A QUERY COMPOSER AND A COMPOSITE SERVICE SPECIFICATION GENERATOR FOR DYNAMIC WEB SERVICE COMPOSITION

By

LAKSHMI N. CHAKARAPANI

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2004

I dedicate this thesis to my family

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Dr. Stanley Y. W. Su, chairman of my supervisory committee, for providing me with such an interesting thesis topic and for his valuable guidance and support in this research effort. I would also like to express my gratitude to Dr. Herman Lam, my supervisory committee member, for his feedback and guidance during the design and implementation phases of our research work. I would like to thank Dr. Markus Schneider for serving on my supervisory committee. I feel very proud for having people of this stature related to my work.

I take this opportunity to thank Ms. Sharon Grant, the secretary of the Database Systems Research and Development Center, for all the time and effort she spent in making the Center a pleasant place to work. I would like to thank Mr. John Bowers for providing me guidance and support during my graduate studies.

I thank Althea Liang, Raman Chikkamagaur, Seema Degwekar, Karthik Nagarajan who helped me learn many things and also made my stay in the Database Systems R&D Center enjoyable. I am also grateful to my roommates and friends from the C.I.S.E Department for their help during the implementation of the prototype system.

On a more personal note, I would like to thank my beloved parents, and my brothers and their families. Without their love, support and constant encouragement, none of this would have been possible.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

A QUERY COMPOSER AND A COMPOSITE SERVICE SPECIFICATION GENERATOR FOR DYNAMIC WEB SERVICE COMPOSITION

By

Lakshmi N. Chakarapani

May 2004

Chair: Stanley Y.W. Su
Major Department: Computer and Information Science and Engineering

Web service technology provides a uniform framework for distributed application development and software resource sharing over the Internet. Web services are network accessible applications and business processes published by business organizations that can be used by other organizations to develop distributed applications. With the popularity of Web services, building composite Web services from existing services has become an interesting and challenging research problem. Following this research direction, the trend is to automatically or semi-automatically discover and invoke composite Web services in a very flexible and efficient way based on the services registered with the Web service registry.

The goal of this thesis is to design and implement a user interface mechanism to facilitate composite Web service discovery and to implement an API to generate a Composite Service Specification document from a composite service plan. The very first step in composite Web service discovery is to interact with the service requestor

and understand his/her needs. The query composer module is a Web enabled graphical interface that guides the service requestor to express his/her requirements.

In our work, we have efficiently made use of Wordnet, UNSPSC categorization and IBM's UDDI Test registry to perform composite web service discovery. Once the composite service plan is formed the plan is expressed in the form of a business process model. Specifications like the Web Service Flow Language (WSFL) from IBM have been proposed to describe composition of web services in the form of work flow using XML. Microsoft has made a similar proposal called XLANG for web services composition. We make use of IBM's WSFL specifications. A WSFL specification describes how different web services should be invoked in terms of ordering and parallelism. The WSFL document is generated using Java API for XML Processing.

CHAPTER 1
INTRODUCTION

In today's world, organizations increasingly need to collaborate with each other to share information and application system resources. In the past, the IT industry focused on building systems efficiently. The advent of network-based computing has enabled organizations to build systems that can share data and software resources with all sorts of communities. This allows organizations to do business with customers, business partners, and government agencies with ease.

The Web service framework provides a promising architecture model for network- based interoperability which allows the sharing of application functionalities and business processes to conduct e-business over the Internet. The framework applies the concept of just-in-time integration and encapsulates the underlying service implementation details. With Web services, applications are modularized and wrapped in a set of standards-based interfaces that allows full interoperability with other services. With platform independent UDDI-based repositories, a simple and extensible communication protocol like SOAP [1] and a Web Service Description Language (WSDL) [2] to describe services, applications will be able to discover and interoperate with other remote applications from anywhere. This service-based model assures real-time application interoperability and integration.

1.1 Web Services – An Overview

A Web service is an interface that describes a collection of operations that are network accessible through a standard XML messaging [3]. As the interface hides the

implementation details, it allows the services to be used irrespective of hardware and software platform on which they are implemented. Being reusable, loosely coupled and component-oriented, they represent the next step of evolution from the object-oriented paradigm.

Web services consist of a set of related technologies. They together form the Web services conceptual stack as shown in Figure 1.1.

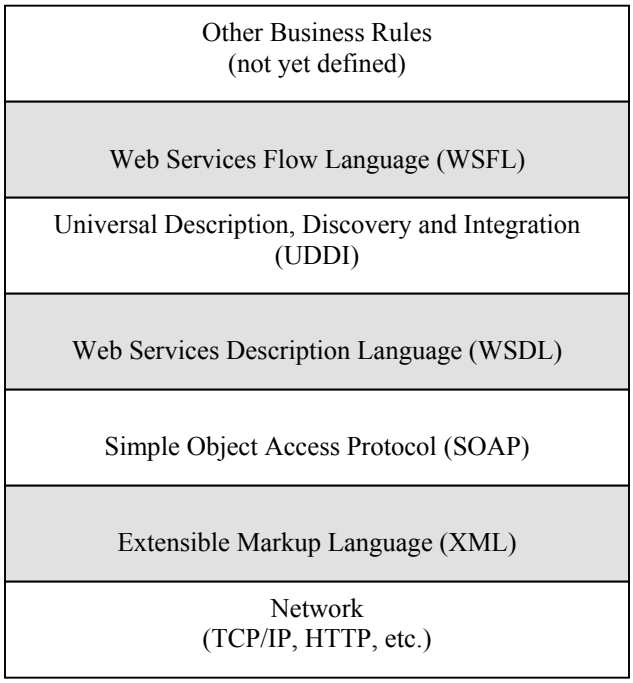

Figure 1.1 Conceptual Web Services Stack

The network layer forms the foundation of the Web services stack. HTTP is the de facto standard network protocol for Internet Web services. The base technology XML is the building block for Web services. The SOAP protocol [1] is used for XML messaging and for making remote procedure calls between applications. SOAP is basically an HTTP post with an XML envelope as the payload. It defines a standard mechanism of sending messages using SOAP headers and encoded operations. The Web Services Description Language (WSDL) [2] is an XML language for describing Web

services as a set of network endpoints that can operate on messages. It allows service description in a standard format. The UDDI (Universal Description, Discovery and Integration [4]) layer facilitates publishing and discovery of Web services. The Web Service Flow Language (WSFL) describes a composition of Web services in the form of workflow in an XML format. A WSFL specification describes how different services should be invoked in terms of ordering and parallelism.

