that are answered by service discovery methods. Service discovery can simply be defined as *the act of finding a service that conforms to user request*. Three requirements for discovery are:

- A language to describe the capabilities of provided services and requests
- Advertisement of the services in a registry
- Discovery methods to locate the services that best satisfy user request among others.

2.3.1. Web Service Description Languages

Two common ways to describe services are Web Services Description Language (WSDL) [9] and Ontology Web Language for Services (OWL-S)[38]. WSDL, which provides a syntactic description for services, is the widely accepted industry standard. OWL-S, on the other hand is the most formal result of applying Semantic Web technologies to Web services domain. It provides a semantic description for services, which reveals out the capabilities of the services better.

2.3.1.1. WSDL

WSDL is a specification which defines how to describe web services in a common XML grammar [27]. It is regulated by a group of companies and its latest version [9] is a W3C candidate recommendation. WSDL describes four critical pieces of data:

- Datatype information for all message requests and message responses.
- Interface information describing all publicly available functions.
- Binding information about the transport protocol to be used.
- Address information for locating the specified service.

In a WSDL specification, two types of components exist: *type system components* and *WSDL 2.0 components*. The type system components define the data types used by the exchanged messages. WSDL uses W3C XML Schema as its preferred schema language.

WSDL 2.0 components are interfaces, bindings and services. In Figure 2.4 a view of conceptual WSDL component model is given. WSDL enables to separate the components that answer the questions what, how and where for the sake of reusability.

Interface answers the question “what ?”. It defines the abstract interface of a Web service as a set of abstract *operations.* Each operation represents a simple interaction between the client and the service, specifies the *types of messages* that the

service can send or receive as part of that operation and also specifies a *message exchange pattern.* A message exchange pattern defines the sequence and cardinality of the abstract messages in an operation.

Binding answers the question "how ?". It specifies a specific set of encoding and transport protocols such as SOAP, HTTP and MIME, that may be used to communicate with an implementation of a particular WSDL interface. It specifies these details for every operation defined in the interface.

Finally, the *service* answers the question "where ?" by pointing to a concrete implementation. A *service endpoint* associates network address with a binding and a *service* groups the endpoints that implement a common interface.

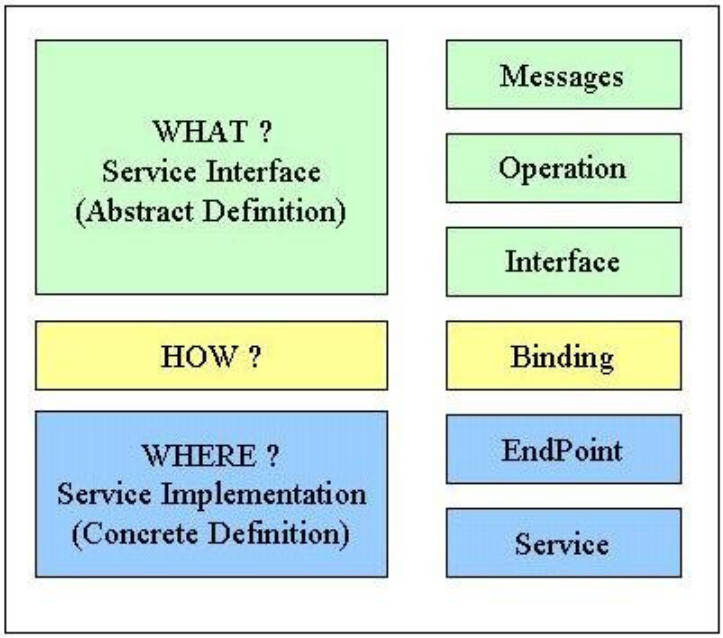

Figure 2.4 Conceptual view of WSDL Component Model

A WSDL interface may be communicated with different sets of transport protocols and encodings and may be implemented by different services. This is achieved by defining multiple bindings and services for a single interface. A WSDL binding can also be implemented by different services. This makes bindings abstract since they do not refer to a concrete implementation.

An example WSDL document from [10] is converted to WSDL 2.0 and is given in APPENDIX A. In the example a simple service providing stock quotes is defined. The service supports a single operation called GetLastTradePrice, which is deployed using the SOAP 1.1 protocol over HTTP. The request takes a ticker symbol of type string, and returns the price as a float.

2.3.1.2. OWL-S

Ontology Web Language for services (OWL-S) [38] is an OWL ontology developed by OWL-S coalition for describing services. It contains four main classes: the first is *Service* which is the organizational point of reference for declaring Web services and the rest are *ServiceProfile*, *ServiceModel* and *Service Grounding* which are used to describe "*what* the service does", "how the service works" and "how to use the service", respectively. The Service is associated to its profile, model and grounding with the properties "*presents*", "*describedBy*" and "*supports*". The general view of the upper level of the OWL-S ontology is given in Figure 2.5.

ServiceProfile

ServiceProfile provides necessary information for a service requester to discover the service. The profile is related to its service with the property "*presentedBy*" and related to its service model with the property "*hasModel*". The properties of the profile are grouped into three categories:

- **Properties intended for human consumption:** The textual description of the provided service, contact information and the service name are the properties in this category.

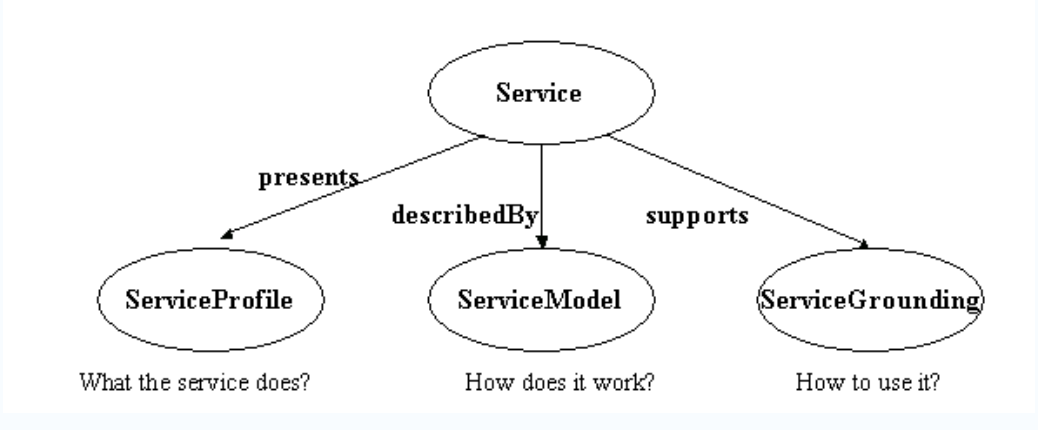

Figure 2. 5 Upper level of OWL-S ontology

- **Properties for functional representation:** OWL-S profile represents two aspects of the functionality of the service: the information transformation (represented with inputs and outputs) and the state change occurred with the execution of the service (represented by preconditions and effects). For example in order to complete online shopping, the inputs required for the service are personal information of the customer (name, address, etc.), a credit card number and its expiration date. The precondition of the service, for a successful execution, is a valid credit card which is not overdrawn. The output of the service is the receipt and the results are the transfer of the items to customer's address and charging the credit card account. Most often, inputs, outputs, preconditions and effects (IOPEs) are defined in service model and the profile just contains references to them. The IOPEs are defined using properties *"hasInput"*, *"hasOutput*", "*hasPrecondition*" and "*hasResult*", respectively.
- **Properties of the profile:** The non-functional properties of the service are classified under this category. The quality guaranties of the service, possible classification of the service (via "*hasCategory*") and additional parameters

(via *"hasParameter"*) are specified using these properties. The range of *"hasCategory"* is ServiceCategory which describes categories of service on the basis of some classification possibly outside OWL.

The profile provides these properties, but does not mandate any representation of the service. It is possible to create specialized representations of services by subclassing the ServiceProfile class.

ServiceModel

The Service Model gives a detailed description of how the service works. Each service is mapped to a Process. Each process has a set of IOPEs. Three types of processes are *Simple*, *Atomic* and *Composite*:

- An *Atomic Process* represents a primitive service. It takes input parameters, executes and returns the output in a single step. Each atomic process requires a grounding which will be described described in the next section.
- A *Composite Process* represents a complex service. It is composed of a set of atomic processes which are connected to each other with Control Constructs. Control Constructs are Split, Split-Join, Sequence, Choice, If-Then-Else, Iterate, Repeat-While, Repeat-Until and Any-Order which are subclasses of ControlConstruct. The detailed descriptions of these constructs can be found in [38].
- *Simple processes* are abstractions of atomic and composite processes for planning purposes. They have no grounding associated.

The data transformation produced by the process is specified using properties *"hasInput"* and *"hasOutput"* whose ranges are Input and ConditionalOutput classes. Both classes have *"parameterType"* property which shows the range of the values the parameter can take. An input or output may have an XML data type or an OWL ontology concept as the parameter type. For example in the online shopping service scenerio, the credit card number may be of type string

where personal information of the user may be modeled by concept Person and the second input may be of type Person.

Besides data transformation the process also produces a state transition. The state transition is modeled using properties *"hasPrecondition"* and "*hasEffect*" whose ranges are Precondition and ConditionalEffect, respectively.

The effects and outputs of the service may not be the same for every execution. An output may be produced depending on a condition and an effect may also occur depending on a condition. For example in the online shopping scenerio, the receipt may be produced and the credit card may be charged if the credit card is valid and not overdrawn. ConditionalOutput and ConditionalEffect classes represent this requirement using conditions.

OWL-S does not provide any construct for expressing conditions, rather supports the usage of a rule language and leaves the decision of which to use to the modeler of the process. Semantic Web Rules Language (SWRL)[24] which is under development of W3C is an option for this purpose. SWRL is a combination of OWL Lite and OWL DL with Unary/Binary Datalog RuleML which is a subLanguage of Rule Markup Language.

ServiceGrounding

Service Profile and Service Model are abstract definitions and do not provide details about how to use a concrete service. Grounding, on the other hand, deals with concrete level of specification and describes the access details like the protocol to be used, message formats, addressing, serialization and transport.

WSDL's concept of binding which is described in previous section seems consistent with the OWL-S concept of grounding. The developers of OWL-S claim in [38] that, complementary usage of these two languages has two advantages. The first is exploiting powerful description capabilities of OWL-S process model and OWL class typing mechanism. The second is the opportunity to reuse the extensive

work done in WSDL and its underlying technologies like SOAP, HTTP and message exchange softwares.

Due to the stated advantages, instead of creating a new specification for this purpose, WSDL is exploited by binding atomic processes to WSDL operations. Table 2.1 from [2] gives the mapping between two languages. WSDLGrounding class serves to bind processes to their associated WSDL descriptions in OWL-S.

Table 2. 1 WSDL and OWL-S correspondences

OWL-S	WSDL
Atomic Process	Operation
Atomic Process with input then output	Request-response operation
Atomic Process with input only	One-way operation
Atomic Process with output only	Notification operation
Atomic Process with output then input	Solicit-response operation
Atomic process inputs	Operation input message
Atomic Process outputs	Operation output message
Atomic process input and output types	Abstract types

2.3.2. Web Service Discovery Methods

Current Web service discovery methods are classified in [31] into four groups: keyword-based discovery, table-based discovery, concept-based discovery and deductive discovery. This classification is modified in this work as keyword-based discovery, table-based discovery, semantic discovery and deductive discovery. Semantic discovery actually covers a subset of table-based discovery and whole set of concept-based discovery in the first classification, so it seems better to categorize

them under semantic discovery. These methods will be explained after introducing two metrics for evaluating their performance.

Web Service Discovery Metrics

Service discovery is closely related with information retrieval area. The information retrieval paradigm is about a user (physical or not) having a need for information, and a set of information objects from which this need has to be satisfied [6]. In information retrieval two metrics are used to evaluate the quality of the retrieval methods:

- **Recall:** The value obtained by dividing the number of relevant items retrieved by the total number of relevant items in the collection. Recall reaches to its highest value when all the relevant items are retrieved.
- **Precision:** The value obtained by dividing the number of related items retrieved by the total number of items retrieved. Precision reaches its highest value when all the items retrieved are relevant with user request.

The goal in information retrieval is similar to the one in service discovery. Therefore most often recall and precision are used for evaluating the quality of the retrieval methods [31, 6, 68]. These metrics will be used also in this work to evaluate the method's performance.

Keyword-Based Discovery

In keyword-based approaches, discovery is based on the keywords that are extracted from the request. These methods most often have low recall and precision values since the keyword's homonym may be contained in an irrelevant request or a synonym keyword may be contained in a relevant request. Furthermore these methods lack capturing the semantics of the request. In [31] UDDI, the proposals of Salton, Magnini and Prieto-Diaz are classified under this category.

Table-Based Discovery

Table-based approaches define the properties of the request and the advertisements in the form of attribute value pairs and match these pairs (e.g. input=real,output=string). Table-based discovery methods capture some semantics, however the problem of synonyms and homonyms still exists. In [31] UDDI/WSDL and the proposal of Richard are classified under this category.

Semantic Discovery

These methods exploit Semantic Web constructs somehow to describe and discover Web services. These methods will be explained in detail in section 2.3.4.

Deductive Discovery

In these methods service semantics are expressed formally using logic. Discovery is done by deducing which advertisements achieves the functionality described in the request. Although these methods have the highest recall and precision values, due to the diffuculty of formally describe the services and the complexity of matching algorithms, these methods are not suitable for practical usage. [29] is an example to this class of methods.

Figure 2.6 shows the capabilities of methods in a recall/precision framework.

2.3.3. UDDI

The Universal Description, Discovery, and Integration (UDDI) [11] specification is a widely accepted industry standard that provides a standardized way for publishing and discovering services over the Internet. UDDI discovery mechanisms can be classified as both keyword and table-based. A UDDI server provides:

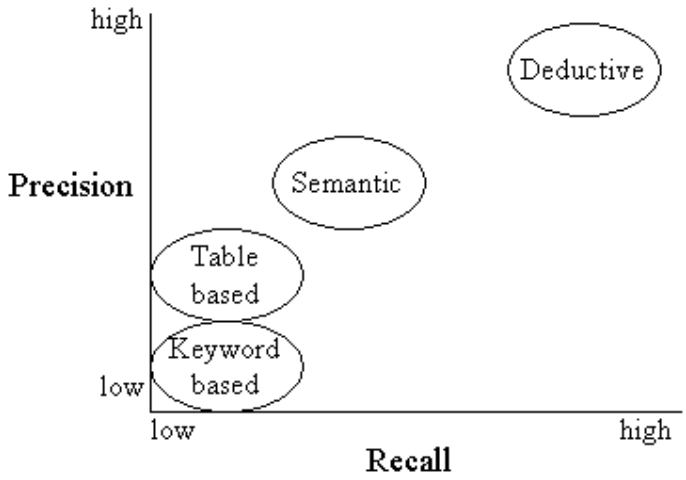

Figure 2.6 Service discovery methods

- **White Pages** where businesses are listed with their contacts, addresses, telephone numbers.
- **Yellow Pages** where businesses register their services under different categories.
- **Green Pages** where technical details of offered services are given.

In UDDI services offered can be anything from a telephone number to a SOAP endpoint for a Web service.

Since UDDI is a sort of database, it has a data model. The core elements of the model are *businessEntity*, *businessService*, *bindingTemplate and tModel*. The UML representation of these elements with their relationships is given in Figure 2.7 from [32].

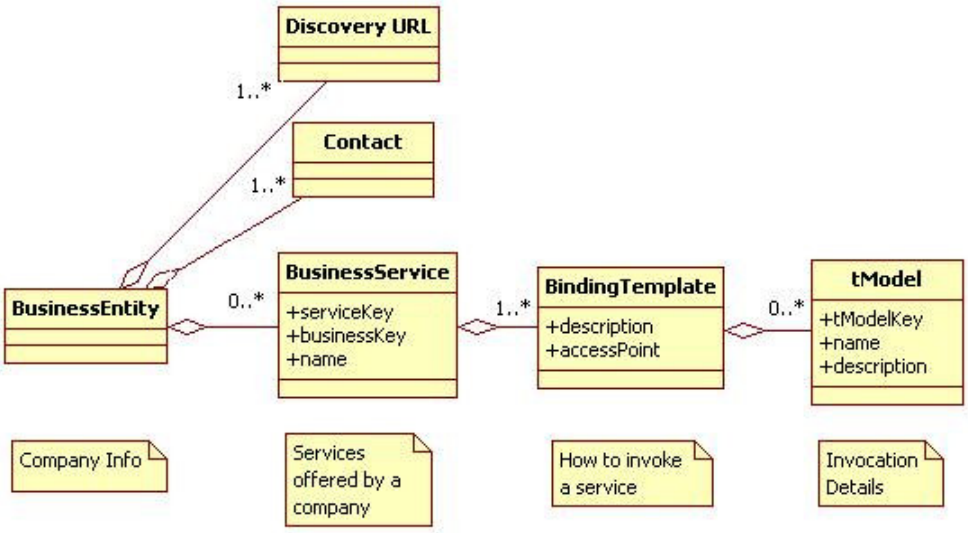

Figure 2. 7 UML representation of UDDI data elements

businessEntity

A physical company is represented by a businessEntity. Each business entity is identified by a unique key and includes information about the company such as name, address, contacts, categories under which the company can be listed, identifiers that the company has, etc.

businessService

It represents a business service offered by a company. It includes the business key of the owning company, name of the service, a text description, binding templates and the categories the service belongs to.

bindingTemplate

It represents the way the business service is accessed. It includes the service key to which it is related, an access point (which may be a URL, phone number or

mail address), and an optional set of tModelInstanceInfo structure which contains a tModelKey and a text description about the usage purpose of tModelKey in the overall service description.

tModel

The service descriptions are required to be meaningful enough for useful searches and comprehensive enough to learn how to interact with them. This can be accomplished by marking a description with information that designates how it behaves, what conventions it follows, or what specifications or standards the service is compliant with. Providing the ability to describe compliance with a *specification*, *concept*, or even a *shared design* is one of the roles that the tModel structure fills [57]. Two main uses of tModels are:

- Representing a technical specification such as OWL-S specification.
- Defining an organizational identity or a classification system such as NAICS (North American Industry Classification System) which is an industry code taxonomy or UNSPC (United Nations Standard Product and Services Classification) which is a product and service category code taxonomy.

All four of the data elements have relationships with tModels, but the meaning of the tModel differs in each context. The *categoryBag* allows businessEntity, businessService and tModels to be categorized according to any of several available taxonomy based classification schemes. The *identifierBag* allows businessEntity or tModel structures to include information about common forms of identification such as tax identifiers. Both are lists of *keyedReferences*. A keyedReference, which is a triple in the form (tModelKey, keyName, keyValue), provides a name-value pair within the scope of the taxonomy or identifier system referred to by its tModelKey. In the case of bindingTemplates, tModels are used to refer to technical specifications. Figure 2.8 shows how bindingTemplates are related to tModels in detail.

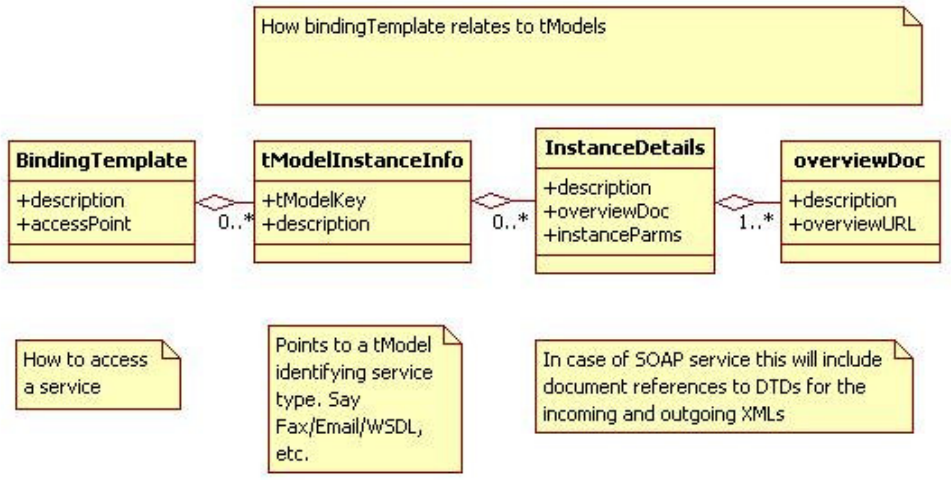

Figure 2. 8 Relationship between bindingTemplates and tModels

Usage of WSDL with UDDI

WSDL provides a specification for a Web service. However, this specification must be advertised somehow in a UDDI compliant server in order to be discovered and used. The usage of WSDL with UDDI is best described in [12] and summarized in Figure 2.9.

When advertising a WSDL service specification via UDDI, abstract parts of the specification are converted to UDDI tModels. Thus, a tModel is created for both interface and binding descriptions of the service. These tModels are classified as WSDL interface and WSDL binding in categoryBags using keyedReferences. This implies that the overviewDocs within the tModels contain a URL pointing to the WSDL document describing the interface and binding of the Web service. The binding tModel is related to the interface tModel via a keyedReference. The WSDL services are converted to businessServices and classified as WSDL services. Each endPoint of the service is converted to the bindingTemplate of the corresponding

businessService. The bindingTemplates provide access to the technical information required to consume the service.

In order to describe a WSDL specification in terms of UDDI data elements a set of predefined UDDI tModels are used to represent WSDL metadata and relationships. A detailed description of the models are given in [12]. In APPENDIX B, necessary UDDI data elements needed to advertise the sample WSDL specification given in section 2.1.1, are listed. In Listing 1, a tModel is defined to represent the WSDL interface of the StockQuoteService. Listing 2 gives the tModel created for the StockQuoteBinding. In Listing 3, businessEntity for company Cape Clear Software, which owns the StockQuoteService, is given. The company is categorized as *"Software Publishers*" according to NAICS and *"US"* according to geography (via ISO 3166). Finally, Listing 4 documents the necessary UDDI businessService entry for StockQuoteService.

Search Capabilities of UDDI

UDDI inquiry API [11] allows for searching a Web service based on a keyword, category, tModel and provider. Also it allows for searching the providers based on a keyword, a category or a tModel and tModels based on a keyword or a category. So when WSDL is used with UDDI as explained above, it is possible to :

- Search for services that are described by a WSDL definition.
- Search for services that implement a specific binding.
- Search for services that implement a specific interface.
- Search for services that support a specific protocol.

UDDI, when used alone is classified under keyword-based discovery methods. However when used with WSDL as described above, it provides table-based discovery capability.

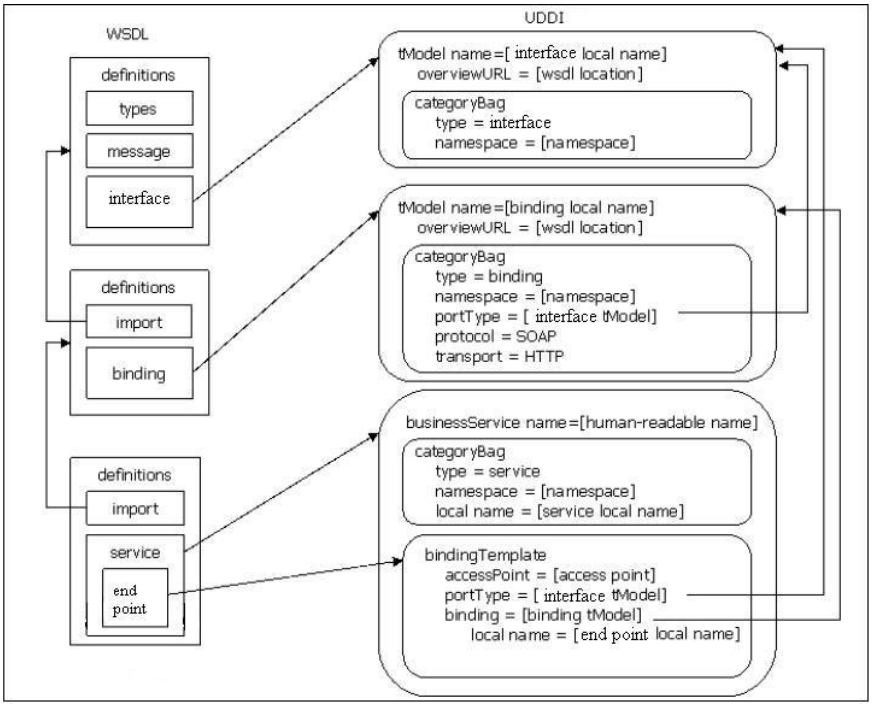

Figure 2.9 WSDL usage with UDDI

2.3.4. Semantic Web Service Discovery Methods

Semantic Web service discovery methods exploit Semantic Web technologies to describe capabilities of Web services. Capabilities correspond to functionalities provided by Web services. Two ways to represent capabilities are:

- Providing a comprehensive functionality ontology and classifying services using this ontology. For example in Figure 2.10, a part of functionality ontology in travel domain developed in SATINE Project [18] is given. Using this ontology TempoAirBooking service of Tempo Tourism is defined as an instance of AirBookingServices class, since this class represents its functionality.

- Providing a generic description of services in terms of state transformation that it produces. For example Tempo may specify a service which requests passenger information, seat class, flight number, a valid credit card number and delivery address and produces a state transition where flight tickets are delivered to address, credit card is charged and number of available seats on flight is decreased.

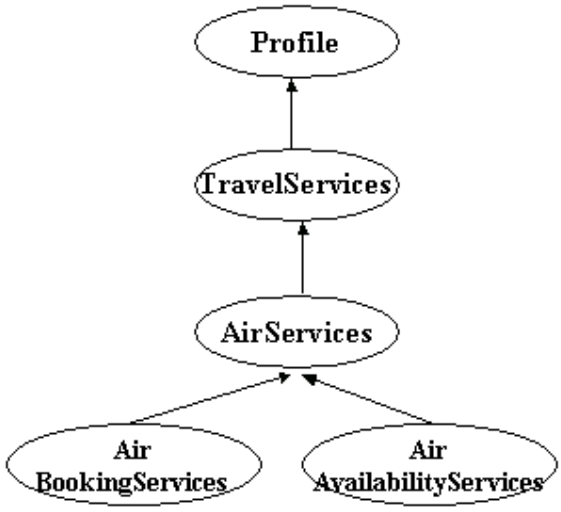

Figure 2.10 Functionality ontology in travel domain

The use of an explicit ontology of functionalities facilitates discovery process since the matching process is reduced to subsumption relation between concepts in the function ontology. However development and maintenance of such a comprehensive function ontology, even for restricted domains is diffucult. Moreover, there may be different function ontologies developed by different parties in the same domain. OWL-S supports this option since it enables subclassing ServiceProfile

class. Matching algorithms in [18], [33] and [30] use this approach for the discovery of services. [18] provides an OWL functionality ontology in tourism domain by subclassing ServiceProfile and assumes the service profiles are created as instances of the service classes which represent their functionality in this ontology. [33] assumes the availability of a functionality ontology and matches DAML-S based service ontologies using Description Logic reasoner.

[30] depends on MIT Process Handbook Project[MIT] in which a process knowledge repository is under development. The Handbook is highly focuced on business processes. It models a process using six aspects: *attributes, ports, decomposition, dependencies, exceptions* and *specialization*. Process attributes correspond to both nonfunctional and human consumable properties of OWL-S Service Profile. Decomposition and dependencies correspond to OWL-S process model where a process is modeled as a collection of sub-activities and resource flow and coordination between activities are managed by dependencies. Ports describe inputs and outputs of the process. Exceptions describe failure situations and handling these situations. Finally, specialization indexes the processes and their attributes by mapping them to concepts in taxonomies for later retrieval.

When capabilities are represented by describing state transitions, discovery process becomes more complex. An advertisement is accepted if its state transitions are matched with the request's. OWL-S also supports this option with the functional properties provided with ServiceProfile. OWL-S Matchmaker in [52, 53, 62], which is also basis for this thesis, uses this representation with its matching algorithm.

The OWL-S Matchmaker is a matching engine based on OWL-S. The service capabilities are presented using ServiceProfile class and the advertisements and requests are matched based on their profiles. The algorithm matches an advertisement with a request when all the outputs of the request are matched by the outputs of the advertisement, and all the inputs of the advertisement are matched by the inputs of the request. This guarantees that the advertised service satisfies all the

needs of request and the request provides all the inputs needed by advertised service to operate correctly.

The *degree of match* between two inputs or two outputs depends on the relation between the concepts represented by them in their OWL ontologies. These ontologies are assumed to be same and preloaded to the ontology database when registering the advertisement. If OutR represents the concepts of an output of a request, and OutA that of an advertisement, four degrees of match is recognized by the algorithm:

- **Exact:** If OutR and OutA are same or OutR is an immediate subclass of OutA. Exact is of certainly the best level match.
- **Plug in:** If OutA subsumes OutR, then OutA can be plugged instead of OutR. Plug in is the next level best match since the output returned can be used instead of what the requester expects.
- **Subsume:** If OutR subsumes OutA, then the provider may or may not completely satify the requester. This makes subsume the third level match.
- **Fail:** There is no subsumption relation between OutA and OutR.

The worst level among the match levels between outputs / inputs are selected as the global match level between the outputs / inputs of the advertisement and the request. The advertisements are sorted by the highest score in the outputs. Input matching is used only when equally scoring outputs exist.

The Matchmaker pre-computes the subsumption relation between the concepts represented by inputs and outputs of an advertisement and the concepts in the ontology database during registration. When a request comes, the matching engine just looks up for the subsumption relation between the concepts of outputs/inputs of advertisements and requests. No inference is required during the query phase.

The problem with Matchmaker is that it exploits only a subset of the state transitions produced by the service. Using only inputs and outputs for matching

services causes the retrieval of services that have similar signatures, but different purposes.

The proposed methods in [65] and [60] use both options to increase the precision and recall values. The former describes advertisements and requests as RDF graphs . Each advertisement is an instance of a service type and is the root of the graph. The properties (inputs, outputs, etc.) of the service are the other nodes in the graph. The matching algorithm matches two graphs, if the service types are equal or one is subtype of the other and their properties match.

In [60], WSDL is enriched using DAML+OIL ontologies and the descriptions are kept in UDDI in such a way that they enable inquiries based on ontology elements. Existing WSDL constructs are related to DAML+OIL ontologies via *extensibility* in elements and attributes supported by WSDL specification version 1.2. Both operations and message parts which are input and output parameters are mapped to ontological concepts. Moreover new tags are added for preconditions and effects.

These semantic annotations are kept in existing structures of UDDI and an interface is provided to enable queries using semantic annotations. Four different tModels are created in UDDI registry: one for representing functional taxonomy of operations in a specific domain, one for the input concepts, one for the output concepts and finally one for grouping operations with their inputs and outputs into keyedReferenceGroups.

The semantic matching algorithm, in the work, requires the specification of the user request in terms of ontological concepts. Afterwards, this specification is converted into a UDDI query. The algorithm first finds the services that are classified under user specified functionality concept and then matches input and output concepts and finally, as an optional step, matches precondition and effect concepts. This approach enriches existing industrial standards with semantics without requiring comprehensive modifications and additions.

Semantic discovery methods operate on descriptions based on shared understanding. This decreases synonyms and homonyms problems that occur in keyword-based and table-based methods and increases recall and precision values. All the semantic discovery methods exploit ontologies somehow. The problem with their usage is that they always assume the advertisements and requests share the same ontologies. This assumption is unrealistic in a huge domain like Web. In this work, a solution is proposed to this problem.

2.4. Web Service Composition

Most often a business process or a user requirement can not be carried out by a single Web service and therefore composition of different services based on user requirements and constraints, is needed. The composition problem is defined in [2] as finding an integrated schema of atomic Web service operations such that when executed, the desired effect is produced. The composition problem can be reduced to three subproblems[64]:

- **Composition planning:** Given a user request, the first step is to make a plan that describes how Web services interact and how the functionality they offer can be integrated to provide a solution which satifies the request.
- **Web service discovery**: Following the plan generated in the first phase, it is necessary to find the Web services that perform the tasks required.
- **Web service execution:** Given a plan and a set of Web services that perform the tasks in the plan, the third problem is to execute the services and manage the interaction with them.

Service composition problem has been actively studied by database and agent communities. Manual, semi-automated and automated solutions are proposed to the problem. In manual solutions, the user generates the workflow, finds the services and sends them to the execution engine. However due to the increase on the number of services it becomes more and more diffucult for users to deal with

locating the exact services and integrating them. Semi-automated techniques, facilitates user tasks by making suggestions for service selection, however the user is still responsible for constructing the workflow and making service selection from a short list. Automatic techniques are explained in detail in the next section.

2.4.1. Automated Web Service Composition Techniques

Automated composition aims to perform all the phases in composition life cycle without human intervention. This includes discovery, planning and execution phases. Automated discovery involves the automatic location of the services that have a particular capability and that adhere to requested constraints[38]. This requires formally self described (having semantics in the desription itself) and machine processible web services and algorithms that match these services. Keyword and table-based discovery methods like UDDI and UDDI with WSDL are not suitable for this purpose since they are mostly syntactic and have low precision and recall values. Semantic discovery algorithms well suit these requirements since they are capable of operating on self described data and have greater precision and recall values. From this point of view, deductive methods are most suitable for automation, however due to the efficiency problems, semantic methods are preferable.

For the automation of the planning and execution phases, AI or similar technologies are exploited. The automation techniques are classified into two categories in [2]: rule-based composition techniques and planning-based composition techniques.

2.4.1.1. Rule Based Composition Tecniques

These techniques are based on a static set of rules to determine whether two services can be composed or not. If the properties of two services match according to the rules they are added to the composition. One example is [43] in which the request is described using CSSL (Composite Service Specification Language).

Afterwards, composition plans that satify the request are generated using the composability rules. If more than one plan is generated, the requester chooses one of them in the selection phase. Finally, the detailed description of composite process is presented to the requester. SWORD [54] is another example to rule based approaches in which requester specifies the initial and final states of composite service and then the plan is generated using a rule-based expert system.

2.4.1.2. Planning Based Composition Tecniques

AI planning techniques are mostly used techniques for automated web service composition due to the resemblance of two problems. Given an initial and final state, both seek an ordered set of operations that would lead to the final state from the initial state.

Different planners are applied to the problem. The major approaches are using PDDL, situation calculus and Hierarchial Task Network (HTN) planning and a new approach uses event calculus for planning purposes.

In HTN planning, a set of primitive tasks is found by recursively decomposing the given initial composite task into subtasks until primitive tasks are reached. Finally a plan is generated which is an execution order of primitive tasks that are elements of this set. This method is applied in SHOP2 [69] .

Situation calculus, which is a first order logical system of actions and situations, models the dynamically changing world. The actions, when executed, either changes the domain knowledge or changes the current state. The work in [42] extends and uses GOLOG, which is a logical programming language built on top of situation calculus, for solving the service composition problem. In the system, the user request which is in the form of a generic procedure and the user constraints are coded in situation calculus. The web services are conceived as actions which can be either primitive or complex. The preconditions and effects of the service are also coded in situation calculus. The system uses GOLOG solver to execute the generic

procedure and generate the aimed effect. The output of this process is a running application which satisfies user request is built.

PDDL is widely recognized as a standardized input for the state-of-the-art planners [63]. DAML-S (earlier version of OWL-s) has been strongly influenced by PDDL language. This facilitates mapping from one representation to another. During planning DAML-S descriptions can be translated to PDDL format and then different planners can be exploited for solving the problem. The work in [40] is an example to Web service composition methods based on PDDL.

More information for composition techniques can be found in [2] and [63].

2.4.2. Automated Composition using Event Calculus

Event Calculus, which is another logical formalism in basic theory, is applied to Web service composition problem in [2]. Two usage purposes of the formalism related with composition problem are :

- Finding a final situation for a theory that is composed of Event Calculus rules, axioms and initial state.
- Finding a *narrative* which is a series of events, through abduction that satisfies the theory and the goal state.

The formalism is used in this work for both purposes.

2.4.2.1. Event Calculus

Event calculus is suitable for domains where *events* dyamically affect and change the state of the world. These changes are captured in *fluents* of the world. Similar to variables, fluents change their valuations as a result of affecting events. Events can be simple or compound. Compound events contain simple or compound events in their time frames. The events and fluent valuations are relativized with respect to time.

In order to define the theory of a problem domain some predicates are used. The event calculus predicates are given in Table 2.2 (taken from [2]). Each event calculus theory is composed of:

- Default axioms of Event Calculus
- Axioms that are specific to problem domain
    - Initial state axioms
    - Effect axioms
    - Unique names axioms

Table 2.2 Event Calculus predicates

Predicate	Meaning
$T_1<T_2$	Time point T_1 precedes time point T_2 in the time line.
Happens(E, T_1, T_2)	Event E happens during time frame between T_1 and T_2 where $T_1<T_2$.
Happens(E, T)	Instantaneous Event E happened at T defined as Happens(E, T, T).
HoldsAt(F, T)	Fluent F holds at time T.
Initially(F)	HoldsAt(F, T_0) where T_0 is the time of the initial state.
Initiates(E, F, T)	HoldsAt(F, T) when Happens(E, T, T_A).
Terminates(E, F, T)	HoldsAt(¬F, T) when Happens(E, T, T_A).
Releases(E, F, T)	Valuation of Fluent F is released, removing the inertial constraints on it.
Clipped(T_1, F , T_2)	HoldsAt(F, T) where $T_1< T <T_2$
Declipped(T_1, F , T_2)	HoldsAt(¬F, T) where $T_1< T <T_2$

In order to demonstrate the formalism, the example taken from [2] is given below. The theory is "*If there is a turkey in the world and if a pointed gun is fired than the turkey dies*."

The initial state axioms are:

Initially(alive(Turkey)) (2.1)

Initially(¬loaded(Gun)) (2.2)

The effect axioms are:

Initiates(load(Gun), loaded(Gun), T) ←

HoldsAt(¬loaded(Gun),T) (2.3)

Terminates(fire(Gun), alive(Turkey), T) ←

HoldsAt(loaded(Gun), T) ∧ HoldsAt(alive(Turkey), T) (2.4)

Terminates(fire(Gun), loaded(Gun), T) ←

HoldsAt(loaded(Gun), T) (2.5)

Unique names axioms

fire ≠ load (2.6)

The narrative which satisfies the theory :

Happens(load(Gun), T1) (2.7)

Happens(fire(Gun), T_2) (2.8)

$T_1<T_3$ (2.9)

$T_2<T_3$ (2.10)

The theory which is composed of default axioms and domain specific axioms (2.1-2.6) together with the narrative (2.7-2.10) leads to HoldsAt(¬loaded(Gun), T_3) and HoldsAt(¬alive(Turkey), T_3).