The Web services architecture basically involves the collaboration of three main components: the Service Provider, the UDDI Service Registry, and the Service Requestor.

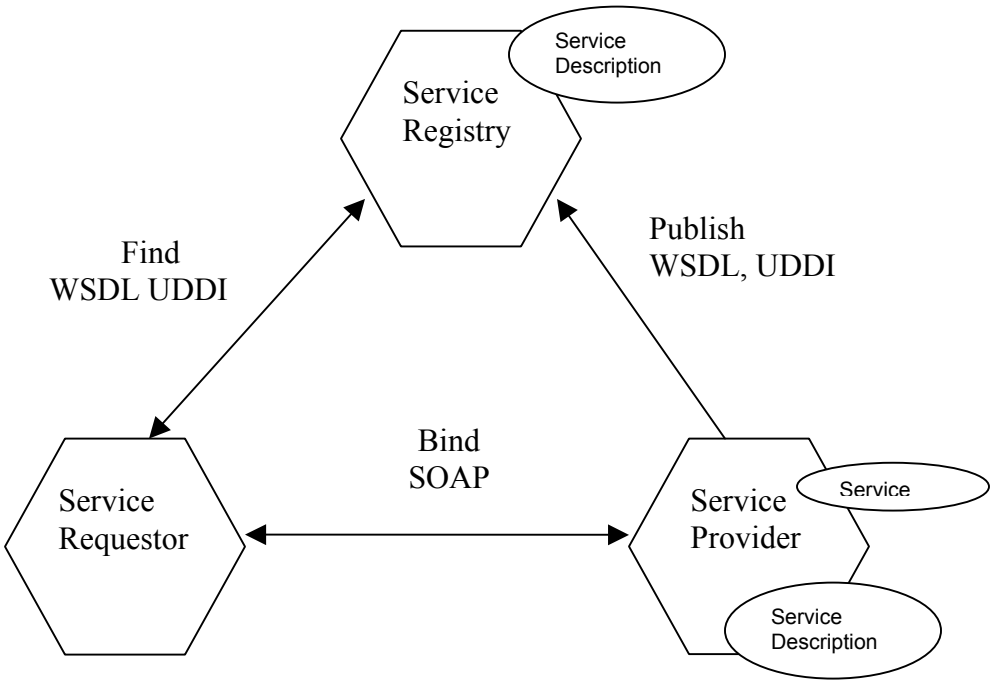

Figure 1.2 The Web Services Architecture

It also describes three operations that define the interaction between the components: publish, find and bind. Service providers publish their services to the service registry. Service requestors find the required services from service registry and bind them to appropriate service providers.

1.2 Challenges and Proposed Solutions

Current Web services allow applications to operate independently. In order to offer value added services, existing Web services should be composed in an efficient way to satisfy users' needs. Service composition can considerably reduce the development time and cost instead of building new applications. However, several extensions are needed to automate Web-based integration and application development. For example, with the current Web service model and its implementation, composite Web services can not be automatically discovered or dynamically constructed out of the existing registered services to meet users' service needs. When a service requestor browses the service registry, s/he may fail to find a service that meets her/his needs. However, his/her service request may be met by a number of services registered with the service registry. The current UDDI [5] registry (e.g., IBM's UDDI v.2.0) is not able to automatically discover composite Web services. Nor does it offer a mechanism, such as a user-system interaction to allow an easy description of a service query for finding a composite service. At the Database Systems R & D Center of the University of Florida, we have developed an Intelligent Registry capable of providing dynamic and semi-automatic Web service composition based on the request and constraint information provided by service requestors and service providers.

The Intelligent Registry extends the IBM's UDDI registry with constraint specification and matching capabilities, and provides support for constructing, describing and invoking composite Web services. The Intelligent Registry has a query composer module, which guides the service requestor to build a composite Web service and express his/her constraints associated with its component services. During the discovery process,

it interacts with the service requestor to assist him/her to identify useful services and prompts him/her to provide the needed data to complete the construction of a composite Web service. A Constraint-based Broker [6], a component of the Intelligent Registry, matches the constraints of the service request with those of registered services so that the constructed composite service will satisfy the user's constraints and suitable service providers of its component services can be found. The constructed composite service has an execution structure that dictates how a set of Web services should be invoked in a specific order. It also contains relevant information to map input-output data between these services. Based on the composite service, a corresponding WSFL document [7] is generated by a WSFL Generator. After the WSFL document is generated by the Intelligent Registry, it will invoke a Composite Service Processor [8] to execute it.

1.3 Goal of This Thesis and the Intended Contribution

Our work on Web service composition is the collaboration of several researchers at the Database Systems R&D Center. The architecture of the Intelligent Registry and the design and implementation of several of its major components are the work of Qianhui (Althea) Liang and the design and implementation of the Composite Service Processor (a WSFL execution engine) is the work of Raman N. Chikkamagalur. The focus of this thesis is to design and implement a user interface to facilitate composite Web service discovery and to implement an API to generate a Web Service Flow Language (WSFL) Document from a composite service plan. The very first step in composite Web service discovery is to interact with the service requestor and understand his/her needs. The query composer module is a Web enabled graphical interface that guides the service requestor to express his/her requirements.