2.4.2.2. Composition with Abductive Planning

In the first part of the work in [2], the Event Calculus is used for planning purposes. The Web services are bound to theory by modeling Web services as events with parameters and external Web service calls as predicates with parameters. The corresponding event of a Web service which finds the availability of a flight between two locations is given in Figure 2.11.

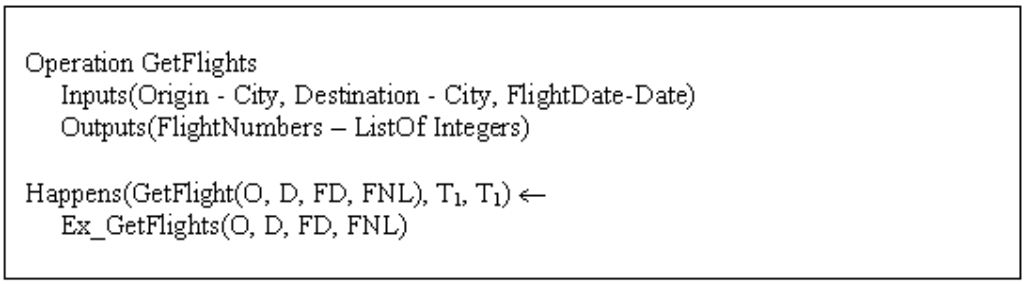

Figure 2.11 Translation of a Web service to an event

Planning in this work is done by using *abduction* with the Event Calculus. Abduction is a technique used with theorem solvers for planning. The main idea in abduction is to start from the goal and find the axioms that might result in satifying the goal. An abductive theorem prover is extended to cover issues related with service composition.

2.4.2.3. Web Service Execution with the Event Calculus

In the second part of the work, the Event Calculus is used for executing OWL-S composite processes. This requires the conversion of OWL-S process model constructs to event calculus constructs. Atomic processes for instance are converted to simple events with parameters. The translation of composite procceses which contain control constructs are done by using compound events. Details about the conversion can be found in [2]. The work also provides an algorithm for the conversion.

Assuming the Web services that correspond to external Web service calls are pre-discovered, in the second part of the work a given generic procedure expressed in OWL-S is executed.

2.4.2.4. Results of Using Event Calculus in Web Service Composition

The work in [2] showed that both composition planning and execution can be accomplished within the Event Calculus framework. This formalism is especially suitable for modelling concurrent and ordered events using time frames.

The proposal in [2] neglects the discovery phase and assumes that the Web services are pre-discovered. The discovery approach implemented in this thesis will be integrated to this work to complete the missing piece of the solution.

CHAPTER 3

BACKGROUND INFORMATION ON SCHEMA MATCHING

Advances in information technology have lead to an explosion in the number of data sources accessible to a user. Many new database applications have arisen over time, which require semantic integration of these data sources. Schema integration, data warehousing, data integration systems, e-commerce and semantic query processing are examples of such application domains [17]. Since the data sources, to be integrated, are generally designed independently, most often to be self-contained, there might be several kinds of semantic and structural heterogeneities between them. So in all of these application domains, schema matching is the key operation before going further. In data integration systems, for instance, the users are provided with a uniform query interface (called mediated schema) to a multitude of data sources and the user query is constructed based on the mediated schema [36]. However before posting the query to data sources, it must be reconstructed to conform to the schemas of the data sources. So the mediated schema and the source schemas are matched first and then the query is rewritten depending on the matching results. Manual schema matching is tedious, error-prone and clearly not possible in the scale of the application domains. This has led to development of the automated techniques for the matching process.

This chapter is structured as follows: in Section 3.1, the matching problem is defined with illustrations. Section 3.2 describes the architecture of a matching system and Section 3.3 gives the classification of matching techniques. Section 3.4 presents the current proposals to matching problem. Finally, Section 3.5 focuses on ontology matching.

3.1. Definition of Matching Problem

Schema matching is the process of finding *semantic correspondence* between the two given schemas. Semantic correspondence is defined as a set of *direct matches* each of which binds a source schema element to a target schema element, if two schema elements are semantically equivalent [70].

Semantic equivalence is based on both the concept denoted by the schema element and the context in which the element is defined. In Figure 3.1, independent from the contexts, the real world concept denoted by label “France” in both parts, corresponds to “a country in western Europe”. However, when contexts are considered, the element labeled with “France” on the right-hand side corresponds to landscape paintings from France in the real world.

Two schema elements might be semantically similar despite some schematic heterogeneity. The matcher must be capable of handling them and finding corresponding elements despite them. Different classifications for these heterogeneities are given in [13, 28]. Before listing them, two simple schemas, given in figure 3.2, will be introduced for illustrative purposes. The semantically equivalent concepts “Instructor” and “Teacher” are modeled in the first and second schemas, respectively. Besides, “HomeAddress” which is range of “homeAddress” attribute is modeled in both schemas.

Four types of heterogeneties are attribute definition incompatibilities, concept definition incompatibilities, abstraction level incompatibilities and finally, schematic discrepancies.

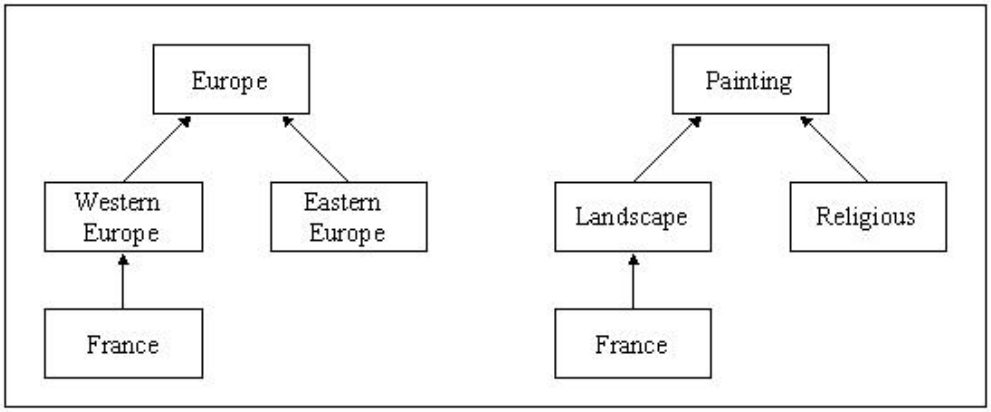

Figure 3.1 Different contexts for the same concepts

Attribute Definition Incompatibilities

These incompatibilities arise when two attributes that are semantically alike are modeled in different ways. These differences can be classified into five categories:

- **Naming conflicts:** The similar attributes might have different names, which are synonyms like the attributes "firstName" in the first schema and "givenName" in the second.
- **Data Representation Conflicts:** The similar attributes might have different data types or representations. The attribute "ID" in the first schema may be represented as a 10-digit integer, where in the second as 12-character string.
- **Data Scaling/Precision Conflicts:** The similar attributes might be represented using different units, measures or precision. The attribute salary might have values in $ where the wage in YTL, or salary might have values which are multiples of 1000.

- **Default value Conflicts:** The similar attributes might have different default values. The default value of “gender” might be “male” in the first and “female” in the second.
- **Constraint Conflicts:** The similar attributes might have different constraints. The “salary” might have values in [1000, 5000] interval, where the “wage” might have values in [750, 3500] interval.
- **Temporal Conflicts:** The similar attributes might have temporal differences as in current “salary” vs. past “wages”.

Concept Definition Incompatibilities

These incompatibilities arise when two semantically alike concepts are modeled differently and are classified into three:

- **Naming conflicts:** Similar to attributes, the concepts might have synonymous names as in “Instructor” and “Teacher” case.
- **Attribute number-precision Conflicts:** An attribute of one concept may be more preciously represented than the other. In the first schema, the “Instructor” has just “phone” attribute, where in the second, this attribute is detailed as “mobile” and “home” phone. In addition, there might be absence-presence differences between attributes specified.
- **Missing data item Conflicts:** These conflicts arise when one of the concepts has a missing attribute, which can be deduced using an inference mechanism. For example, “Math Instructor” in the first schema corresponds to “Teacher” in the second, whose “type” attribute always has value “Math”.

Abstraction Level Incompatibilities

Two semantically similar entities, which can be either concepts or attributes, might be represented using different levels of abstraction. Two kinds of abstraction level incompatibility are:

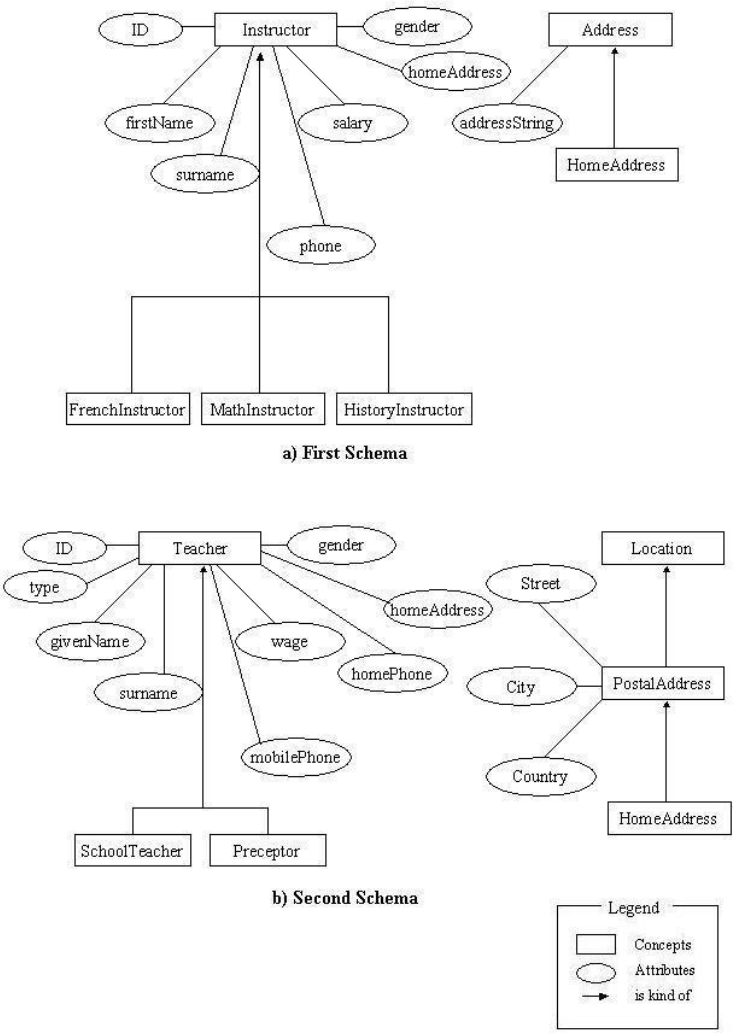

Figure 3. 2 Examples of heterogeneities between schemas

- **Generalization-Specialization conflicts:** These conflicts reflect classification differences in the subclass hierarchy. Three types are:
    - Location based differences: Two similar entities might be located at different levels in their hierarchies. In the first schema, “Home Address” is specialized from “Address”, in the second it is specialized from “Postal Address” which is specialized from “Location”.
    - Criteria-based differences: Two similar concepts might be specialized based on different criteria. In the first schema, “Instructor” is specialized depending on what the instructor teaches. However, in the second, “Teacher” is specialized based on where the teacher teaches.
    - Specialization kind differences: Two similar concepts might be specialized in different ways. The “Instructor” is specialized by course type in the first schema. In the second, the course type is listed for each “Teacher”.
- **Aggregation conflicts:** These conflicts arise when an aggregation is used in one schema to represent a set of entities in the other. The attribute “addressString” of “Address” in the first schema aggregates “Street”, “City” and “Country” attributes of “PostalAddress” in the second. The aggregation can also be in the form of n-to-m where two or more attributes in one schema aggregate two or more attributes in the second.

Schematic Discrepancies

These conflicts arise when data value, attribute and concept are used interchangeably in different schemas. To illustrate them, three different schemas are given in Figure 3.3. The first schema consists of a single “Schedule” relation that has a tuple per “instructor” per “class” with “dayTime” information, which shows when the instructor meets that class. The second schema has also one “Schedule” relation that has a tuple, which consists of an instructor and an attribute for each class that

shows daytime information. Finally, the third schema has one relation per class each consist of an instructor and daytime information.

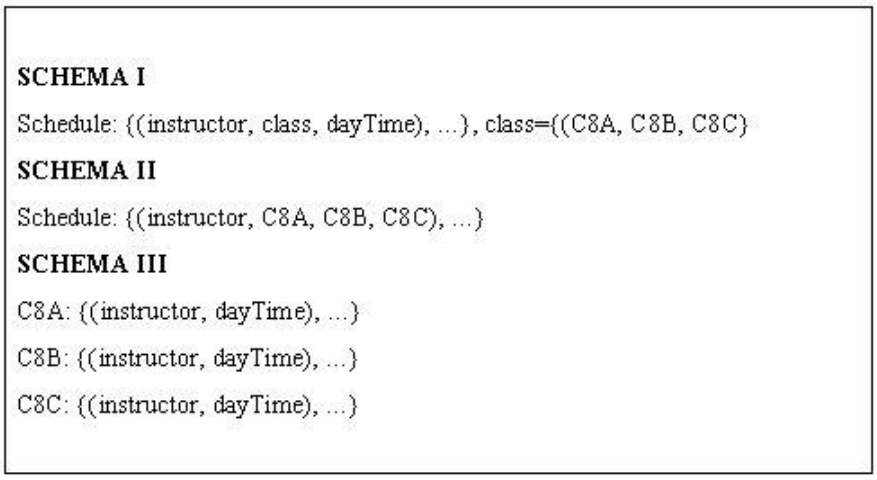

Figure 3. 3 Examples of schema discrepancies

Three different schema discrepancies are:

- **Data value attribute conflict:** This conflict can be defined as the usage of a data value of an attribute in one schema as an attribute in the other. The values of "class" attribute in the first schema correspond to attributes in the second schema.
- **Attribute concept conflict:** This conflict can be defined as the usage of an attribute in one schema as a concept in the second. The attribute "C8A", for instance, corresponds to relation "C8A" in the second.
- **Data value concept conflict:** This conflict can be defined as the usage of a data value of an attribute in one schema as a concept in the second as in

Schema I and Schema III where the data value “C8C” corresponds to relation “C8C”.

3.2. Architecture of a Generic Matching System

Given two or more schemas, a schema matching system outputs the matches between the input schema models. A simplified overview of a matching system is given in Figure 3.3. Inputs to the system are schemas, user feedback and external knowledge, whereas the outputs are matches between schema elements.

In order to implement a generic solution, different schema models (ontology, XML document, relational schema, etc.) are converted to an internal representation. The internal representation may be a tree [34, 15] or directed labeled graph [44].

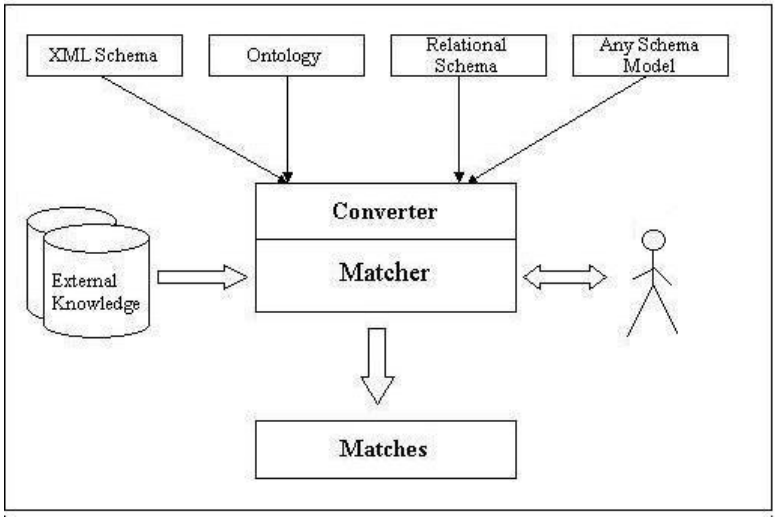

Figure 3. 4 Architecture of a Matcher

Frequently, the schema information or the data instance in hand is not enough for accurate matching. So in most of the schema matching approaches external

knowledge is exploited to improve the quality of the results. External knowledge may be domain specific rules [49], domain knowledge in the form of ontology, precompiled thesaurus or general purpose lexicons like WordNet [46]. Knowledge gained from previous matches can also be an input to the matching system. For instance, in [35], they use a mapping knowledge base, which contains a classifier for each of the schema element seen in the past. Also in [15] and [1], the results of previous matches are stored and used during new matches.

The output of the system is a set of matches. However schema matching is inherently subjective and schemas may not completely capture the semantics of the data they describe. There may be several plausible or completely false mappings. So user intervention is essential in order to accept, modify or reject mappings found by the system. Several iterations may be done based on user feedback to improve the quality of matches as in [1].

3.3. Classification of Matching Approaches

Database community has been studying the problem of automating schema matching for a long time. A classification of matching approaches is proposed in [55]. A slight variation of this classification is given in Figure 3.5. The approaches are classified mainly as *individual* and *combined* matchers. Individual matchers are early solutions to the problem and find matches using only a single criterion. Combined matchers, in contrast, find matches based on multiple criterions, by using individual matchers as building blocks. Hybrid matchers combine the individual matchers in a fixed manner, where as composite matchers allow dynamic combination of both individual and hybrid matchers depending on schema characteristics, application domain and even results of previous matches [59].

Instance-based solutions depend on data content and most of them use information retrieval or learning techniques to extract mapping information from instance data. In contrast, schema-level matchers consider only schema information such as structures (data types, classes, attributes) as well as properties of schema

elements like name, type, etc. Schema level matching can be done either at element level, where only element relevant information is considered or at structure level, where the context in which the element is located is also taken into consideration.

Element level matching can be done with syntactic, semantic or constraint based methods. In s*yntactic matching* schema elements with the same or similar names are matched. The similarity is determined by using syntax driven techniques such as prefix-suffix (affix) matching, edit distance, n-gram or synonym-hyponym relations. Exploiting synonym-hyponym relations requires using external resources: domain, enterprise-specific or general purpose dictionaries, global namespaces, thesaurus or is-a taxonomies containing common names, synonyms and descriptions of schema elements, abbreviations, etc. Most of the time, the matching result is presented to the user with similarity coefficient, which is a value in the interval [0,1] and shows the degree of system's confidence about the mapping.

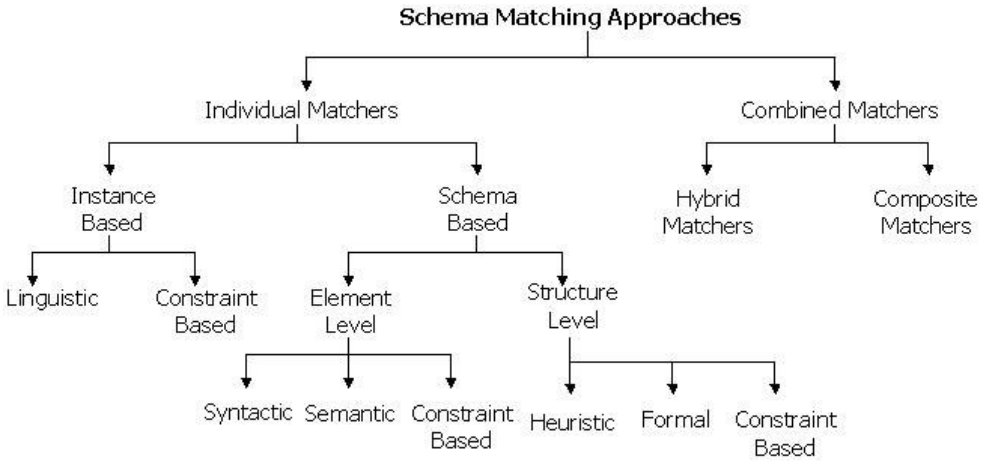

Figure 3.5 Classification of matching approaches

Semantic matching, tries to map the concepts denoted by the element names. Finding concepts requires the usage of external resources such as domain ontology or lexical databases. The mapping is presented with a similarity relation that holds between the concepts denoted by the element names. Similarity relation may be equivalence, more/less general, mismatch and overlapping [20]. *Equivalence* is the strongest semantic relation. *More general* and *less general* implies that one of the concepts is the specialization of the other. *Mismatch* tells two concepts are disjoint which means the first concept has containment relation with the negation of the second or vice versa. Finally, overlapping implies there exists neither a containment nor a mismatch relation between the concepts. In some works [48, 39], these relations are converted to coefficients.

Constraint based methods, take into account the type similarity, key properties, referential integrity relations, etc.

Structural matching can be done using heuristic, formal or constraint based solutions. In *heuristic methods*, all the names along a path in the schema hierarchy are concatenated. Similarity is searched between the concatenated names using string comparison methods. Alternatively, external taxonomic structures are used to compare the paths taken from schemas with these structures and identify similar terms. In *formal methods*, the general approach is to translate the matching problem, which is composed of the two schemas and possible matches, into propositional formula and then to check it for its validity via SAT decider. A SAT decider is a correct and complete decision procedure for propositional satisfiability [20].

3.4. Current Proposals To Schema Matching Problem

In most of the recent studies on schema matching, hybrid or composite matchers are used and individual matchers are exploited as building structures. Cupid [34] and S-Match system [20] are implemented as hybrid matchers where as COMA [15], COMA++ [1], Protoplasm [3] and H-Match system [7] are composite matchers. They all use different kinds of individual matchers.

Cupid is a hybrid of a heuristic structural matcher and several syntactic element level matchers. It provides a generic and schema based solution. In order to provide a generic solution, all input schema models are converted to tree structures. The main flow of the system consists of three phases: syntactic matching, structural matching and match selection. The first phase computes a syntactic similarity coefficient between the names (labels) of schema elements. The syntactic comparison includes morphologically normalizing and categorizing element names and comparing them using synonym-hyponym relation by using a pre-compiled thesaurus. If no relation is found, affix matchers are used. In the second phase, with a bottom-up approach weighted similarity is calculated for each pair of nodes based on both structural and syntactic similarity coefficients. Structural similarity coefficient is calculated based on data type compatibility of for leaf nodes. For non-leaf nodes, structural similarity between two nodes is based on the number of leaf node pairs whose weighted similarity exceeds a given threshold. Thus the similarity of two non-leaf nodes mostly depends on the similarity of their leaf nodes. The aim of this approach is to decrease the effect of differences in abstraction levels.

S-Match system [20, 21] is a hybrid of a formal structural matcher and several syntactic and semantic element level matchers. Syntactic matchers are similar to the element level matchers used in Cupid and COMA, except that the comparison is done between concepts denoted by the schema elements instead of labels and result of the comparison is a semantic relation instead of a similarity coefficient. Semantic matchers exploit the senses and glosses of the concepts found from WordNet. The first sense based matcher return the semantic relation, which is provided by WordNet, between two senses. The second measures distance between two senses in the WordNet hierarchy. Gloss based matchers, exploit NLP techniques on glosses of the senses. One of them, for instance, extracts labels of the first sense and counts the number of their usage in the gloss of the second. If the number exceeds a threshold, it returns less-general relation. Using these element level matchers, S-Match system takes two schemas converted to tree structure and

computes the strongest semantic relation holding between the *concepts of the nodes* from two trees. The concept of a node is computed as the intersection of the concept denoted by the label at that node and the concepts denoted by the labels of ancestor nodes. The concepts are expressed in logical propositional language. The atomic concepts are atomic formulas. Complex concepts are obtained by combining atomic concepts using connectives of the set theory. For each pair of node, a *context,* which is the conjunction of all relations holding between concepts of labels mentioned in two nodes, is computed. Finally the strongest relation between the two nodes is found by testing satisfiability of each semantic relation (starting from the strongest) against the context via SAT decider.

COMA [15], conforms to composite matchers in matcher hierarchy. It provides an extensible library of algorithms which consists of individual matchers which are schema-based and syntactic, hybrid matchers, which are composed of individual and structural matchers, and a completely novel one, a reuse-oriented matcher, which exploits previously obtained results for the new matching requests. The library can be extended with new matchers as well. A distinct feature of COMA is its support to user interaction. The user can construct the matching strategy, select the matchers, determine the combination strategy of the matchers' results, approve/disapprove the results and re-iterate the matching task to gradually refine and improve the accuracy of the results. COMA++ [1] extends COMA with major improvements. A comprehensive graphical user interface and a new matcher for ontology matching are implemented. Problems related to matching of large schemas are reduced. Based on the fragment-based matching approach that was proposed in [56], the large schemas are divided into fragments, matches between the fragments are found and their results are combined later on.

Protoplasm [3] is inspired by COMA and tries to provide an industrial-strength schema matcher, one that avoids fragility problems, is customizable for use in practical applications, and extends the range of matching solutions being offered.

H-Match system is devoted to matching ontologies. Similar to previous works, it converts the input ontologies to a language independent model, which is called H-Model. It provides three individual matchers, one of which is a linguistic matcher, and four structural matchers, which address different levels of richness in ontology descriptions. The linguistic matcher names an ontology element name as *basic term* if there exists an entry for that name. Otherwise it is named as *compound term,* which is composed of several basic terms. The basic term appearing on the left side of the compound term is assumed to denote the specification of the meaning of the term appearing on the left side and semantic relations between names are determined based on this assumption. Four structural matchers are *surface*, *shallow*, *deep* and *intensive*. Surface matcher considers just linguistic features of concepts and suitable for poorly structured ontologies. Shallow matcher considers the concept names and properties. Deep matcher takes into account the semantic relations of the concept besides its properties. Finally, intensive matcher, in addition to the deep matcher, considers property values during matching.

All of the works that are described above deal with *one-to-one matches* where two individual object sets in different schemas are matched. 'address'='location', 'lecturer'='teacher' are examples to one-to-one matches. While these matches are common, correspondences between real world schemas also include one-to-n, n-to-one or n-to-m matches, which are named as complex matches. A *complex match* specifies that a combination of elements in one schema may correspond to another combination in the second schema (aggregation conflicts). Schema matching approaches can further be classified depending on the cardinality they handle. Most of the work on schema matching deals with one-to-one matches. There have been only a few works on complex matching. [13], [16] and [70] and [22] are known works that handle complex matches which are all instance-based, hybrid matchers.

Finding complex matches is fundamentally harder than one-to-one matches since the number of candidate one-to-one matches between a pair of schemas is bounded (by the product of sizes of two schemas), but the number of candidate

complex matches is unbounded. The iMAP system in [13] employs a set of *searchers* each of which considers a meaningful subset of the candidate space, corresponding to specific types of attribute combinations. It exploits domain knowledge both in match generation and match selection phases. A mechanism for explaining decisions made by the system is also included which enables users ask some questions on matching results and make corrections according to the answers.

The work in [70] makes use of domain ontology. Three types of matches are defined: *merged/split values*, *superset/subset values* and *object set name as value*. The limitation to the approach is the need to manually construct application specific domain ontology.

The GLUE system in [16] handles simple and complex mappings between ontologies. A similarity definition is constructed from the *joint probability distribution* of the concepts involved. The system deals with only concepts. The properties of the concepts are not taken into account. Machine learning techniques and multi-strategy learning are used for computing concept similarities. Available domain constraints and heuristic knowledge are exploited to improve the matching accuracy. For complex matching two kinds of operators are used: union and difference. The mapping candidates that are built on these two operators are considered only.

Given a set of web interfaces in the same domain, DCM framework in [22] solves the problem of discovering matches among those interfaces. In contrast to previous works, which handle two schemas at the same time, it matches all the schemas at the same time by using data mining techniques.

3.5. Ontology Matching

The Semantic Web vision, as described in the previous chapter, can only be realized with the use of ontologies. Ontologies provide the shared-understanding required for converting Web content from human-consumable form into machine-

consumable form, which leads to automation, integration and reuse of data across various applications. However, due to the de-centralized nature of the Web, most probably there will be an explosion in the number of ontologies, most of them describing similar or overlapping domains, but using different languages, conceptualizations, modelling styles and terms. In order to process data from disparate ontologies, semantic correspodences between their elements must be found first [16, 7, 67]. This requirement is similar to schema matching problem. This similarity has led to the extention of schema matching systems to cover ontologies as well [1, 20]. In addition, new approaches are proposed which focus on just ontologies as in [16, 7] .

In this thesis, a subset of these approaches will be adapted to overcome the ontology difference problem in semantic Web service discovery.

CHAPTER 4

SEMANTIC SERVICE DISCOVERY WITH SCHEMA MATCHING

Service discovery methods and the issues that limit the quality of them are discussed in the second chapter. **S**emantic **S**ervice Discovery with **S**chema **M**atching (SSSM) approach addresses an untouched issue in semantic service discovery. It proposes an extension to the semantic discovery methods to cope with different ontologies that model the same or similar domains. This chapter introduces SSSM approach. Section 4.1 positions it among other discovery approaches. Section 4.2 describes the system architecture. The service matching approach is explained in Section 4.3. The Schema Matcher extension is presented in Section 4.4. Section 4.5 describes the SSSM integration with Event Calculus executer presented in Chapter 2.

4.1. Positioning of SSSM

Semantic service discovery methods exploit ontologies to provide a shared understanding between advertiser and requester. This improves discovery results by capturing semantics of the request better and addressing the problems related to synonyms and homonyms. Three usage types of ontologies during discovery are:

- Providing a common functionality ontology to classify services and retrieve services conforming to requested functionality by using subsumption relations.

- Describing the state tranformation produced by the service using ontologies and matching the state transitions.
- Exploiting both.

SSSM uses the second approach and extends it by incorporating a schema matcher component. This extension can be integrated with other usage methods, as well. The method increases the recall value by retrieving services that are rejected by other semantic matchers due to the ontology differences. Figure 4.1 shows the positioning of SSSM in the recall/precision framework. When the advertisement and the request share the same ontology, SSSM behaves similar to other semantic methods. The intersection area in the figure shows the discovery results under these conditions. The rest of the area shows the cases in which different ontologies are used.

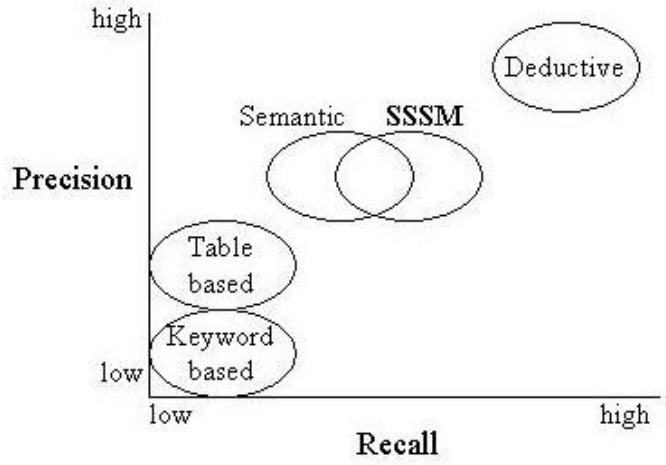

Figure 4.1 SSSM position in recall/precision framework

4.2. SSSM System Architecture

The general view of the SSSM system is given in Figure 4.2. The main components of the system are: Repository Manager (RM), Service Base Manager (SBM), KnowledgeBase Manager (KBM), Matching Engine (ME) and Schema Matcher (SM).

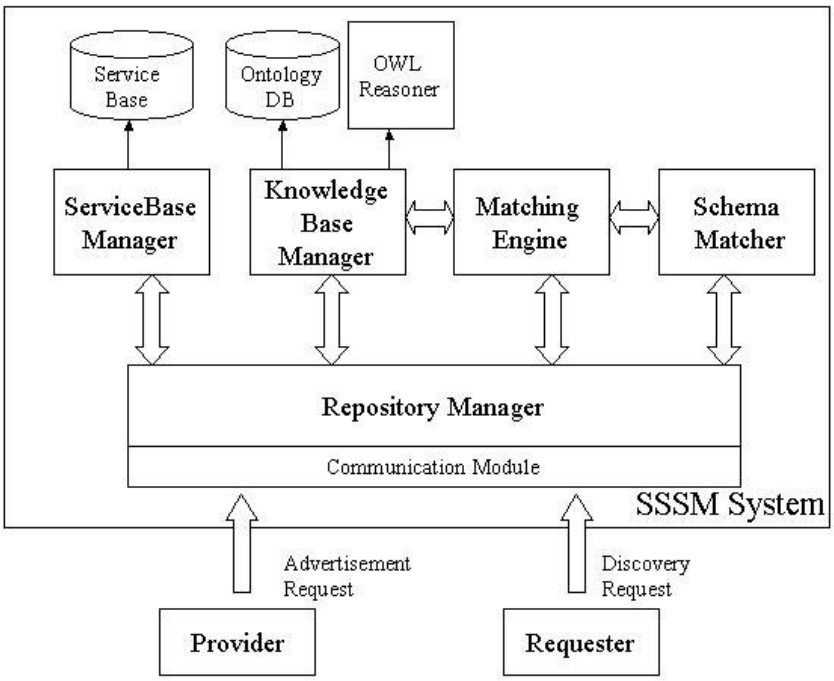

Figure 4.2 SSSM System Architecture

Repository manager accepts the requests and orchestrates other components. SBM processes advertisements and keeps the parts of the service descriptions that are needed for later comparison in a database table. MySQL [50] is used as the database management system. KBM manages OWL ontologies and evalutes the

semantic relation between two concepts using an external OWL reasoner. Pellet [47], which is a sound and complete OWL-DL reasoner, is used. The OWL ontologies are processed using Jena API [26] . The ontologies are saved in a database. The ME, matches the request with a set of advertisements. SM finds the semantic similarity degree between two concepts using schema matching algorithms.

Repository Manager receives requests from the external world via communication module. Two kinds of requests are:

- **Service Advertisement Request** with which the provider registers the service it offers.
- **Service Discovery Request** with which the requester describes the service it needs and searches for the services that conform this description.

Both the advertisement and discovery requests point to OWL-S Profiles. In the former, the OWL-S profile describes the capabilities of the service to be advertised. In the latter, the service with the needed capabilities is described. In this work, similar to [52] description of capability is reduced to describing inputs and outputs. The preconditions and effects are omitted. Figure 4.3 shows an advertisement and a discovery request with the ontologies that describe their parameters. The advertisement finds the sports activities available in a city during a time frame. The discovery request, on the other hand, requests all the social activities available in a city during a time frame. The descriptions of the advertisement and discovery requests in OWL-S are given in Appendix C. The OWL-S files are processed using OWL-S API [47].

Depending on the request, Repository Manager starts an interaction between other components. Figure 4.4 shows the interactions between components required to fullfil the requests. When the request is an advertisement, the service profile is delivered to ServiceBase at first. ServiceBase manager keeps the aspects of the service needed for comparison. In [52], information retrieval methods are used to filter unrelated advertisements and comparison is done between the request and a set

of related advertisements. Such an initial filtering mechanism is out of the scope of this work, so it is not implemented. The only filtering criteria is the number of inputs and outputs. The services, whose number of outputs equal to or greater than the request's number of outputs and number of inputs is equal to or smaller than the request's number of inputs, are selected initially. The rationale behind this filtering will be described in the next section. For the purpose of this work, only input/output related information is kept in the ServiceBase for later retrieval. The SBM registers the service and returns a set of concepts, which are used to describe input/output parameters, with their ontologies. The ontologies are added to the ontology database via KBM and the concepts are processed by Schema Matcher.

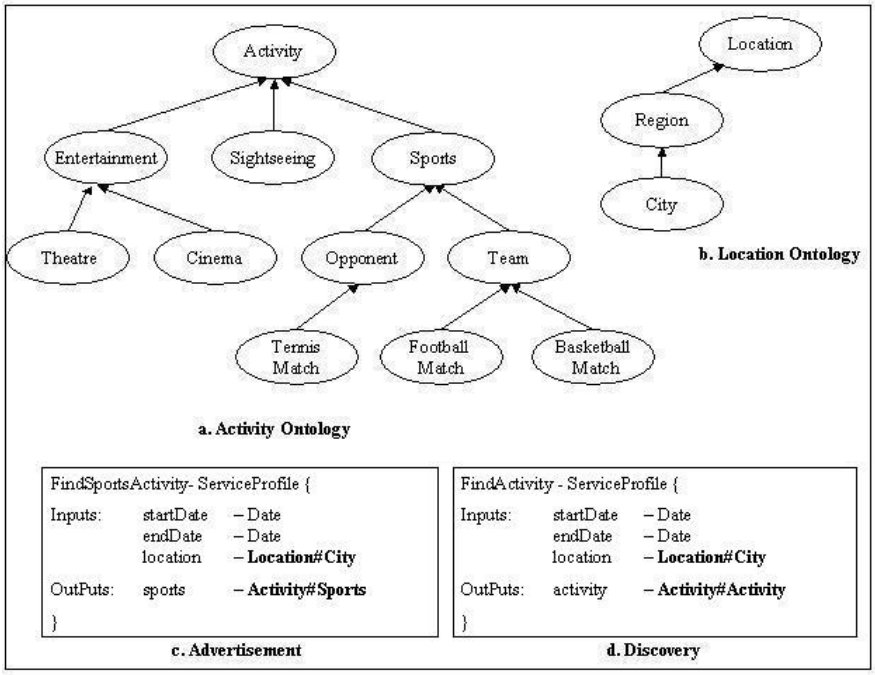

Figure 4.3 An advertisement and discovery request sample

When the request is for discovery, the profile, which descibes the required capabilities, is retrieved to SBM first. Depending on the number of input/output parameters, SBM returns a list of candidate advertisements. The Matching Engine compares the request with the advertisements and returns the ones that satify the request with input match degree, output match degree and a list of input/output matches. During comparison, Matching Engine refers to either KBM or the Schema Matcher to find the semantic similarity between two concepts. KBM computes the degree of match between them using subsumption relations. Schema Matcher computes the similarity using schema matching methods.

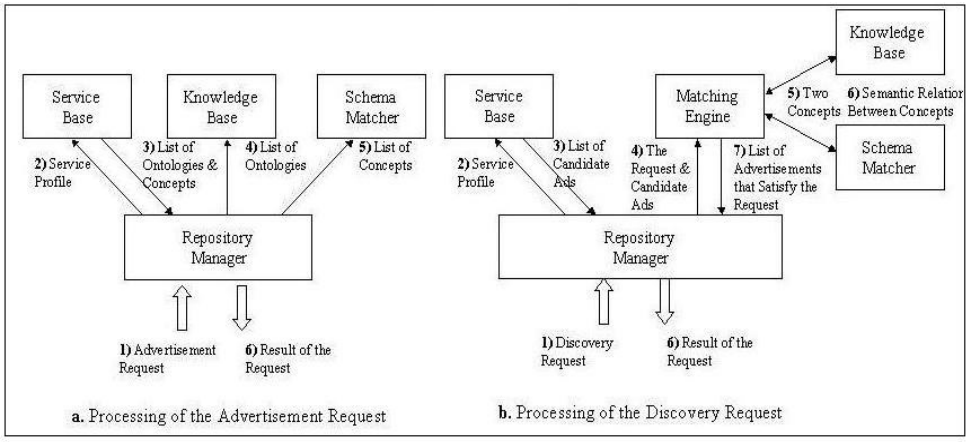

Figure 4.4 Processing of advertisement and discovery request

4.3. SSSM Matching Approach

During service discovery, the goal is to retrieve the services which are *sufficiently similar* to the request. A restriction of "sufficiently similar" to "exactly the same" most probably leads to an empty result set. The main idea that SSSM based on is to retrieve the services which can somehow fullfil the described request.