In this work, we make use of Wordnet [9], UNSPSC [10] categorization and IBM's UDDI Test registry to perform composite Web service discovery. Wordnet is an online database of lexical relationships that generates synonyms for given words. Service providers tend to use different terms to express their services. We make use of Wordnet to solve this problem of semantic heterogeneity. The United Nations Standard Products and Services Code (UNSPSC) is a classification convention and it contains a name value pair that is used to numerically identify products and services. We use UNSPSC codes to get the domain of services that match the service request. IBM's UDDI registry is queried with a list of UNSPSC codes to get the list of services that match the service request. Finally, the query composer module captures the constraints of input and output data entities and the order of execution of services if the user has any preferences.

Once the composite service plan is formed the plan is expressed in the form of a business process model. Specifications like the Web Service Flow Language (WSFL) from IBM have been proposed to describe composition of Web services in the form of workflow using XML. Microsoft has made a similar proposal called XLANG for Web services composition. We make use of IBM's WSFL specifications. A WSFL specification describes how different Web services should be invoked in terms of ordering and parallelism. The WSFL document is generated using Java API for XML Processing [11] (JAXP). JAXP supports processing of XML documents using DOM, SAX, and XSLT.

1.4 Organization of Thesis

The organization of this thesis is as follows. Chapter 2 presents a survey of the enabling technologies. Chapter 3 describes the overall system architecture. Chapter 4 describes the framework used to discover composite Web services. Chapter 5 explains

the Composite Service Specification (CSS) Generator. Chapter 6 gives the implementation details. Chapter 7 gives a conclusion and suggests some future work.

CHAPTER 2
RELATED RESEARCH AND TECHNOLOGIES

This chapter presents a survey of the standards, tools and software that have been used for the development of the Query Composer and Composite Service Specification Generator.

2.1 UDDI

UDDI stands for Universal Description, Discovery and Integration [4]. The UDDI specification enables businesses to quickly, easily, and dynamically find and transact with one another. In today's e-commerce environment, there is no standard mechanism for a company to describe their services and specify how they prefer to conduct e-business. UDDI makes it possible for organizations to quickly discover the right business and to describe their services.

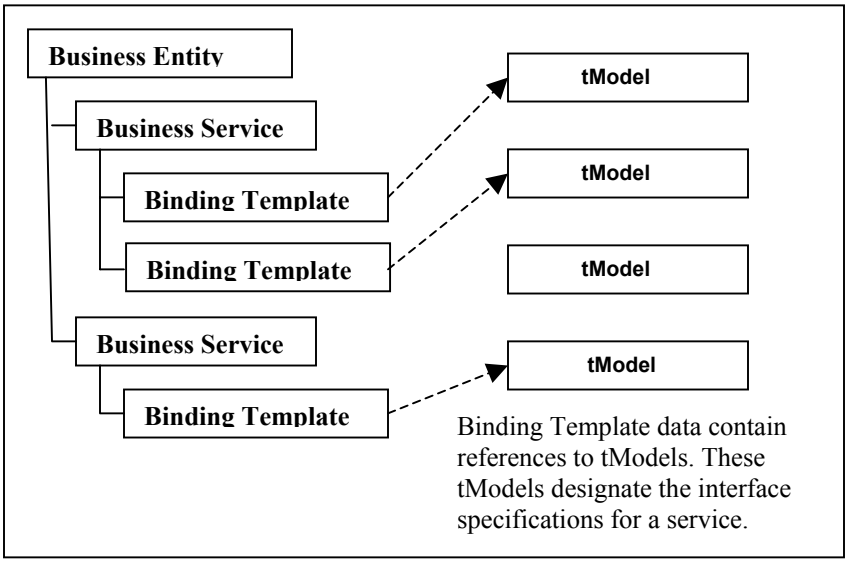

Figure 2.1 UDDI Data Structure

The UDDI Business Registry contains information about businesses and the services they offer. The information is organized as follows:

Business entity. A business entity provides information about a business. This structure is the topmost level of the data structure hierarchy. Each business entity will contain a unique identifier, the business name with which the service provider wants to register, a short description of the business services they provide, some contact information, a list of categories and identifiers that describe the business, and a URL pointing to more information about the business specified in a WSDL document.

Business service. Every business entity is associated with a list of business services offered by it. Each business service entry contains a business description of the service, a list of categories that describe the service, and a list of pointers to references. The technical information such as binding information, service access points, a reference to the service interface, etc is also stored via containment of another structure called the Binding Template structure.

Binding templates. The business service entry in turn will be associated with a list of binding templates that point to specifications and other technical information about the service. For example, a binding template might point to a URL that supplies information on how to invoke the service. The pointer also associates the service with a service type.

Service types. A service type is defined by a tModel. Multiple businesses can offer the same type of service, as defined by the tModel. A tModel specifies information such as the tModel name, the name of the organization that published the tModel, a list of categories that describe the service type, and pointers to technical specifications for the

service type such as interface definitions, message formats, message protocols, and security protocols.

In this work, we have made use of a publicly available UDDI-enabled registry – the IBM UDDI version 2.0 Business Test Registry. We will look at the features and design principles of it in Chapter 4.

2.2 WSDL

The Web Services Description Language (WSDL) [2] is an XML language for describing Web services as a set of network endpoints that can operate on messages. A WSDL service description basically contains an abstract definition for a set of operations and messages, a concrete protocol binding for these operations and messages, and a network endpoint specification for the binding.

The abstract information is kept separate from a concrete network deployment. The abstract information is called as service interface information and the network information is called as service implementation information. As WSDL separates these two kinds of information, it allows reuse of the abstract definitions. Let us examine the elements of a WSDL document [2].

Definitions. The <definitions> element contains the definition of one or more services. The <definitions> tag also has the following attribute declarations: name (name of service), targetNamespace (defines a logical namespace for the information in service) and xmlns (The default namespace of a WSDL document is set to http://schemas.xmlsoap.org/wsdl/).