In [52], a softer definition is adapted for “sufficiently similar” by defining a discrete scale to rank the similarity. SSSM is based on this definition and extends it.

Since the capability matching of two services is reduced to input/output matching in SSSM, a request is said to be “sufficiently similar” to an advertisement if and only if all outputs of a request are matched by the outputs of the advertisement and all the inputs of the advertisement are matched by the inputs of the request. So the matching algorithm compares the input/output parameters to determine the similarity degree.

In this section, the SSSM similarity scale is introduced, first. The matching algorithm is presented, next.

4.3.1. SSSM Similarity Scale

Two outputs or two inputs are matched if there exists a *semantic relation* or a *semantic similarity* between the concepts that describe them. The scale used in the first case is based on the subsumption relation between concepts. It consists of four match degrees: *exact*, *plug in*, *subsumes* and *fail,* where *exact* is the most preferable, *plug in* is the next and *subsumes* is the third best level. The closeness of relationship determines where to map the relation in the scale. These degrees are defined in Chapter 2. To sum up here; suppose outR and outA are the concepts that describe the outputs of the request and the advertisement, respectively. The match degree between outR and outA is :

- “exact” if outR and outA are equivalent or outR is an immediate subclass of outA
- “plug in” if outR is sublass of outA but not an immediate one
- “subsumes” if outA is a subclass of outR
- “fail” if there exists no subsumption relation between them.

Consider the advertisement given in Figure 4.3 whose outA=Sports. If a request, whose outR is equal to Sports or Team is received, the match degree will be “exact”. If a request whose outR is equal to Football Match is received, the match

degree will be “plug in”. If a request with outR equals to Activity is received, the match degree will be “subsume”. If a request looks for a service which lists sightseeing activities, the match degree will be “fail”. All these match degrees are defined under the assumption that outA and outR are members of the same ontology.

In the second case, no explicit semantic relation is defined or can be inferred between outR and outA. However, they model similar real world entities in different ontologies. The match degree depends on the semantic similarity degree of them in this case. Semantic similarity degree is calculated by comparing both the labels denoted by the concepts and the contexts in which they appear. In order to provide uniformity, semantic similarity degree, which is in interval [0,1], has to be mapped to the scale described above. Two ways for mapping are :

- Calculating the similarity degree of outR and outA and mapping it to “exact” or “fail” depending on whether it exceeds a specified threshold or not.
- Finding the concept which has the highest similarity degree with outR (providing that it exceeds a specified threshold) among the concepts, including outA, its ancestors and descendents and identifying the result as “exact”, “plug in” or “subsume”, depending on the concepts relative location to outA.

Although the second way maps the semantic similarity degree more precisely, it requires more comparison. Since the comparison of two concepts is an expensive operation, the first method is adopted.

Consider the discovery request given in Figure 4.5 , which looks for a service that displays sports contests held within a city. The result of comparing the advertisement in Figure 4.3 with this request is *exact* since the semantic similarity degree between “SportsContest” and “Sports” is calculated as 0.76 which is sufficiently high to map it to match degree “exact”.

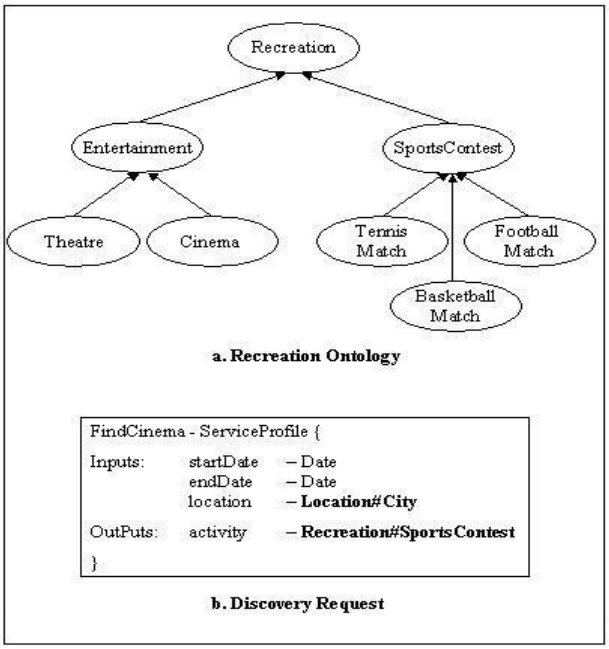

Figure 4.5 Discovery request exploiting a different ontology

4.3.2. SSSM Matching Algorithm

The main control loop of the algorithm is shown in Figure 4.6. The request is compared with a list of advertisements stored in the repository. Whenever the match degree is greater than *fail,* the advertisement is appended to the result set. Finally, the result set is sorted in order to present the advertisements, which produce the requested output with the highest match degrees, to the user, firstly. Consider two advertisements, namely Adv_1 and Adv_2, the scoring rules used to sort them are as follows:

- If $OMD_1 > OMD_2$ then $Adv_1 > Adv_2$
- If $OMD_1 = OMD_2$ & If $IMD_1 > IMD_2$ then $Adv_1 > Adv_2$
- If $OMD_1 = OMD_2$ & If $IMD_1 = IMD_2$ then $Adv_1 > Adv_2$

where OMD stands for output match degree, IMD stands for input match degree. They are equal to lowest match degrees among match degrees of output and input parameters, respectively.

```
matchServices(request - ServiceProfile, advList - List){
resultList – List
For each advertisement in advList{
        result=matchRequest(request, advertisement)
        if result!=Fail
                resultList.append(advertisement) }
return sort(resultList)
}
```

Figure 4.6 Main control loop

When matching a request with an advertisement, the output parameters are compared first. If all the output parameters of the request are matched by output parameters of the advertisement, then the inputs are compared. If all the input parameters of the advertisement are matched by the input parameters of the request then the advertisement is accepted to satisfy the request somehow and added to the result set. The algorithm for comparing parameters is given in Figure 4.7. A matrix of match degrees is constructed by calculating the *degree of match* for each pair (a,b) where a and b are elements of the first and second parameter list. If one of the parameters in the first list can not be matched by any of the parameters in the second, the overall match will fail. Otherwise, a parameter assignment list, which shows one-to-one mappings of parameters, is constructed.

A parameter in the first list can be matched with more than one parameter in the second. Choosing the one with the highest degree for assignment can, sometimes, cause the overall matching to fail. Consider the match degree matrix given below:

```
matchParameters(firstList - ParameterList, secondList - ParameterList){
for each fParam in firstList{
        for each sParam in secondList
                matchMatrix[fParam][sParam]= DegreeOfMatch(fParam, sParam)
        if all degrees in matchMatrix[fParam]=Fail
                return Fail
}
findBestParameterAssignment(matchMatrix)
return smallest match degree
}
```

Figure 4.7 Algorithm for matching parameters

$$\begin{array}{c} \\ a1 \\ a2 \\ a3 \end{array} \begin{array}{c} \begin{array}{ccc} b1 & b2 & b3 \end{array} \\ \begin{bmatrix} exact & plugin & fail \\ plugin & fail & fail \\ fail & fail & subsume \end{bmatrix} \end{array}$$

a_1 matches b_1 and b_2 with degrees "exact" and "plugin". However, choosing the parameter with degree "exact" for assignment causes the overall match to fail, since no parameters are left for a_2. So, the parameter assignment method must find the best assingment set providing no parameter is left unmatched. This problem is an instance of assignment problem, which is about matching things that belong in two separate sets in a way that maximizes (or minimizes) the sum of some quantities. These quantities represent the benefits (or costs) associated to each matching. Each object in one set can be matched with only one object from the other set. Hungarian (Munkres') assignment algorithm is one of many algorithms that solves the assignment problem. Since it reduces the complexity of the problem which is O(n!) to $O(n^3)$, an implementation of Hungarian algorithm, taken from [51], is used in this work.

Two outputs of the algorithm in Figure 4.7 are the parameter assignment list and the match degree which is the smallest match degree among the ones associated to the matchings in the assignment list. This match degree corresponds to *output match degree* when comparing outputs and *input match degree* when comparing inputs.

The algorithm for calculating the degree of match between two parameters is given in Figure 4.8. A parameter type can either be a simple data type (integer, string, etc.) or an ontology concept. When both are data types, the similarity degree between two types is calculated by DataTypeMatcher of the Schema Matcher which will be explained in the next section. When both are concepts, the subsumption relation between them is found in case they belong to the same ontology. Otherwise, semantic similarity degree is calculated. In case a parameter in the first list matches two parameters in the second due to type similarity, parameter names are compared to distinguish the closer one from the other.

```
degreeOfMatch(fParam - Parameter, sParam - Parameter){
fType=fParam.type, sType=sParam-type
if fType=sType return Exact
if both fType, sType are DataTypes
        return compareDataTypes(fType, sType)
if both fType, sType are Concepts
        if belong to same Ontology
                return findSubsumptionRelation(fType, sType)
        else return findSemanticSimilarity(fType, sType)
return Fail
}
```

Figure 4.8 Calculation of the match degree

4.4. Schema Matcher

Schema matcher compares two ontology classes and calculates the semantic similarity degree of them in the interval [0,1]. It consists of *the matcher library*, *the match composer* and *the concept directory*. The architecture of the Schema Matcher component is given in Figure 4.9. Concept Directory keeps the concepts that describe the input/output parameters of the services. Matcher Library consists of a set of matchers, which are used to compare different aspects of the concepts. Finally, the Match Composer executes matchers and combines their results based on a given policy.

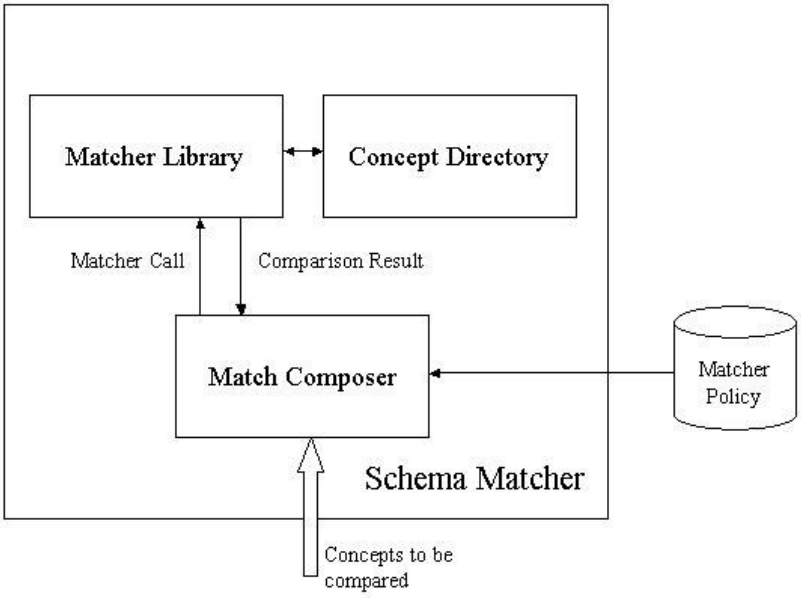

Figure 4.9 Schema Matcher architecture

4.4.1. Concept Directory

The concepts that describe service parameters are kept in Concept Directory with their contexts. When matching two concepts, not only the labels denoted by the concepts but also the contextual features of the concepts constitute an important source of information. In most of the schema matching works, all or a subset of the contextual features are used [34, 15, 20, 7]. Contextual features refer to the properties and the other concepts that the concept has a semantic relation with. The context of a concept c is defined as:

$$Ctx(c) = P(c) \cup C(c) \tag{4.1}$$

P(c) is the set of properties appearing in property constraints related to concept c. Each $p \in P(c)$ is a 4-tuple of the form $p=(n_p, v_p, k_p, t_p)$, where n_p is the name, v_p is the value, $k_p \in \{0,1\}$ is the minimal cardinality and $t_p \in \{dt, ob\}$ is the type of the property . *dt* stands for data type property and *ob* stands for object type property. When t_p=dt, v_p is one of the data types listed in [61] or a user defined data type. When t_p=ob, v_p is another concept. The properties with k_p =0 are called weak properties and k_p =1 are called strong properties.

C(c) is the set of concepts c has semantic relation with. This includes all ancestors and successors of c, derived recursively from the concept hierarchy of the ontology, and the declared equivalent classes.

When adding a concept to the concept directory, the schema fragment that contains the concept and its context is converted to the internal model. During conversion, similar to H-Model [7], OWL class declarations are abstracted into *concept vertices*, OWL data type and object property restrictions are abstracted into *property vertices*, and OWL class relations and operators, that are permitted by OWL-Lite, are abstracted into *semantic links*. Semantic links provided are *eqClass* and *subclass* that correspond to EquivalentClass and subClassOf relations of OWL-Lite. In addition, the intersectionOf operator is abstracted by means of subClass

relation. For example A ≡ B ∩ C is converted into two semantic links A subClass B and A subClass C.

4.4.2. Matcher Library

The results of the works in schema matching domain showed that a single approach to schema matching problem is not enough for a complete solution. This has led to the usage of different combinations of algorithms exploiting different techniques on different parts of the schemas. The same approach is adopted in this thesis and a matcher library composed of different kind of matchers is implemented.

4.4.2.1. Simple Matchers

Concept and property names are important sources of information when calculating similarity between concepts. Names can be compared syntactically or semantically. Three approximate string matching techniques; *affix, n-gram* and *edit distance* are implemented for syntactically matching names. *SemanticMatcher,* on the other hand compares concepts based on their meanings.

Data types are another source of information especially when comparing data type properties of the concepts. *DataTypeMatcher* deals with types.

SemanticMatcher

The semantic matcher finds a similarity degree depending on the meanings of the concepts. To capture the meanings of the concepts, an external thesaurus that consists of terms and terminological relations among them is needed. In this work WordNet [46], which is a lexical database providing a large repository (nearly 75.000 concepts) of English lexical items, is used as the external thesaurus. WordNet is accessed using JWNL(Java WordNet Library)[25].

In WordNet, minimum set of the related concepts is ‘synonym set’ or ‘synset’. This set contains the definitions of the word sense, an example sentence and all the word forms that can refer to the same concept[19]. The synsets are organized into tree-like structures where nodes are linked by semantic relations between them.

Hypernym (generalization of), hyponym (kind of), troponym(way to) and meronym (part/substance/member of) are some examples of semantic relations. A full table of relations is included in [19]. There are nine main hierarchies for nouns, 628 for verbs in WordNet.

There are many methods for determining similarity between two concepts. In [19], these methods are explained in detail and a comparison of them is given. The test results of the work showed that Wu and Palmer's conceptual similarity when used with tagged sense approach gives better results.

Wu and Palmer's method is a distance based similarity method. Distance based similarity methods depend on counting edges in a tree or graph based ontology. Wu and Palmer's method uses IS-A hierarchy. The formula of the method is:

$$WuAndPalmer = \frac{2 \bullet depth(lso(c_1, c_2))}{depth(lso(c_1, c_2)) + \underbrace{len(c_1, lso(c_1, c_2)) + len(c_2, lso(c_1, c_2))}_{len(c_1, c_2)}}$$

where $lso(c_1,c_2)$ is the lowest super-ordinate of synset c_1 and synset c_2, $len(c1,c2)$ is the length of the shortest path from synset c_1 to synset c_2 and $depth(c)$ is the depth of synset c, in other words length of shortest path from synset c to root.

When comparing two words, the similarity can be evaluated for combinations of either all senses or a subset of them. WordNet evaluates the frequencies of the senses depending on a corpus and tags the senses. Using only tagged senses for evaluation not only increases performance but also improves the accuracy of the results.

Wu and Palmer with tagged senses approach is adopted and used for nouns in this work. The algorithm is given in Figure 4.10. For verbs, adjectives and adverbs only synonym relation is used. Since there are many verb hierarchies, no exact paths between verb concepts exist [19].

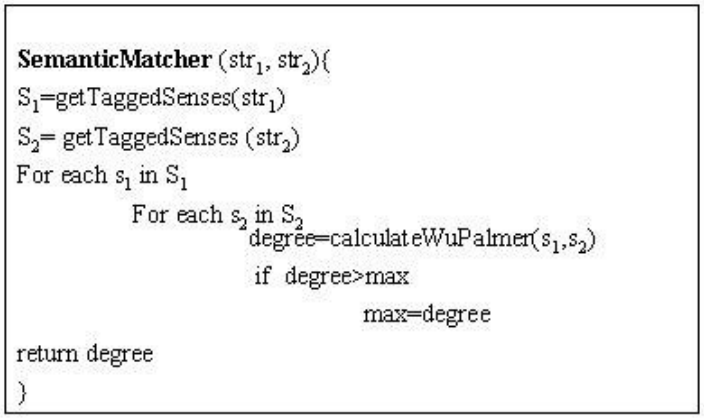

Figure 4. 10 SemanticMatcher for nouns

Affix

Affix looks for common prefixes and suffixes. If one of the strings starts or ends with the other, it returns a degree of exact match, otherwise returns fail.

n-Gram

n-Gram divides both of the strings into their n-grams (sequences of n characters) first. For example, trigrams for the word *computer* are *com, omp, mpu, put, ute, ter*. Then counts the number of common ones. The similarity degree is calculated by dividing the obtained number with *max(numberOfNgrams(str_1), numberOf-Ngrams(str_2))*.

EditDistance

EditDistance calculates the edit distance measure between two strings. It calculates the number of operations (delete, insert and replace) needed to convert one string into the other. The similarity degree is calculated by dividing the obtained number with max(len(str_1),len(str_2)).

DataTypeMatcher

DataTypeMatcher looks up a data type similarity matrix for the similarity of two data types. The data type similarity matrix gives similarity degree between data types that are listed in [61]. Similarity degrees in the table are determined using the type hierarchy given in [4].

4.4.2.2. Hybrid Matchers

Hybrid matchers find similarity degree by combining one or more simple matchers. Five hybrid matchers are implemented namely StringMatcher, ChildrenMatcher, AncestorMatcher, ContextMatcher and Leaf Matcher. The first deals with string matching and the rest deal with a subset of the context whose definition is given in (4.1) (See section 4.4.1).

StringMatcher

StringMatcher finds the similarity degree between two concepts or property names by using semantic and syntactic matchers (Figure 4.11). The names may be basic or compound. The basic ones represent one concept in the thesaurus, where compound names are composed of more than one basic concept and have to be decomposed. The *tokenizer* parses the compound names into tokens using punctuation, digits, special symbols and upper cases, omits common words and if an acronym finder is attached, expands abbreviations and acronyms. StringMatcher compares each token of the first name with the tokens of the second and finds the maximum similarity degree for each. It first looks for a semantic similarity between two strings. A return value of “fail” results with a search in syntactic aspects. The results of the syntactic matchers are combined using the combination strategy selected among the ones explained in section 4.4.3. The final similarity degree is the average of the similarity degrees of the tokens.

```
StringMatcher(str1, str2){
T1=tokenize(str1)
T2=tokenize(str2)
for each t1 in T1{
        max=0
        for each t2 in T2{
                results=nameSimilarity(t1,t2)
                if result>max max=result }
        results[t1]=max }
return average(results)
}
```

```
NameSimilarity(str1, str2){
if  str1 equals str2
        return exact
degree=semanticMatcher(str1, str2)
if degree equals fail{
        for each matcher in SyntacticMatcherSet
                computeSimilarity (str1, str2)
        degree= combineMatcherResults }
return degree
}
```

Figure 4. 11 Algorithm for String Matcher

ContextMatcher

ContextMatcher implements the intensive matching model given in [7]. This model is defined to consider concept names and a subset of the context whose definition is given in (4.1)(See section 4.4.1). The sub-context is defined as:

$$sCtx(c) = P(c) \cup C_s(c) \tag{4.2}$$

where $C_s(c)$ is the set of *adjacent* concepts that the concept has semantic relation with.

In the model, a weight W_{sr} is given to each semantic relation to denote the strength of the connection expressed by the relation on the involved concepts. The greater the weight associated, the higher the strength of the semantic connection between concepts. Moreover weak and strong properties are also assigned weights to capture the importance of the property in characterizing the concept. Since strong properties are mandatory, they have higher weight. The weights used are listed in Table 4.1.

Table 4. 1 Weights associated with contextual features

Context Element	Weight
EqClass	1.0
subclass,superClass	0.8
strong property	1
weak property	0.7

The weights of the contextual features are considered using property and relation closeness function $C(e_1, e_2) \rightarrow [0,1]$ which calculates a measure of distance between two elements of context (two properties, two semantic relations or one property and one semantic relation). The formula of the function is given as:

$$C(e_1, e_2) = 1 - |W_{e1} - W_{e2}| \tag{4.3}$$

The model considers concept names, the whole context of concept and also property values. The similarity degree of two names is as follows in this implementation:

$$A(n_1, n_2) = StringMatcher(n_1, n_2) \tag{4.4}$$

Each element $e_i \in sCtx(c_1)$ is compared against all elements $e_j \in sCtx(c_2)$ using (4.3) and (4.4) and the best matching value $m(e_i)$ is defined as below:

$$m(e_i) = \max\{A(n_{ei}, n_{ej}) \bullet C(e_i, e_j)\}, \forall e_j \in sCtx(c_2) \tag{4.5}$$

In addition to the features given above, the model takes the value of the properties into account and calculates a matching value $k(p_i)$ for each property $pi \in P(c)$. The matching value $k(p_i)$ is defined as:

$$k(p_i) = \begin{Bmatrix} \max\{A(n_{pi}, n_{pj}) \bullet T(v_i, v_j)\}, \forall p_j \in P(c_2) & iff\ t_{pi} = dt \\ \max\{A(n_{pi}, n_{pj}) \bullet A(n_{vpi}, n_{vpj})\}, \forall p_j \in P(c_2) & iff\ t_{pi} = ob \end{Bmatrix} \quad (4.6)$$

$T(n_{vi},n_{vj})$ is the similarity of two types and defined as below in this implementation:

$$T(t_1, t_2) = DataTypeMatcher(t_1, t_2) \quad (4.7)$$

Given two concepts c1 and c2, the model calculates the semantic similarity degree (SSD_{ctx}) as the weighted sum of name similarity calculated using (4.4) and contextual similarity which is the average of (4.5) and (4.6):

$$SSD_{ctx}(c1, c2) = W_l \bullet A(n_{c1}, n_{c2}) + (1 - W) \bullet \frac{\sum_{i=1}^{|sCtx(c)|} m(e_i) + \sum_{j=1}^{|P(c)|} k(p_j)}{|sCtx(c)| + |P(c)|} \quad (4.8)$$

AncestorMatcher

The schema matcher component, as stated above is responsible for calculating a similarity degree between two concepts that denote service input/output parameters. It is assumed that a concept, which denotes a parameter, is most probably closer to leaves rather than root in class hierarchy. AncestorMatcher, specific to this domain, depends on this assumption and claims that: *"a concept, besides the label that denotes it, is defined by its ancestors and the properties, that it owns and it inherits"*. So given two concepts c_1 and c_2, the algorithm calculates the semantic similarity degree(SSD_{anc}) as the weighted sum of name similarity calculated using (4.4) and structural similarity which is the average of (4.10) and (4.6):

$$SSD_{anc}(c1, c2) = W_l \bullet A(n_{c1}, n_{c2}) + (1 - W) \bullet \frac{\sum_{i=1}^{|Anc(c)|} m(a_i) + \sum_{j=1}^{|P_{all}(c)|} v(p_j)}{|Anc(c)| + |P_{all}(c)|} \quad (4.9)$$

where Anc(c) is the set of all ancestors derived recursively by traversing concept hierarchy and $P_{all}(c)$ is the set of all properties concept c owns and inherits. Each element $a_i \in Anc(c_1)$ is compared against all elements $a_j \in Anc(c_2)$ using (4) and the best matching value $m(a_i)$ is defined as below:

$$m(a_i) = \max\{A(n_{ai}, n_{aj})\}, \forall a_j \in Anc(c_2) \tag{4.10}$$

ChildrenMatcher

ChildrenMatcher, inspired from [15], determines the similarity between two concepts based on the combined similarity between their child elements, which in turn can be another concept or a property. The similarity between child elements depends on the type of the element. If both are property, either (4.4) and (4.7) or just (4.4) are used depending on the type of the first property. If one of them is property and the other is concept, the similarity is the product of (4.3) and (4.4). If both are concept, the similarity needs to be computed recursively. The similarity of two child elements is defined as:

$$cSim(ch_i, ch_j) = \begin{cases} SSD_{chd}(ch_i, ch_j) & iff\ ch_i \in S(c_1)\ and\ ch_j \in S(c_2) \\ A(ch_i, ch_j) \bullet C(ch_i, ch_j) & iff\ ch_i \in P(c_1)\ and\ ch_j \in S(c_2)\ or\ vice\ versa \\ A(ch_i, ch_j) \bullet T(v_{chi}, v_{chj}) & iff\ ch_i \in P(c_1)\ and\ ch_j \in P(c_2)\ and\ t_{chi} = dt \\ A(ch_i, ch_j) \bullet A(n_{vchi}, n_{vchj}) & iff\ ch_i \in P(c_1)\ and\ ch_j \in P(c_2)\ and\ t_{chi} = ob \end{cases} \tag{4.11}$$

where $ch_i \in Chd(c_1)$, $ch_j \in Chd(c_2)$ and Chd(c) is the union of P(c) and S(c), which is the set of all adjacent subclasses of concept c.

Each element $ch_i \in Chd(c_1)$ is compared against all elements $ch_j \in Chd(c_2)$ using (11) and the best matching value $m(ch_i)$ is defined as below:

$$m(ch_i) = \max\{cSim(ch_i, ch_j)\}, \forall ch_j \in Chd(c_2) \tag{4.12}$$

Finally, given two concepts c_1 and c_2, the algorithm calculates the semantic similarity degree(SSD_{chd}) as the weighted sum of name similarity calculated using (4.4) and children similarity which is the average of (4.12):

$$SD_{chd}(c_1, c_2) = W_l \bullet A(n_{c1}, n_{c2}) + (1 - W_l) \bullet \frac{\sum_{i=1}^{|Chd(c)|} m(ch_i)}{|Chd(c)|} \tag{4.13}$$

Leaf Matcher

Leaf matcher, which is adopted from [34], determines the similarity of two concepts mostly based on their leaf similarity. The rationale behind this approach is to match concepts which have moderate structural differences (e.g. nesting elements) but have the same data content (similar leaves). However in order not to neglect the extreme differences and strong similarities in the concept structures, the similarity degree of upper layers is reflected to the leaves somehow. In Figure 4.12, the algorithm of the matcher is given.

The algorithm compares two trees, $T(c_1)$ and $T(c_2)$ which are rooted at c1 and c2. T(c) is composed of the concept c, its property vertices and all the subtrees rooted by concepts which are linked to object properties of concept c.

Initially, the structural similarity degrees between the leaves of both trees are calculated. A leaf node can be either a data property node or a concept node with no properties. The initial similarity degree between two leaves is defined as:

$$LFSIM(l_1, l_2) = \begin{cases} 1 & l_1, l_2 \text{ are concepts} \\ 0 & l_1 \text{ is concept}, l_2 \text{ is property} \\ & \text{or vice versa} \\ T(v_{l1}, v_{l2}) & l_1, l_2 \text{ are properties} \end{cases}$$

```
LeafMatcher(t(c1), t(c2)){
For each s ∈ Leaves(t(c1)) and t ∈ Leaves(t(c2))
        ssim(s,t)=LFSIM(s,t)
T1= PostTraverse (t(c1))
T2= PostTraverse (t(c2))
for each s in T1
        for each t in T2 {
                ssim(s,t) = SSD(s,t)
                wsim(s,t) = W1 * LSIM(s,t) +(1- W1)*.ssim(s,t)
                if wsim(s,t) > thhigh
                        increase-struct-similarity(leaves(s),leaves(t),cinc)
                if wsim(s,t) < thlow
                        decrease-struct-similarity(leaves(s),leaves(t),cdec) }
wsim(c1, c2)=W1 * LSIM(s,t) +(1- W1)* SSD(s,t)
return wsim(c1, c2)
}
```

Figure 4. 12 Leaf Matcher algorithm

The definition of $T(v_{11}, v_{12})$ is given in (7). Afterwards, for each node in both trees structural similarity degree is calculated. The structural similarity between two nodes is defined as:

$$SSD(s,t) = \begin{cases} LFSIM(s,t) & if\ \ s,t\ are\ leaves \\ NLFSIM(s,t) & if\ \ s,t\ are\ non-leaf \\ & and\ \left|leaves(s)\right| / \left|leaves(t)\right| \le 2, or\ vice\ versa \\ 0 & otherwise \end{cases}$$

where NLFSIM(s,t) corresponds to non-leaf structural similarity. When a leaf node in the one tree has a weighted similarity degree (wsim) above a threshold with a leaf node in the other, it is said that the first node has *strong link* to the second. The structural similarity of two non-leaf nodes depend on the number of leaves which have at least one strong link to some leaf in the other tree and defined as:

$$NLFSIM = \frac{\left|\begin{array}{l}\{x \mid x \in leaves(s) \wedge \exists y \in leaves(t), stronglink(x, y)\} \\ \cup \{x \mid x \in leaves(t) \wedge \exists y \in leaves(s), stronglink(y, x)\}\end{array}\right|}{\left|leaves(s) \cup leaves(t)\right|}$$

The weighted similarity of two nodes is the weighted sum of the linguistic similarity of the node names and the structural similarity of the nodes. If this value exceeds a threshold, the structural similarity of their leaves are increased, otherwise decreased. Finally, the weighted similarity of c1 and c2 is calculated again after reflecting structural similarity degrees of the non-leaf nodes to the leaves.

4.4.3. Match Composer

When finding similarity degree between two concepts, which syntactic and structural matchers will be used and how the execution results will be combined depends on the user specified policy. The policy consists of a set of syntactic and structural matchers, their combination strategies, linguistic weight (W_l) and a threshold value. Each matcher has a state value $\rightarrow$\{on,off\}, which states whether the matcher will be used during similarity calculations and a weight value $\rightarrow$(0,1).

The combination strategy can be one of *max, min, average* or *weighted*. When *max* is selected, maximum similarity value, when *min* is selected, minimum similarity value among the results of matchers is returned. *Weighted* provides the user with the opportunity to give weights to matchers depending on their performance and returns weighted sum of the results. Finally, *average* returns the average similarity over all matchers.

Given two objects to be matched, a set of matchers and a combination strategy, match composer executes matchers on the objects in turn and combines the results depending on the strategy. The objects can be either two strings or two concepts.

4.5. SSSM System Integration with Event Calculus Executer

In [2], Event Calculus is used for executing OWL-S composite processes as described in Chapter 2. The discovery of the Web services that carry out the atomic tasks in the composite processes is assumed to be pre-discovered. As a future work, SSSM will be integrated with the Event Calculus executer in order to provide a complete solution. The final system will execute generic procedures which fulfill user requests based on the user's preferences and constraints. The generic procedures are expressed as OWL-S composite procedures whose atomic processes has no associated grounding.

The overall system architecture is given in Figure 4.13. The system provides a set of generic procedures which are assumed to be pre-generated either manually or by the Event Calculus planner using domain ontologies that are kept in the system. The user selects one of them for execution via the *User Interface*. The constraints and preferences can either be stated by the user explicitly or can be derived from user profile. The name of generic procedure is passed to the *Controller* which manages other modules. The Controller passes the generic procedure name to the Requester for discovery at first. The *Requester* decomposes the composite processes in the procedure into atomic processes recursively and generates an OWL-S profile definition for each. Afterwards, the discovery phase starts in which each profile definition is sent to SSSM system. SSSM system sends the services conforming to the request described in each profile, with the parameter assignments and match degrees. This information is kept for later selection and execution. After the completion of discovery phase, the Controller invokes the Event Calculus Executer. The *Event Calculus Executer* executes the generic procedure based on user preferences and contraints. During execution, whenever an external service call is required, it refers to Service Caller. The *Service Caller* selects the service with the highest match degree among the ones discovered before and the interaction with the concrete service begins. In case of a failure, the next highest ranked service is

chosen. The Service Caller must have the capability to map the concepts that describe the parameters of the request and advertisement if they are not same but semantically similar.

In this thesis, besides the SSSM system, a simple command driven user interface and the Requester module are implemented for evaluating the performance of the SSSM system. The rest are left as future work.

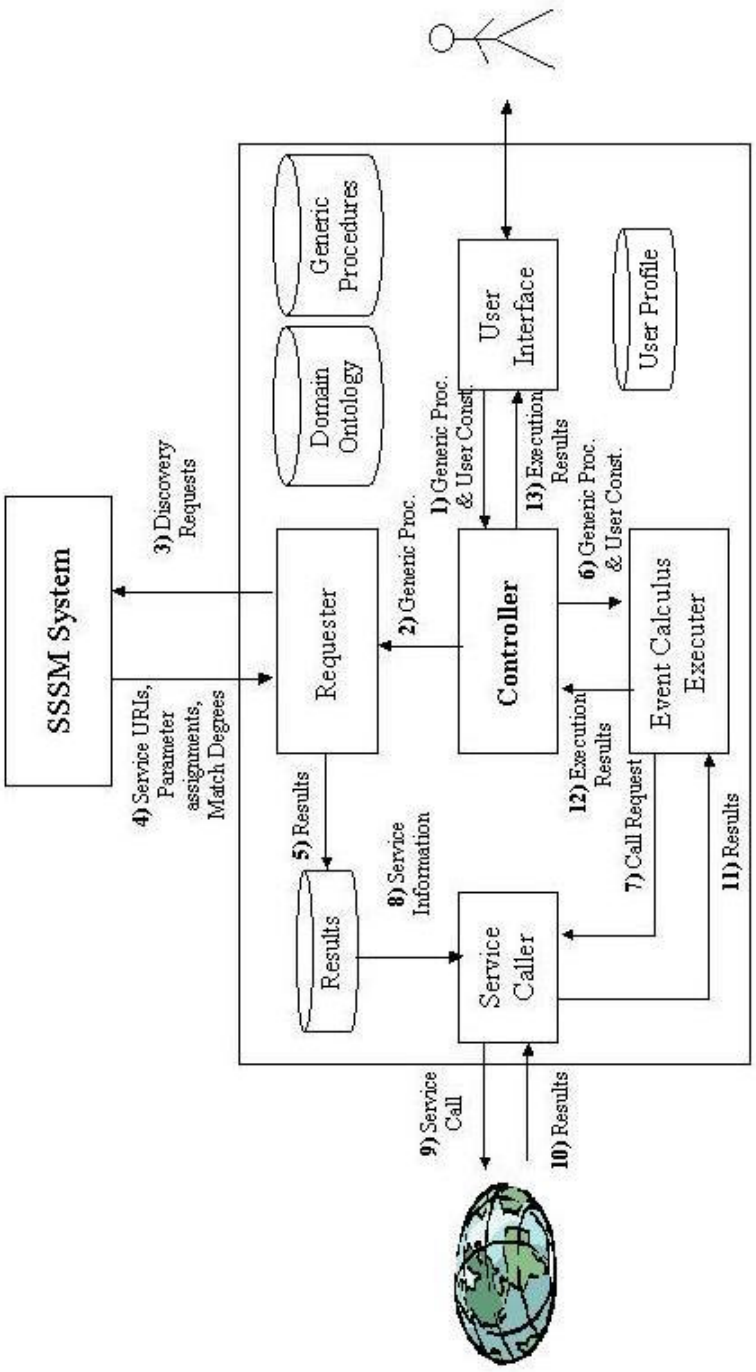

Figure 4. 13 Overall System Architecture

CHAPTER 5

SYSTEM EVALUATION

The main goal of this thesis was to improve semantic Web service discovery methods by proposing a schema matcher extension. In order to conclude whether the goal is achieved or not, the system must be evaluated. During evaluations, the Web service composition system depicted in Figure 4.13 is used as the service requester. This chapter presents the results of the evaluation. It is organized as follows: Section 5.1 describes the test data. Section 5.2 presents the test results. Finally, Section 5.3 discusses whether the goal is reached or not.

5.1. Test Data

The composition system is assumed to make travel arrangements on behalf of the user based on user preferences and constraints. It contains a set of generic procedures for trip arrangements. The procedure given in Figure 5.1 is used to arrange a personal trip for the user. It first makes a hotel reservation and depending on the result of the reservation, makes flight arrangements and rents a car. Finally, it finds the entertainment activities that will be held in the destination city during the vacation period.

The Requester decomposes the whole generic procedure into seven atomic processes namely SearchHotel, SearchFlight, SearchCar, ReserveFlight, BookHotel,

RentCar and FindEntertainment. It generates OWL-S profile descriptions for each. In order to describe the needed capabilities, OTA Hotel Message Ontology and OTA Air Message Ontology from SATINE project are used for hotel and flight arrangement services, respectively. These ontologies are mostly flat, no class hierarchy exists. For describing car arrangement and entertainment search services, Transportation Ontology and Activity Ontology, which are developed for this thesis, are used. These ontologies have a more layered structure.

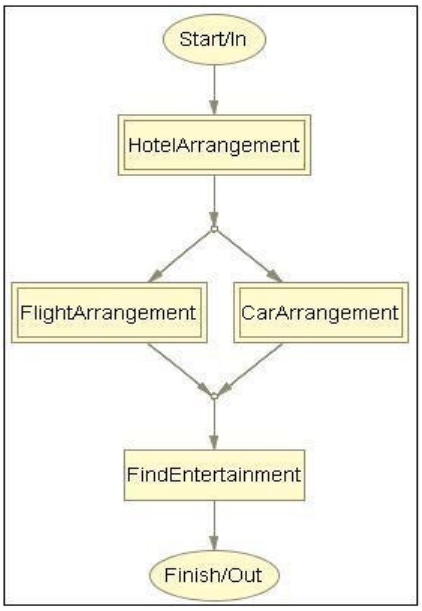

Figure 5.1 The Generic Procedure for Trip Arrangement

The SSSM service base contains 24 services. The advertisements for SearchHotel, SearchFlight, SearchCar and ReserveFlight are provided by the SATINE Project [15] and all the generated requests for these services use different ontologies. Four of the six advertisements provided for SearchCar and RentCar use different ontology from the request where the last two share the same ontology with

the request. Finally, the advertisements provided for FindEntertainment and the request all share the same ontology.