Within definitions, there are three conceptual sections:

- <message> and <portType>: What operations the service provides.
- <binding>: How the operations are invoked.

- <service>: Where the service is located.

Message. A `<message>` corresponds to a single piece of information moving between the invoker and the service. A regular round trip remote method call is modeled as two messages, one for the request and one for the response. Each `<message>` can have zero or more parts, and each part can have a name and an optional type.

PortType. A `<portType>` corresponds to a set of one or more operations, where an `<operation>` defines a specific input/output message sequence. The message attribute of each input/output must correspond to the name of a `<message>` that was defined earlier. WSDL has four operation primitives that an endpoint can support – one-way, request-response, solicit-response, and notification.

Binding. A `<binding>` corresponds to a `<portType>` implemented using a particular protocol such as SOAP or CORBA. The type attribute of the binding must correspond to the name of a `<portType>` that was defined earlier. Because WSDL is protocol neutral, you can specify bindings for SOAP, CORBA, DCOM and other standard protocols.

Service. A `<service>` is modeled as a collection of ports, where a `<port>` represents the availability of a particular binding at a specific end-point. The binding attribute of a port must correspond to the name of a `<binding>` that was defined earlier.

As WSDL is an XML format, it is open to language extensions. Also, it provides a uniform manner of service description. For maximum interoperability, any service provider will use WSDL to describe his/her service. Also, UDDI provides a data structure support to both service interface and service implementation information. In our earlier work [6] we have extended WSDL specifications to include constraint information.

2.3 SOAP

SOAP (Simple Object Access Protocol) [1] allows a program to invoke service interfaces across the Internet, without the need to share a common programming language or distributed object infrastructure. For example, any company in any corner of the network may be using a programming language and a computing platform that are different from the programming language and the platform used by some other company in some other part of the network. The SOAP framework allows two companies or Business centers to interact with each other even though they may be using different infrastructures. XML provides a unique cross platform approach for encoding and formatting data. SOAP on the other hand, which is basically built on top of XML, provides a simple approach to package information for transfer across the network. It provides a way for systems to make remote procedure calls to some other system to make use of services that are published at the Business center. Hence XML and SOAP, provides a cross-language, cross-platform approach that enables two companies to be compatible with each other.

A SOAP message consists of three parts:

- The SOAP envelope construct defines an overall framework for expressing what is in a message, who should deal with it and whether it is optional or mandatory.
- The SOAP encoding rules defines a serialization mechanism that can be used to exchange instances of application-defined data types.
- The SOAP RPC representation defines a convention that can be used to represent remote procedure calls and responses.

When a service requestor finds a service that he needs using the information provided in the WSDL service implementation document of the service provider, he can

then invokes the SOAP server of the service provider to establish a connection. Once this is done he can then use the services.

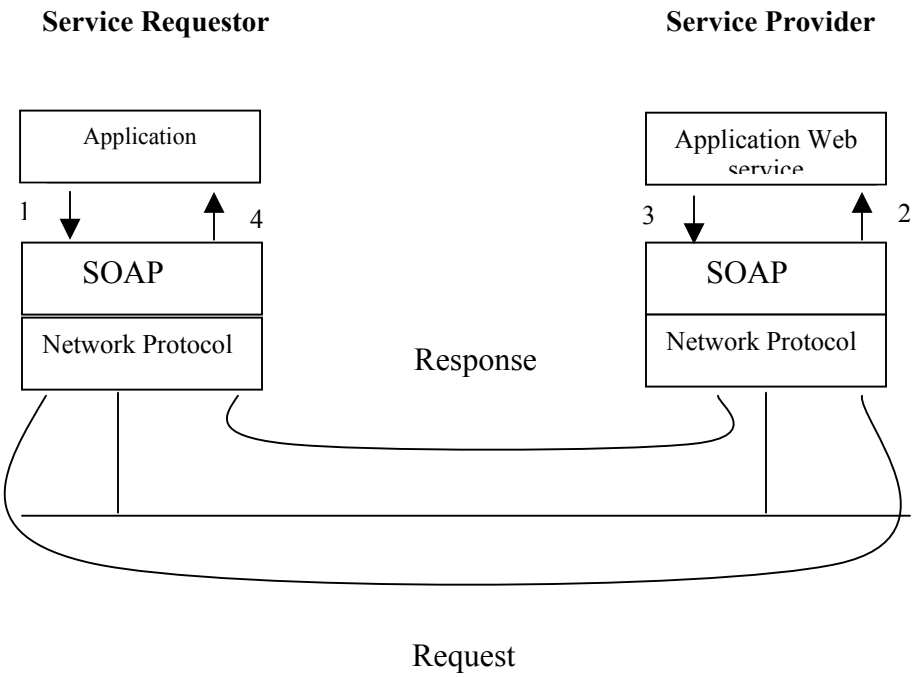

Figure 2.2 XML messaging using SOAP

Figure 2.2 shows XML Messaging using SOAP. It is a four-step process:

1. Service requestor's client application generates a SOAP message and makes the service invocation request. The service requestor binds this message together with the network address of the service provider to the SOAP infrastructure (for example, a SOAP client at runtime). The SOAP client at runtime interacts with an underlying network protocol (for example HTTP) to send the SOAP message over the network to the service provider.

2. The network infrastructure delivers the message to the service provider's SOAP runtime (SOAP server). The SOAP server routes the request message to the service provider's Web service. The SOAP server is responsible for converting the XML message into programming language-specific objects if required by the application.

3. The Web service is responsible for processing the request message and formulating a response. The response is also a SOAP message. The SOAP server sends the SOAP message response to the service requestor over the network.

4. The response message is received by the networking infrastructure on the service requestor's node. The message is routed through the SOAP infrastructure; potentially converting the XML message into objects in a target programming language. The response message is then presented to the application.

2.4 POM

The Persistent Object Manager (POM) [12] consists of object to relational mapping engine. It provides a persistent storage facility for storing information in the system. Its high level API can be used to store, retrieve, update and delete objects without having to know the internal structures of the objects. POM is implemented on top of an Object- Relational database system called Cloudscape. In our work we have used POM to store UNSPSC [10] codes. POM is explained in detail in Chapter 4.

2.5 UNSPSC

Coding products and services according to a standardized classification convention is necessary for streamlining commerce among companies. United Nations Standard Products and Services Code [10] is a classification convention that is used to numerically identify all products and services. It is the most efficient, accurate and flexible classification system available today and allows full exploitation of electronic commerce capabilities. The UNSPSC is a hierarchical classification, having five levels. The levels allow users to search products more precisely as searches will be confined to logical categories and eliminate irrelevant hits.

Each level contains a two-character numerical value and a textual description as follows:

Segment. The logical aggregation of families for analytical purposes
Family. A commonly recognized group of inter-related commodity categories
Class. A group of commodities sharing a common use or function

Commodity. A group of substitutable products or services

Business function. The function performed by an organization in support of the commodity.

For example, Theme parks have a UNSPSC code 90.15.17.01.00, which is under Amusement parks whose UNSPSC code is 90.15.17.00.00. The hierarchical classification of products allows easy searching.

2.6 JWORDNET

WordNet [9] was developed by the Cognitive Science laboratory at the Princeton University. It is an online lexical reference system whose design is inspired by current psycholinguistic theories of human lexical memory. English nouns, verbs, adjectives and adverbs are organized into synonym sets, each representing one underlying lexical concept. Different relations link the synonym sets. JWordnet is an object-oriented wrapper on top of the Wordnet database. It provides an application program interface to query Wordnet.

2.7 Java Related Technologies

2.7.1 Java Sever Pages

Java Server Pages (JSP) [13] enables to generate Web content dynamically. JSP allows Web designer to rapidly develop and easily maintain information-rich, dynamic Web pages that leverage existing business systems. As part of the Java family, JSP technology enables rapid development of Web-based applications that are platform independent. It separates the user interface from content generation enabling designers to change the overall page layout without altering the underlying dynamic content.

JSP technology uses XML-like tags that encapsulate the logic that generates the content for the page. Additionally, the application logic can reside in server-based resources that the page accesses with these tags. All formatting (HTML or

XML) tags are passed directly back to the response page. By separating the page logic from its design and display and supporting a reusable component-based design, JSP technology makes it faster and easier than ever to build Web-based applications.

2.7.2 Tomcat Engine

Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet and Java Server Pages technologies. The Java Servlet and Java Server Pages specifications are developed by Sun under the Java Community Process. Tomcat used Java Development Kit (JDK 1.2 or above) to compile JSP and Servlets. Tomcat can be easily installed and maintained in windows and unix based machines.

In our work, the Query composer is implemented by using JSP technology and Tomcat as Web server.

2.8 WSFL

The goal of Web services is to enable seamless application integration over the Internet regardless of programming languages or operating environments. Web services composition (or Web services workflow) is to enable the same type of seamless integration across business processes and transaction lifecycles that make use of many Web services. It is a process of constructing complex Web services from primitive ones. This enables the rapid and flexible creation of new services that lead to a significant reduction in development cost and time.

IBM recently announced the release of a new XML grammar for defining software workflow processes for Web services composition within the framework of the Web services architecture, called Web Services Flow Language (WSFL) [7]. WSFL is a tool to model a workflow using an XML syntax that can be read by both humans and

machines [8]. By taking WSFL document as input, a workflow engine can interpret the business process and execute it. Given the revolutionary power of Web services to bridge cross-platform boundaries, the power of WSFL lies in its ability to model business processes that span technology boundaries as well as across business boundaries.

WSFL is an XML language for the description of Web services compositions. WSFL considers two types of Web services compositions. The first type specifies the appropriate usage pattern of a collection of Web services in such a way that the resulting composition describes how to achieve a particular business goal. Typically, the result is a description of a business process. In WSFL, this is called a flow model. The second type specifies the interaction pattern of a collection of Web services. In this case, the result is a description of the overall partner interactions. In WSFL, this is called a global model.

2.8.1 Flow Model

In a flow model, a composition is created by describing how to use the functionality provided by the collection of composed Web services. WSFL models these compositions as specifications of the execution sequence of the functionality provided by the composed Web services. Execution orders are specified by defining the flow of control and data between Web services. Flow models can especially be used to model business processes or workflows based on Web services. This section describes the main concepts of the Meta model underlying WSFL for specifying flows.

Operations of Web services are used within business processes as implementations of activities. An activity represents a business task to be performed as a single step within the context of a business process contributing to the overall business goal to be achieved. The operation used may be perceived as the concrete implementation of the abstract activity to be performed. Activities correspond to nodes in a graph. Each

activity has a signature that is related to the signature of the operation that is used as the implementation of the activity. Thus, an activity can have an input message, an output message, and multiple fault messages. Each message can have multiple parts, and each part is further defined in XML XSD type system similar to WSDL messages.

Activities are wired together through control links. A control link is a directed edge that prescribes the order in which activities will have to be performed (that is, the potential "control flow" between the activities of the business process). The endpoints of the set of all control links that leave a given activity represent the possible follow-on activities. Which of the following activities actually have to be performed in the concrete instance of the business process (that is, the concrete business context or business situation) is determined by so-called transition conditions. A transition condition is a Boolean expression that is associated with a control link. The formal parameters of this expression can refer to messages that have been produced by some of the activities that preceded the source of the control link in the flow. Typically, parallel work has to be synchronized at a later time. Synchronization is done through join activities. An activity is called a join activity if it has more than one incoming control link. By default, the decision whether a join activity is to be performed or not is deferred until all parallel work that can finally reach the join activity has actually reached it. Activities that have no incoming control are called start activities. Activities that do not have outgoing control are called end activities.