In Figure 5.2, the profile description generated for SearchHotel and one of the advertisements provided for searching available hotels are given with the ontology fragments used to describe them, as an example. The semantic similarity degrees calculated by the Schema Matcher for the input and output parameters of the request and advertisement are 0.77 and 0.82, respectively.

During tests the *threshold* value is set to *0.6* which is the value usually chosen in schema matching proposals. The combination strategy chosen is Max, which selects the highest similarity degree among the ones calculated by different matchers.

5.2. Test Results

The test results are shown in Table 5.1. The table gives the recall and precision values for each matcher and combination of several matchers.

Table 5.1 Matcher Performances

Matcher	Number of Relevant Services	Number of Services Retrieved	Number of Relevant Services Retrieved	Recall (%)	Precision (%)
Ancestor	20	18	14	70	77,78
Child	20	9	9	45	100
Context	20	10	10	50	100
Leaf	20	14	13	65	92,86
Leaf-Context	20	16	15	75	93,75
Leaf-Ancestor	20	21	17	85	80,95
Leaf-Ancestor-Context	20	21	17	85	80,95
OWL-S MatchMaker	20	4	4	20	100

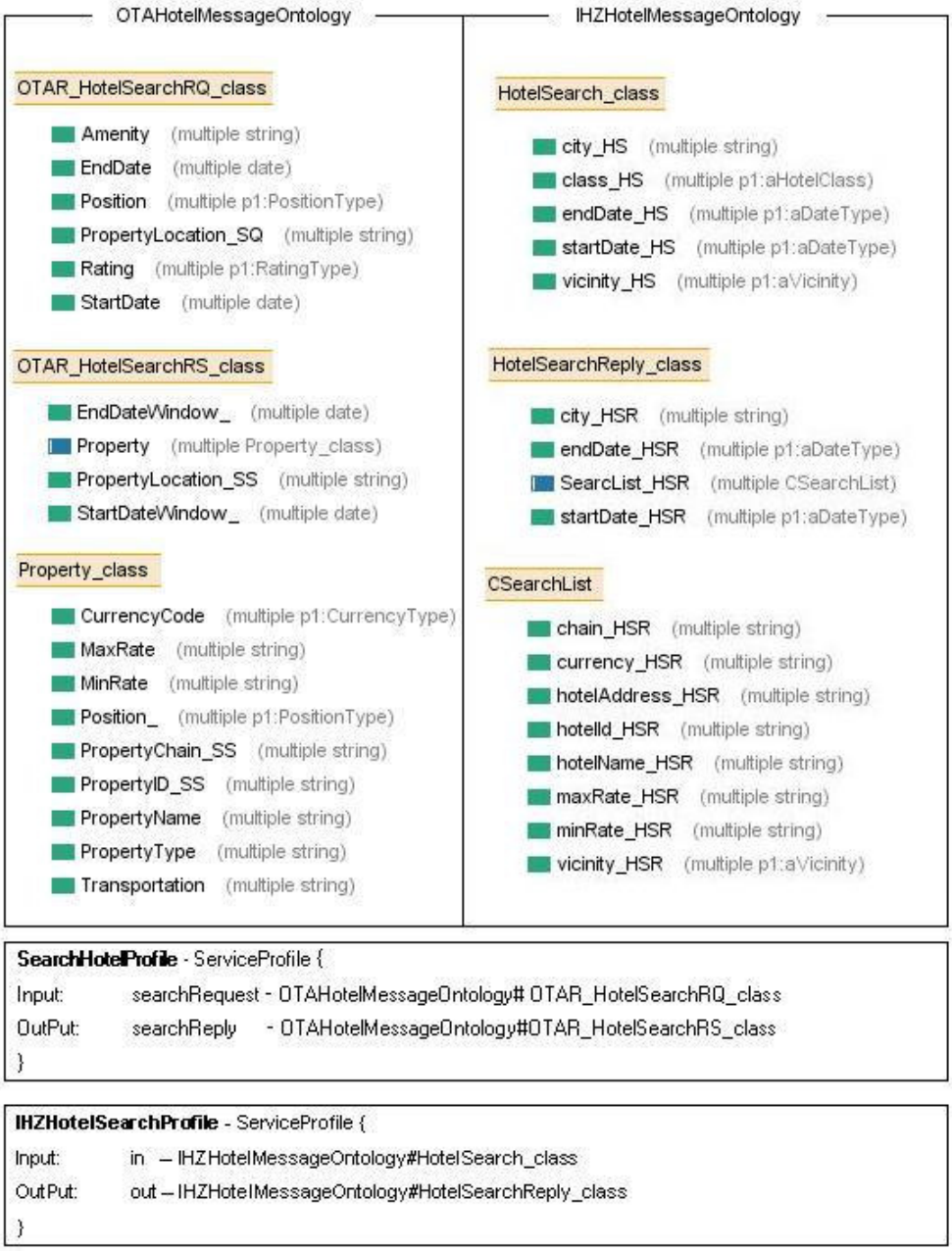

Figure 5. 2 Description of SearchHotel and One of the Advertisements

At the last row, the test results for the MatchMakers approach are given. Since Matchmaker is extended in this work and the closest system to SSSM, the SSSM system is just compared to it.

The *Context* matcher gives the best precision value, however it can retrieve only the half of the related items. Leaf-Ancestor and Leaf-Ancestor-Context combination, on the other hand, give the highest recall value, but their precision value is far from acceptable for automation. *Leaf-Context* combination is the most preferable among them since it can retrieve 75% of the related items and its precision value is near to acceptable.

The OWL-S MatchMaker approach can retrieve only the 20% of the relevant services which share the same ontology with the request.

Table 5.2 gives recall and precision values of the Leaf-Context combination for each atomic service. When the execution of the system and the test data is inspected closely, four major reasons are found, two for low recall and two for low precision values. The first is the Semantic Matcher whose capability directly depends on the WordNet. Since WordNet is a general purpose lexical database, it is insufficient in capturing domain specific semantic relations. For instance, “origin - departure” and “destination - arrival” are used interchangeably in advertisements and request. However, Semantic Matcher calculates the similarity degree as 0.44 for the former and 0.15 for the latter. When a domain specific synonym finder is attached which contains an entry like “pickup - start”, the results get better as in services related to the arrangement of car rental.

The second reason is the data content in which property names are led by codes that relate the property to the concept. For instance, when comparing “name_MAR” with “name”, the first is tokenized as “name” and “MAR”. Each token is compared to “name”. Since MAR has no correspondent the final similarity degree is calculated as 0.5, where it must be 1. The match results are more successful in comparisons where these codes are not used.

The first reason for low precision values is inherited from the approach which the SSSM is based on. Since the approach reduces the capability matching to input/output matching, the services with similar parameters but different functionalities are sometimes retrieved.

The second reason is related with the String Matcher which tokenizes the compound words in a straightforward manner, depending on capital letters, digits, etc. Such a tokenization sometimes can not capture the semantic relation between words that constitute the compound word. For instance, firstName is a specialization of name. However when comparing “name” to “firstName” , the result is calculated as 1 which must be a smaller value.

Table 5.2 Leaf and Context matches with combination strategy Max

Service Request	Number of Relevant Services	Number of Services Retrieved	Number of Relevant Services Retrieved	Recall (%)	Precision (%)
SearchHotel	3	2	2	66,67	100
BookHotel	3	3	3	100	100
SearchFlight	3	1	1	33,33	100
ReserveFlight	3	2	1	33,33	50
SearchCar	3	3	3	100	100
RentCar	3	3	3	100	100
FindEntertainment	2	2	2	100	100
TOTAL	**20**	**16**	**15**	**75**	**93,75**

5.3. Discussion

The goal of the SSSM approach is to improve the recall values of the semantic discovery methods. However, since the requester of the system is an

automated Web service composition system, a higher precision value is as important as a higher recall value for the system to produce the proper results. So it is important to improve the recall value without worsening the precision value. The test results showed that despite a decrease in precision values, number of relevant services retrieved increases 55%.

Besides recall and precision, the total time spent for discovery is also important. In this thesis, performance considerations are neglected. However, before using the SSSM system in a real environment, these problems must be addressed. The first problem is the time spent for string comparison. Schema matching mainly depends on comparison of concept and property names. For comparing two strings, semantic name matcher, which depends on WordNet, and the syntactic matchers are used in turn. The semantic name matcher spends 30 ms for the best case, where two strings are equal and 120 ms for the worst case where there exists no relation between the strings in WordNet. Even the best case value is too high, since the comparison of a concept, which has three properties, with itself takes at least 1380 ms (under the assumption that all the names are atomic). Exploiting previous comparison results via a cache reduces WordNet references for previously compared strings, however the total time spend is still too high.

Another problem is using Schema Matcher for every concept that belong to two different ontologies. This sometimes leads to comparison of totally unrelated concepts and causes waste of time.

Although not at the desired level, it is shown that the initial goal for implementing SSSM system is achieved. The reasons for the failure points are listed in previous section. Before using the system in a real environment, it must be improved to handle these failures and its performance problems must be addressed.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Web services provide businesses an efficient and faster way of reaching consumers, and conducting business with other partners. WSDL and UDDI, which are current industry standards, provide a standard way of describing, advertising and discovering services. With the widespread usage of the Web services, the user or business requirements get more complex and can be handled only by composing several services in a way that produces the desired effect. However, the increase in the number of available services and the dynamic nature of the Web and user requirements, it is getting harder to manually locate the services and integrate them. In the literature, several solutions, which exploit AI planning algorithms, are proposed to automate the composition process.

Web service discovery, is the most essential phase of the composition process to automate. Automated discovery requires semantically described, machine processible Web services and semantic matching algorithms that are capable of processing these descriptions. WSDL lacks the constructs that reveals out the capabilities of the services and UDDI can not fully capture the semantics of the descriptions. OWL-S, which is the application of Semantic Web constructs to Web services domain, addresses the semantic specification requirement. In the literature, there are several semantic Web service discovery methods, which process semantically enriched descriptions. These proposals exploit ontologies to describe

the capabilities of services in different ways, however they all assume that the relevant advertisements and the requests share the same ontology.

In this thesis, an extension is proposed to the matching approach used in OWL-S MatchMaker in order to handle ontology differences. OWL-S MatchMaker exploits ontologies to describe the input/output parameters of the services and reduces the capability matching to input/output matching. It proposes a discrete scale to rank the similarity of two parameters, and where to map the similarity depends on the subsumption relation between the concepts that describes the parameters. This work extends this scale to cover not only the concepts which have subsumption relation but also the ones that have semantic similarity, since they describe similar real world entities. In addition, a schema matcher component, which exploits different schema matching algorithms to calculate the semantic similarity between two components, is proposed. The new approach is called Semantic Service Discovery with Schema Matcher (SSSM).

It is shown that SSSM achieves the initial goal: It retrieves the services which OWL-S MatchMaker fails to and increases the recall value. However, the precision value falls due to some mismatches. The first of the two reasons for the deficiency, related with the SSSM system, itself, is the usage of a general purpose thesaurus like WordNet, which is incapable of capturing domain specific semantic relations. The second reason is the String Matcher which lacks capturing the semantic relation between the words that constitute a compound word.

As a future work, the String Matcher may be improved using NLP techniques to match concept and property names better.

The performance considerations are neglected in this work. However, schema matching is a very expensive process and must be held as efficiently as possible. Therefore, another future work may focus on improving the performance of the system. One improvement may be implementing ontology classification system and integrating it to SSSM, in order to avoid comparing extremely different concepts. The advertisement ontologies can be classified during registry and

comparison can be done only between two concepts that are classified under the same category. Another improvement may be, dynamically selecting a matcher based on the structure of concepts instead of executing the matchers in a fixed manner and avoiding executing several matchers. Exploiting smaller domain ontologies instead of WordNet, which is a huge general purpose ontology for string comparison may also improve performance results.

Another future work is the integration of this system to the Event Calculus planner implemented in [2] which lacks web service discovery mechanisms. The integration is described in Section 4.5 in detail and the general view of the resulting system is given in Figure 4.13. In [2], the generic procedures are assumed to be pre-discovered. An improvement to the system may be using the Event Calculus planner to generate a procedure corresponding to user request expressed in a high level way.

REFERENCES

1 Aumueller, D., Do, H., Massmann, S. & Rahm, E. (2005). Schema and ontology matching with COMA++. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (906-908). Baltimore, Maryland, USA.

2 Aydin, O. (2005). *Automated Web service composition with event calculus.* Unpublished master thesis, Middle East Technical University, Turkey.

3 Bernstein, P.A., Melnik, S., Petropoulos, M. & Quix, C. (2004). Industrial-strength schema matching. *SIGMOD Record, 33(4),* 38-43.

4 Biron, P.V., Permanente, K. & Malhotra, A. (2004). *XML schema part 2: datatypes second edition*. W3C Web site: http://www.w3.org/-TR/2004/REC-xmlschema-2-20041028/, visited on April 2005.

5 Brickley, D. &Guha, R.V. (2004). *RDF vocabulary description language 1.0: RDF Schema*. W3C Web site: http://www.w3.-org/TR/rdf-schema/, visited on December, 2005.

6 Broens, T. (2004). *Context-aware, ontology based, semantic service discovery.* Unpublished master thesis, University of Twente, Enschede, The Netherlands.

7 Castano, S., Ferrara, A. & Montanelli, S. (2003). *H-MATCH: an algorithm for dynamically matching ontologies in peerbased systems.* Paper presented at the 1st International Workshop on Semantic Web and Databases (SWDB). Berlin, Germany.

8 Castano, S., Ferrara, A., Montanelli, S. & Racca, G. (2004, August). *From surface to intensive matching of semantic Web ontologies.* Paper presented at the 3rd DEXA International Workshop on Web Semantics, IEEE Computer Society, Zaragoza, Spain.

9 Chinnici, R., Moreau, J.J., Ryman, A. & Weerawarana S. (2006). *Web services description language (WSDL) version 2.0 Part 1, core language*. W3C Web site: http://www.w3.org/TR/2006/CR-wsdl20-20060327, visited on April, 2006.

10 Christensen, E., Curbera, F., Meredith, G. & Weerawarana, S. (2001). *Web services description language (WSDL) 1.1.* W3C Web site: http://www.w3.org/TR/2001/-NOTE-wsdl-20010315, visited on September, 2005.

11 Clement, L., Hately, A., Riegen, C.V., Rogers, T. (2004). *UDDI Spec Technical Committee Draft*. UDDI Web site http://uddi.-org/pubs/uddi-v3.0.2-20041019.htm, visited on September, 2005.

12 Colgrave, J., Januszewski, K. (2004). *Using WSDL in a UDDI registry, version 2.0.2 - Technical Note*. OASIS Web site: http://www.oasisopen.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v202-20040631.htm, visited on January 2006.

13 David, G. (2005). Understanding structural and semantic heterogeneity in the context of database schema integration. *Journal of Computing@UCLan, 4*, 29-44.

14 Dhamankar, R., Lee, Y., Doan, A., Halevy, A. & Domingos, P. (2004, June). iMAP: Discovering complex matches between database schemas. *Proceedings of ACM SIGMOD International Conference on Management of Data* (383-394). Paris, France.

15 Do, H. & Rahm, E. (2002). Coma: A system for flexible combination of schema matching approaches. *Proceedings of 28th Conference on Very Large Databases (VLDB) (610-621)*. Hong Kong, China.

16 Doan, A., Madhavan, J., Dhamankar, R., Domingos, P., & Halevy, A. (2003). Learning to match ontologies on the semantic Web. *VLDB Journal, 12,* 303-319.

17 Doan, A., Halevy, A. (2005 Spring). Semantic integration research in the database community: A brief survey. *AI Magazine, 26(1), Special Issue On Semantic Integration,* 83-94.

18 Dogac, A., Kabak, Y., Laleci, G., Sinir, S., Yildiz, A., Kirbas, S., et al. (2004). Semantically enriched Web services for the travel industry, *ACM Sigmod Record, 33(3),* 21-27.

19 Erozel, G. (2005). *Natural language interface on a video data model.* Unpublished masters dissertation, Middle East Technical University, Turkey.

20 Giunchiglia, F., Shvaiko, P. & Yatskevich, M. (2004, May). S-Match: An algorithm and implementation of semantic matching. *Proceedings of First European Semantic Web Symposium*, 61-75.

21 Giunchiglia, F. & Yatskevich, M. (2004, November)**.** *Element level semantic matching*. Paper presented at the Third International Semantic Web Conference (ISWC) on Meaning Coordination and Negotiation Workshop . Hiroshima, Japan.

22 He, B., Chang, K.C.C. & Han, J. (2004). *Discovering complex matchings across Web query interfaces: A correlation mining approach.* Proceedings of Tenth ACM SIGKDD Conference (KDD) (148-157). Seattle, Washington, USA.

23 Herman, I. (2006, May). *Semantic Web activity statement.* Presented at the W3C advisory committee meeting, Edinburgh, Scotland, UK.

24 Horrocks, I. Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B. & Dean, M. (2004). *SWRL: A semantic Web rule language combining OWL and RuleML.* W3C Web Site: http://www.w3.org/Submission/SWRL/, visited on June, 2006.

25 Java WordNet Library. SourceForge Net Web Site: http://sourceforge.net/-projects/jwordnet, visited on March, 2006.

26 Jena Web site: http://jena.sourceforge.net/, visited on October 2005

27 Jerami, E. (2002). *Web services essentials.* CA, USA: O'Reilly Press.

28 Kashyap, V. & Sheth, A. (1996). Semantic and schematic similarities between database objects: a context based approach. *The VLDB Journal, 5*, 276-304.

29 Kifer, M., Lara, R., Polleres, A., Zhao, C., Keller, U., Lausen, H. &Fensel, D.(2004). *A logical framework for Web service discovery.* Paper presented at the 3rd International Semantic Web Conference (ISWC2004). Hiroshima, Japan.

30 Klein, M. & Bernstein, A. (2001, July). *Searching for services on the semantic Web using process ontologies.* Paper presented at the First Semantic Web Working Symposium (SWWS-1) Stanford, CA, USA.

31 Klein, M. & Bernstein, A. (2004). Towards high-precision service retrieval. *IEEE Internet Computing,8(1),* 30-36.

32 Komatineni S. (2002). *Understanding UDDI and JAXR*. O'Reilly Web Site: http://www.onjava.com/pub/a/onjava/2002/02/27/uddi.html, visited on September, 2005.

33 Li, L. & Horrocks, I. (2004). A software framework for matchmaking based on semantic Web technology. *International Journal of Electronic Commerce (IJEC), 8(4),* 39-60.

34 Madhavan, J., Bernstein, P. & Rahm, E. (2001, September). Generic schema matching with Cupid. *Proceedings of the 27th International Conference on Very Large Data Bases* **,** 49-58. Rome, Italy.

35 Madhavan, J., Bernstein, P., Chen, K., Halevy, A. & Shenoy, P. (2005). *Corpus-based schema matching.* Paper presented at the International Conference on Data Engineering (ICDE). Tokyo, Japan.