There is a second kind of directed edge in the graphs of the metamodel, the so-called data links. A data link specifies that its source activity passes data to the flow engine, which in turn has to pass (some of) this data to the target activity of the data link.

A data link can be specified only if the target of the data link is reachable from the source of the data link through a path of (directed) control links. It is not required that data be always passed to an immediate successor of its producer. Many different activities might be visited along the path made from control links from the source of a data link to the target of the data link. An activity might be the target of multiple data links. For example, this allows aggregating input from multiple sources, or it allows specifying alternative input from activities from alternative parallel paths. To facilitate this, data links are weighted by so-called map specifications. A map prescribes how a field in a message part of the target's input message of a data link is constructed from a field in the output message's message part of the source of the data link. It even allows multiple maps to be defined for the same message part target. This is needed, for example, when alternative paths in the control are specified and data needed further on can be produced along each of the paths.

2.8.2 Global Model

In the global model, no specification of an execution sequence is provided. Instead, the composition provides a description of how the composed Web services bind to actual service provider. The interactions are modeled as links between endpoints of the Web services' interfaces, each link corresponding to the interaction of one Web service with an operation of another Web service's interface. Every activity defined in a WSFL flow model can be implemented in the form of a Web service defined by WSDL. The separation of the flow model and the global model allows us to keep the abstract definition of the workflow process (the flow model) separate from the specific details about how a given process has been implemented (the global model).

A global model defines the interaction between a set of service providers. Interactions are modeled using plug links between “dual” operations on the service provider types involved in the composition.

CHAPTER 3
OVERALL SYSTEM ARCHITECTURE

In this chapter, we provide a detailed explanation of the overall system architecture of the "Intelligent Registry" and the Web service registration, discovery, description and invocation process in a composite Web services model. The purpose of the system is to provide dynamic and semi-automatic Web service composition, based on end users' requests taking into consideration the constraint information provided by service requestors and service providers. This chapter is organized as follows. Section 3.1 gives the overall system architecture. Section 3.2 explains the working of the Intelligent Registry in the composite Web services model.

3.1 Overall System Architecture

Figure 3.1 shows the architecture and the components of our system. The four roles in the composite Web services model are

- Service Provider
- Service Requestor
- Intelligent Registry
- Composite Service Processor

The Service Provider is the owner of a Web service who also hosts access to the service. The Service requestor is a user/application looking for and invoking or initiating an interaction with a Web service registered with the Intelligent Registry. The service requestor's role can be played by a browser driven by a person or by a program (e.g., an application system or another Web service). The Intelligent Registry has the following functionalities: (1) interacts with the service requestor to compose service query, (2)

discovers simple and composite Web services, and (3) generates composite service specifications for discovered composite services. The composite service processor invokes the individual Web services.

The Intelligent Registry consists of the following five components:

- Constraint-based Broker
- Query Composer
- Service Composer
- Service Dependency Graph (SDG) Generator
- Composite Service Specification (CSS) Generator

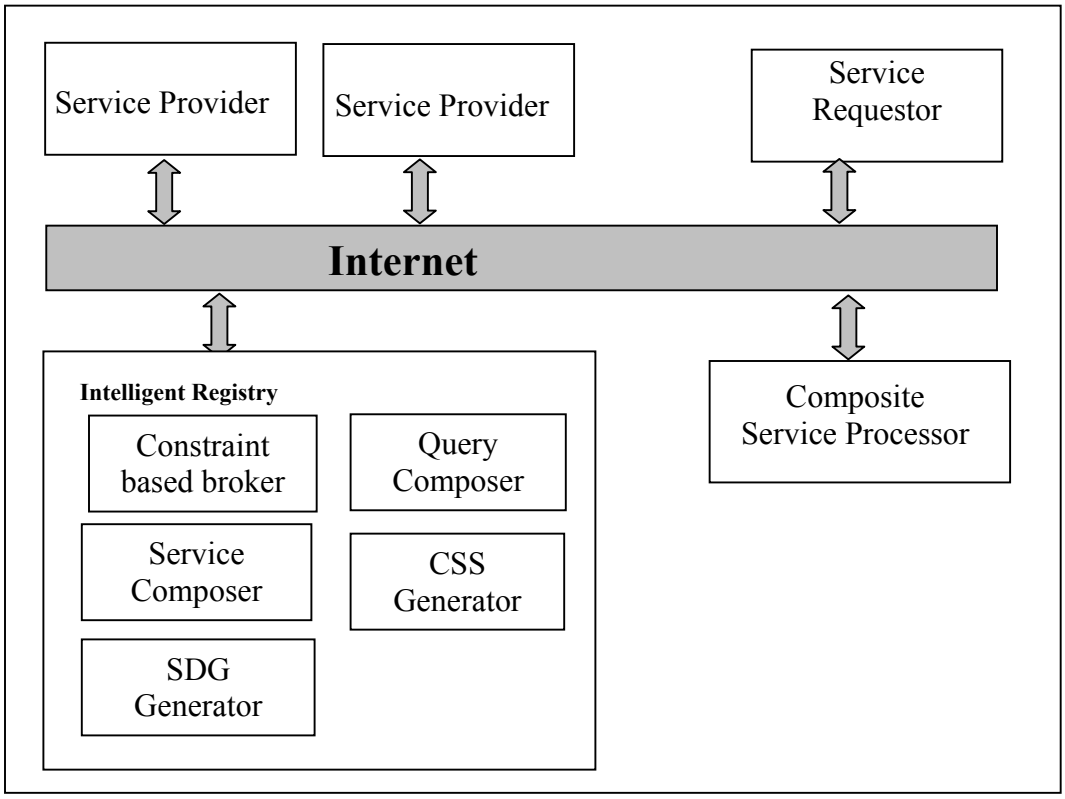

Figure 3.1 System Architecture

Constraint-based Broker. The Constraint-based Broker [6] is implemented on top of IBM's UDDI registry. It supports all basic operations like publish, find and save, as per the UDDI specification. Additionally, it allows a service provider to specify constraints

associated with the registered services and service operations, and constraints associated with the input and output data of the service operations. It is also capable of matching the constraints specified by the user with those of service providers to select the suitable providers. The WSDL specification is extended [6] to include constraint specifications in a service description.

Query Composer. The Query Composer has a user interface module that guides the user to describe his/her service request and constraints. The user interface module makes use of various components like Wordnet, UNSPSC, IBM's UDDI registry to assist the service requestor to express his/her service request. The Service request is expressed in the form of a XML document. The components of the Query Composer are explained in detail in Chapter 4.

SDG Generator. A Service Dependency Graph (SDG) is dynamically constructed by the SDG Generator based on those registered simple and composite services that are relevant to the service request. By searching SDG, Service Composer constructs a "composite service template," which is a structure of data entities and service operations that forms a composite service.

Service Composer. The service request is issued as a query to the Service Composer, which processes the request against a Service Dependency Graph (SDG). The Service Composer finds a sub-graph of SDG that satisfies the users' request and comes up with a composite service template, which specifies an execution plan. The template is a directed acyclic graph that represents a set of Web services connected in a planned executable order. It dictates how a set of Web services should be invoked in a specific order. It also

contains mapping information between input and output data entities between the services.

CSS Generator. The constructed template is given to the Composite Service Specification (CSS) Generator to generate a composite service description document in WSFL. The generated WSFL document will capture the sequence of execution of the Web services and the dataflow between the Web services in an XML format. The WSFL document also contains the endpoint information of the individual Web services.

Composite Service Processor. The WSFL document is then used by the Composite Service Processor [8] to invoke the component services of the discovered composite service. We have deployed the Composite Service Processor as a Web service. In this way, the Composite Service Processor can easily be plugged into the current Web services architecture as a value-added component.

3.2 Web service Discovery, Description and Invocation in the Composite Web Services Model

Figure 2 illustrates the Web service registration, discovery, description and invocation process. The service provider registers his/her service with the Intelligent Registry by specifying its operations and constraints in an extended WSDL document (Label 1). The user makes a service request (Label 2). Based upon the request, the Intelligent Registry will first attempt to find registered simple/composite services to satisfy the user's request. If one cannot be found, it will attempt to construct a composite service that will satisfy the request. If a composite service can be constructed by the Service Composer, it will be displayed to the user for his/her verification and approval (Label 3). Upon approval by the user, a Composite Service Specification is generated as a WSFL document and stored in a persistent storage of the Intelligent Registry. The URL

for accessing the service of the Composite Service Processor is given to the user. The user can then issue an invocation request programmatically to the Composite Service Processor (Label 4).

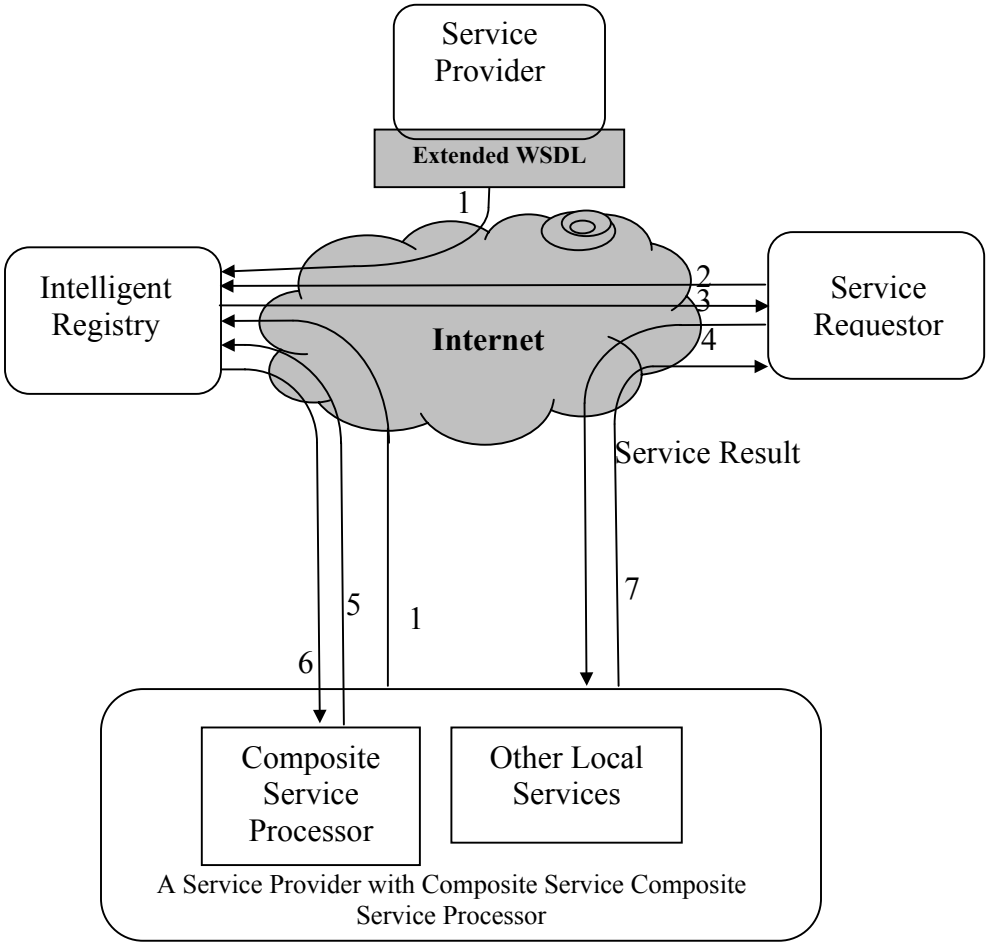

Figure 3.2 Web Service Registration, Discovery and Invocation process

The Composite Service Processor accesses the WSFL document in the Intelligent Registry (Label 5). It receives (Label 6) and follows the specification of the document to invoke the component Web services. The result is then sent back to the service requestor (Label 7).