36 Mandreoli, F., Martoglia, R. & Tiberio, P. (2004). *Approximate query answering for a heterogeneous XML document base.* Paper presented at the Fifth International Conference on Web Information Systems Engineering (WISE 2004), Brisbane, Australia.

37 Manola, F. & Miller, E. (2004). *RDF primer.* W3C Web site: http://www.w3.org/TR/rdf-primer/, visited on December 2005.

38 Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., et al. (2004). OWL-S: Semantic markup for Web services. *W3C member submission.* W3C Web site: http://www.w3.org/-Submission/OWL-S/, visited on January 2006.

39 Masood, N. (2004). A schema comparison approach to determine semantic similarity among schema elements. *Information Technology Journal 3(1),* 57-68.

40 McDermott, D. (2002, April). *Estimated-regression planning for interactions with Web services.* Paper presented at the 6th international conference on AI Planning and Scheduling, Toulouse, France.

41 McGuinness, D.L. & Harmelen, F.V. (2004). *OWL Web ontology language overview.* W3C Web site: http://www.w3.org/-TR/owl-features/, visited on January, 2006.

42 McIlraith, S. & Son, T.C. (2002, April), *Adapting GOLOG for composition of semantic Web services.* Paper presented at the 8th international conference on Knowledge Representation and Reasoning (KR2002), Toulouse, France.

43 Medjahed, B., Bouguettaya, A., Elmagarmid, A. K. (2003). Composing Web services on the semantic Web. *The VLDB Journal, 12(4),* 333-351.

44 Melnik S., Molina-Garcia, H. & Rahm E. (2002). *Similarity flooding: a versatile graph matching algorithm.* Paper presented at the International Conference on Data Engineering (ICDE). San Jose CA.

45 Miller, E. (1998 May 26). An introduction to the resource description framework. *D-Lib Magazine Web Site:* http://www.dlib.org/dlib/may98/miller/-05miller.html, visited on December, 2005.

46 Miller, G.A. (1995). WordNet: A lexical database for english. *Communications of the ACM (CACM), 38(11),* 39–41.

47 MINDSWAP Web site: http://www.mindswap.org, visited on December 2005.

48 Mirbel, I. (1995). *Semantic integration of conceptual schemes.* Paper presented at the first international workshop on applications of natural language to data bases, NLDB'95, Versailles, France.

49 Mitra, P., Wiederhold, G. & Jannink, J. (1999). *Semi-automatic integration of knowledge sources.* Paper presented at the 2nd International Conference on Information FUSION. Sunnyvale, USA.

50 MySQL Web site: www.mysql.com, visited on December, 2005.

51 Nedas, K.A. *Munkres' (Hungarian) algorithm.* Retrieved February, 2006, from http://www.spatial.maine.-edu/~kostas/dev/soft/munkres.htm.

52 Paolucci, M., Kawamura, T., Payne, T. & Sycara, K. (2002, June). *Semantic matching of Web services capabilities.* Paper presented at the 1st international semantic Web conference (ISWC), 333-347.

53 Paolucci, M., Kawamura, T., Payne, T. & Sycara, K. (2002). *Importing the semantic Web in UDDI.* Paper presented at the Web Services, e-Business and Semantic Web Workshop.

54 Ponnekanti, S.R. & Fox, A. (2002, May). *SWORD: A developer toolkit for Web service composition.* Paper presented at the 11th worldwide Web conference, Honolulu, HI, USA.

55 Rahm, E. & Bernstein, P. (2001). On matching schemas automatically. *VLDB Journal 10(4),* 334-350.

56 Rahm, E., Do, H.H. & Massmann, S. (2004). Matching large XML schemas. *SIGMOD Record, 33(4),* 26-31.

57 Riegen, C.V. (2002). *UDDI version 2.03 data structure reference*. UDDI Web Site: http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm, visited on January, 2006.

58 Roman, D., Keller, U., Lausen, H., Bruijn, J., Lara, R., Stollberg, M., et al. (2005).: Web service modeling ontology, *Applied Ontology, 1(1)*, 77-106.

59 Saake, G., Sattler, K. & Conrad, S. (2005). Rule-based schema matching for ontology-based mediators. *Journal of Applied Logic, 3,* 253-270.

60 Sivashanmugam, K., Verma, K., Sheth, A. & Miller, j.A. (2003, June). *Adding semantics to Web services standards.* Paper presented at the First International conference on Web services (ICWS'03), Las Vegas, Nevada.

61 Smith, M.K., Welty, C. & McGuinness, D.L. (2004). *OWL Web ontology language guide*. W3C Web site: http://www.w3.org/TR/-2004/REC-owl-guide-20040210/, visited on September, 2005.

62 Srinivasan, N., Paolucci, M. & Sycara, K. (2004, July). *Adding OWL S to UDDI, implementation and throughput*. Paper presented at the First International Workshop on Semantic Web Services and Web Process Composition, San Diego, USA.

63 Su, X. & Rao, J. (2004, July). *A survey of automated Web service composition methods.* Paper presented at the first international workshop on Semantic Web Services and Web Process Composition, San Diego, California, USA.

64 Sycara, K., Paolucci, M., Ankolekar, A. & Srinivasan, N. (2003). Automated discovery, interaction and composition of semantic Web services. *Journal of Web Semantics, 1(1),* 27-46.

65 Trastour, D., Bartolini, C. & Castillo, J.G. (2001, August), *A semantic Web approach to service description for matchmaking of services.* Paper presented at the First Semantic Web Working Symposium. California, USA.

66 Travel Ontology. Web site of Stanford University: http://protege.stanford.edu/-plugins/owl/owl-library/travel.owl, visited on April, 2006.

67 Uschold, M. (2003). Where are the semantics in the semantic Web?. *AI Magazine, 24(3),* 25-36.

68 Wang, Y., Stroulia, E. (2003, December). Flexible interface matching for Web service discovery. *Proceedings of Fourth International Conference on Web Information Systems Engineering* (147-156). Roma, Italy.

69 Wu, D., Sirin, E., Hendler, J., Nau, D., & Parsia, B. (2003, June). *Automatic Web services composition using SHOP2.* Paper presented at the workshop on Planning for Web Services, Trento, Italy.

70 Xu, L. & Embley, D. (2003). *Using domain ontologies to discover direct and indirect matches for schema elements.* Paper presented at the Semantic Integration Workshop collocated with the Second International Semantic Web conference (ISWC-03), Sanibel Island, Florida.

APPENDIX A

WSDL DESCRIPTION OF STOCK QUOTE SERVICE

```
<?xml version="1.0" encoding="utf-8"?>
<description
    targetNamespace="http://example.com/stockquote/"
    xmlns:tns="http://example.com/stockquote/"
    xmlns:ghns = "http://example.com/stockquote/"
    xmlns:wsoap= "http://www.w3.org/2006/01/wsdl/soap"
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsdlx= http://www.w3.org/2006/01/wsdl-extensions
    xmlns="http://www.w3.org/2006/01/wsdl">
<types>
        <xs:schema
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://example.com/stockquote/schema/"
            xmlns="http://example.com/stockquote/schema/">
                <xs:element name=" TradePriceRequest" type="xs:string"/>
                <xs:element name=" TradePrice" type="xs:float"/>
                <xs:element name="invalidSymbolError" type="xs:string"/>
        </xs:schema>
</types>
<interface  name = " StockQuoteInterface" >
        <fault name = "invalidSymbolFault"
            element = "ghns:invalidSymbolError"/>
        <operation name=" GetLastTradePrice"
            pattern="http://www.w3.org/2006/01/wsdl/in-out"
            style="http://www.w3.org/2006/01/wsdl/style/iri"
            wsdlx:safe = "true">
                <input messageLabel="In"
                    element="ghns: TradePriceRequest " />
                <output messageLabel="Out"
                    element="ghns: TradePrice" />
                <outfault ref="tns:invalidSymbolFault" messageLabel="Out"/>
```

```
        </operation></interface>
<binding name="StockQuoteSoapBinding"
    type="http://www.w3.org/2006/01/wsdl/soap"
    interface="tns: StockQuoteInterface"
    wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">
            <fault ref="tns:invalidSymbolFault" wsoap:code="soap:Sender"/>
            <operation ref="tns: GetLastTradePrice"
                wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
</binding
```

APPENDIX B

UDDI ELEMENTS FOR WSDL DESCRIPTIONS

Listing 1: tModel definition for StockQuoteInterface

```
<tModel tModelKey="uuid:e8cf1163-8234-4b35-865f-94a7322e40c3" >
   <name>
      StockQuoteInterface
   </name>
   <overviewDoc>
      <overviewURL>
         http://location/sample.wsdl
      <overviewURL>
   <overviewDoc>
   <categoryBag>
      <keyedReference
         tModelKey="uuid:d01987d1-ab2e-3013-9be2-2a66eb99d824"
            keyName="interface namespace"
            keyValue="http://example.com/stockquote/" />
      <keyedReference
         tModelKey="uuid:6e090afa-33e5-36eb-81b7-1ca18373f457"
         keyName="WSDL type"
         keyValue="interface" />
   </categoryBag>
</tModel>
```

Listing 2: tModel definition for StockQuoteBinding

```
<tModel tModelKey="uuid:49662926-f4a5-4ba5-b8d0-32ab388dadda">
   <name>
      StockQuoteSoapBinding
   </name>
   <overviewDoc>
      <overviewURL>
         http://location/sample.wsdl
      </overviewURL>
```

```
    </overviewDoc>
    <categoryBag>
        <keyedReference
            tModelKey="uuid:d01987d1-ab2e-3013-9be2-2a66eb99d824"
            keyName="binding namespace"
            keyValue="http://example.com/stockquote/" />
        <keyedReference
            tModelKey="uuid:6e090afa-33e5-36eb-81b7-1ca18373f457"
            keyName="WSDL type"
            keyValue="binding" />
        <keyedReference
            tModelKey="uuid:082b0851-25d8-303c-b332-f24a6d53e38e"
            keyName="interface reference"
            keyValue="uuid:e8cf1163-8234-4b35-865f-94a7322e40c3" />
        <keyedReference
            tModelKey="uuid:4dc74177-7806-34d9-aecd-33c57dc3a865"
            keyName="SOAP protocol"
            keyValue= "uuid:aa254698-93de-3870-8df3-a5c075d64a0e" />
        <keyedReference
            tModelKey="uuid:e5c43936-86e4-37bf-8196-1d04b35c0099"
            keyName="HTTP transport"
            keyValue=" uuid:68DE9E80-AD09-469D-8A37-088422BFBC36"/>
        <keyedReference
            tModelKey="uuid:c1acf26d-9672-4404-9d70-39b756e62ab4"
            keyName="uddi-org:types"
            keyValue="wsdlSpec" />
    </categoryBag>
</tModel>
```

Listing 3: businessEntity definition

```
<businessEntity
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="urn:uddi-org:api"
    businessKey="5E4B4CDA-3448-4B88-ACE1-8584D43B7703"
    operator="Microsoft Corporation"
    authorizedName="Andy Grove"> ...
    <name>Cape Clear Software</name>
        ...
    <categoryBag>
        <keyedReference
```

```
            tModelKey="uuid:c0b9fe13-179f-413d-8a5b-5004db8e5bb2"
            keyName="Software Publishers"
            keyValue="5112" />

        <keyedReference
            tModelKey="uuid:4e49a8d6-d5a2-4fc2-93a0-0411d8d19e88"
            keyName="United States"
            keyValue="US" />
    </categoryBag>
</businessEntity>
```

Listing 4: businessService definition for StockQuoteService

```
<businessService
    serviceKey="102b114a-52e0-4af4-a292-02700da543d4"
    businessKey="5E4B4CDA-3448-4B88-ACE1-8584D43B7703">
    <name>Stock Quote Service</name>
    <bindingTemplates>
        <bindingTemplate
            bindingKey="f793c521-0daf-434c-8700-0e32da232e74"
            serviceKey="102b114a-52e0-4af4-a292-02700da543d4">
            <accessPoint URLType="http">
                http://location/sample
            </accessPoint>
            <tModelInstanceDetails>
                <tModelInstanceInfo
                    tModelKey="uuid:49662926-f4a5-4ba5-b8d0-32ab388dadda">
                    <description xml:lang="en">
                        The wsdl:binding that this wsdl:endPoint implements. The
                        instanceParms specifies the endPoint local name.
                    </description>
                    <instanceDetails>
                        <instanceParms>
                            StockQuoteEndPoint
                        </instanceParms>
                    </instanceDetails>
                </tModelInstanceInfo>
                <tModelInstanceInfo
                    tModelKey="uuid:e8cf1163-8234-4b35-865f-94a7322e40c3">
                    <description xml:lang="en">
                        The wsdl:Interface that this wsdl:endPoint implements.
                    </description>
                </tModelInstanceInfo>
```

```
        </tModelInstanceDetails>1
      </bindingTemplate>
    </bindingTemplates>
    <categoryBag>
      <keyedReference
          tModelKey="uuid:6e090afa-33e5-36eb-81b7-1ca18373f457"
          keyName="WSDL type"
          keyValue="service" />
      <keyedReference
          tModelKey="uuid:d01987d1-ab2e-3013-9be2-2a66eb99d824"
          keyName="service namespace"
        keyValue="http://example.com/stockquote/" />
     <keyedReference
          tModelKey="uuid:2ec65201-9109-3919-9bec-c9dbefcaccf6"
          keyName="service local name"
          keyValue="StockQuoteService" />
    </categoryBag>
</businessService>
```

APPENDIX C

OWL-S SERVICE DESCRIPTIONS

Listing 1: Advertisement

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns:service="http://www.daml.org/services/owl-s/1.2/Service.owl#"
   xmlns:profile="http://www.daml.org/services/owl-s/1.2/Profile.owl#"
   xmlns:process="http://www.daml.org/services/owl-s/1.2/Process.owl#"
   xmlns:grounding="http://www.daml.org/services/owl-s/1.2/Grounding.owl#"
   xmlns="http://www.tez.metu.edu.tr/services/FindSportsActivity#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:activity="http://www.tez.metu.edu.tr/ontologies/Activity.owl#"
   xmlns:location="http://www.tez.metu.edu.tr/ontologies/Location.owl#"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
 xml:base="http://www.tez.metu.edu.tr/services/FindSportsActivity">
 <owl:Ontology rdf:about="">
  <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.2/Profile.owl
   "/>
  <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.2/Process.owl"
  />
  <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.2/Service.owl"
   />
  <owl:imports df:resource="http://www.tez.metu.edu.tr/ontologies/Location.owl"/>
  <owl:imports rdf:resource="http://www.tez.metu.edu.tr/ontologies/Activity.owl"/>
 </owl:Ontology>
 <process:AtomicProcess rdf:ID="FindSportsActivityProcess">
  <process:hasInput>
   <process:Input rdf:ID="city">
    <process:parameterType rdf:datatype="http://www.w3.org/2001/ XMLSchema#
        anyURI"> http://www.tez.metu.edu.tr/ontologies/Location.owl#City
     </process:parameterType>
   </process:Input>
  </process:hasInput>
  <process:hasInput>
```

```
    <process:Input rdf:ID="endDate">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#
          anyURI">http://www.w3.org/2001/XMLSchema#date
      </process:parameterType>
    </process:Input>
  </process:hasInput>
  <process:hasInput>
    <process:Input rdf:ID="startDate">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#
          anyURI">http://www.w3.org/2001/XMLSchema#date
      </process:parameterType>
    </process:Input>
  </process:hasInput>
  <process:hasOutput>
    <process:Output rdf:ID="activity">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#
          anyURI">http://www.tez.metu.edu.tr/ontologies/Activity.owl#Sports
      </process:parameterType>
    </process:Output>
  </process:hasOutput>
 </process:AtomicProcess>
 <profile:Profile rdf:ID="FindSportsActivityProfile">
  <profile:hasInput rdf:resource="#startDate"/>
  <profile:hasInput rdf:resource="#city"/>
  <profile:hasInput rdf:resource="#endDate"/>
  <profile:hasOutput rdf:resource="#activity"/>
   <profile:serviceName rdf:datatype="http://www.w3.org/2001/XMLSchema#
      string">FindSportsActivityService</profile:serviceName>
  <service:presentedBy>
      <service:Service rdf:ID="FindSportsActivityService">
          <service:presents rdf:resource="#FindSportsActivityProfile"/>
      </service:Service>
  </service:presentedBy>
  <profile:textDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#
      string">Searches for activities held in a city during a time frame
   </profile:textDescription>
 </profile:Profile>
</rdf:RDF>

<!-- Created with Protege (with OWL Plugin 2.2, Build 320)  http://protege.stan-
ford.edu -->
```

Listing 2: Request

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns:service="http://www.daml.org/services/owl-s/1.2/Service.owl#"
   xmlns:profile="http://www.daml.org/services/owl-s/1.2/Profile.owl#"
   xmlns:process="http://www.daml.org/services/owl-s/1.2/Process.owl#"
   xmlns:grounding="http://www.daml.org/services/owl-s/1.2/Grounding.owl#"
   xmlns="http://www.tez.metu.edu.tr/services/FindActivity#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:activity="http://www.tez.metu.edu.tr/ontologies/Activity.owl#"
   xmlns:location="http://www.tez.metu.edu.tr/ontologies/Location.owl#"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
 xml:base="http://www.tez.metu.edu.tr/services/FindActivity">
 <owl:Ontology rdf:about="">
  <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.2/Profile.owl"
   />
  <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.2/Process.owl"
  />
  <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.2/Service.owl"
   />
  <owl:imports df:resource="http://www.tez.metu.edu.tr/ontologies/Location.owl"/>
  <owl:imports rdf:resource="http://www.tez.metu.edu.tr/ontologies/Activity.owl"/>
 </owl:Ontology>
 <process:AtomicProcess rdf:ID="FindActivityProcess">
  <process:hasInput>
   <process:Input rdf:ID="city">
    <process:parameterType rdf:datatype="http://www.w3.org/2001/ XMLSchema#
        anyURI"> http://www.tez.metu.edu.tr/ontologies/Location.owl#City
     </process:parameterType>
   </process:Input>
  </process:hasInput>
  <process:hasInput>
   <process:Input rdf:ID="endDate">
    <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#
        anyURI">http://www.w3.org/2001/XMLSchema#date
     </process:parameterType>
   </process:Input>
  </process:hasInput>
  <process:hasInput>
   <process:Input rdf:ID="startDate">
```

```
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#
          anyURI">http://www.w3.org/2001/XMLSchema#date
        </process:parameterType>
      </process:Input>
    </process:hasInput>
    <process:hasOutput>
      <process:Output rdf:ID="activity">
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#
            anyURI">http://www.tez.metu.edu.tr/ontologies/Activity.owl#Activity
        </process:parameterType>
      </process:Output>
    </process:hasOutput>
  </process:AtomicProcess>
  <profile:Profile rdf:ID="FindActivityProfile">
    <profile:hasInput rdf:resource="#startDate"/>
    <profile:hasInput rdf:resource="#city"/>
    <profile:hasInput rdf:resource="#endDate"/>
    <profile:hasOutput rdf:resource="#activity"/>
     <profile:serviceName rdf:datatype="http://www.w3.org/2001/XMLSchema#
        string">FindActivityService</profile:serviceName>
    <service:presentedBy>
        <service:Service rdf:ID="FindActivityService">
            <service:presents rdf:resource="#FindActivityProfile"/>
        </service:Service>
    </service:presentedBy>
    <profile:textDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#
        string">Searches for activities held in a city during a time frame
     </profile:textDescription>
  </profile:Profile>
</rdf:RDF>
```