CHAPTER 4
QUERY COMPOSER

In this chapter, we first give the overall design of our Query Composer and explain its functionalities followed by the design and functionalities of its individual components. This chapter is organized as follows. Section 4.1 explains the query composer architecture. Section 4.2 gives the information flow in the Query Composer. Section 4.3 gives details about UDDI and POM. Section 4.4 explains the usage of categorization in UDDI.

4.1 Query Composer Architecture

The architecture of the query composer is shown in the Figure.

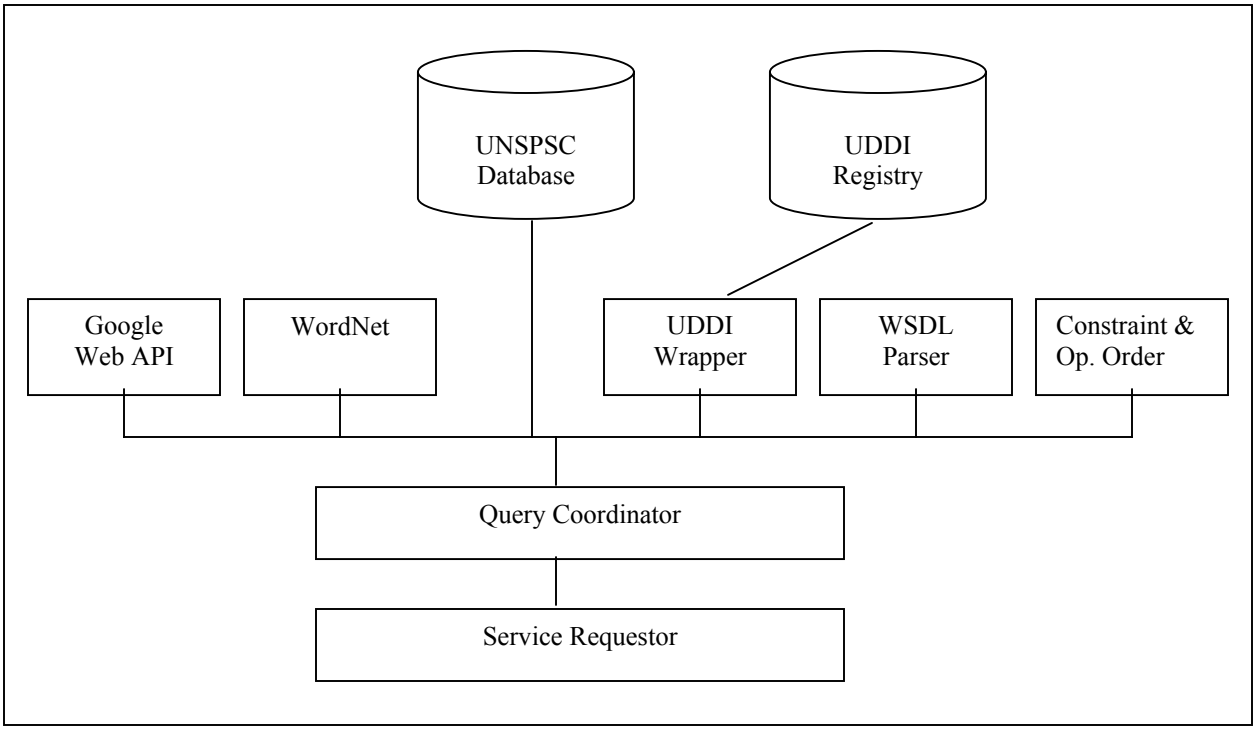

Figure 4.1 Query Composer

The key components of the system are: Google Web API [14], Wordnet, UNSPSC database, UDDI registry, WSDL parser and Constraints and Operation Order Module. The Query Coordinator forms a communication link between these components. The UDDI Wrapper provides the necessary functionality to search the UDDI registry. Google provides spell checking API that can be used within applications to check for spelling errors. We have made use of Google Web API to correct the spelling mistakes of the service requestors, if there are any.

The Query Coordinator handles all requests from the service requestor. The system works as follows: The service requestor uses a term to describe his/her request. The Query Coordinator uses Google's Web API to correct the spelling mistakes if there are any. It queries the Wordnet, an online lexical relationship reference system, to form a domain of words that closely match with the service requestor's request. These terms are matched against the UNSPSC database to obtain a list of categories that correspond to the interest of service requestor. The UDDI Wrapper module obtains the list of services under each category and lets service requestor to select the services s/he is interested in. The WSDL Parser module parses the WSDL document to obtain the list of operations. Once the user selects the list of operations, the Constraint and Operation Order Module obtains the constraint information for input and output attributes, also, it lets the requestor to select the execution sequence of operations if s/he has any preference.

4.2 Information flow in the Query Composer

In the Intelligent Registry, we separate the query composing process into two phases. During the first phase, the lists of services that correspond to the service requestors are obtained. In the second phase, the service requestor selects among the

services that s/he is interested in and gives constraint information and partial execution order of service operations if s/he has any preference.

Figure 4.2 shows the phase 1 of Query Composer. The very first step in composite service discovery is to interact with the service requestors and understand their service needs. The search process begins with the service requestor using his/her own terms to indicate the needed services. For example, if a requestor intends to make a trip, s/he may specify Airline, Car rental and Hotel reservation as the (component) services that s/he needs. Alternatively, s/he may simply type in a general term such as "Travel" to express her/his desire to acquire services for a trip. Our Query Composer is intelligent enough to find its component services.

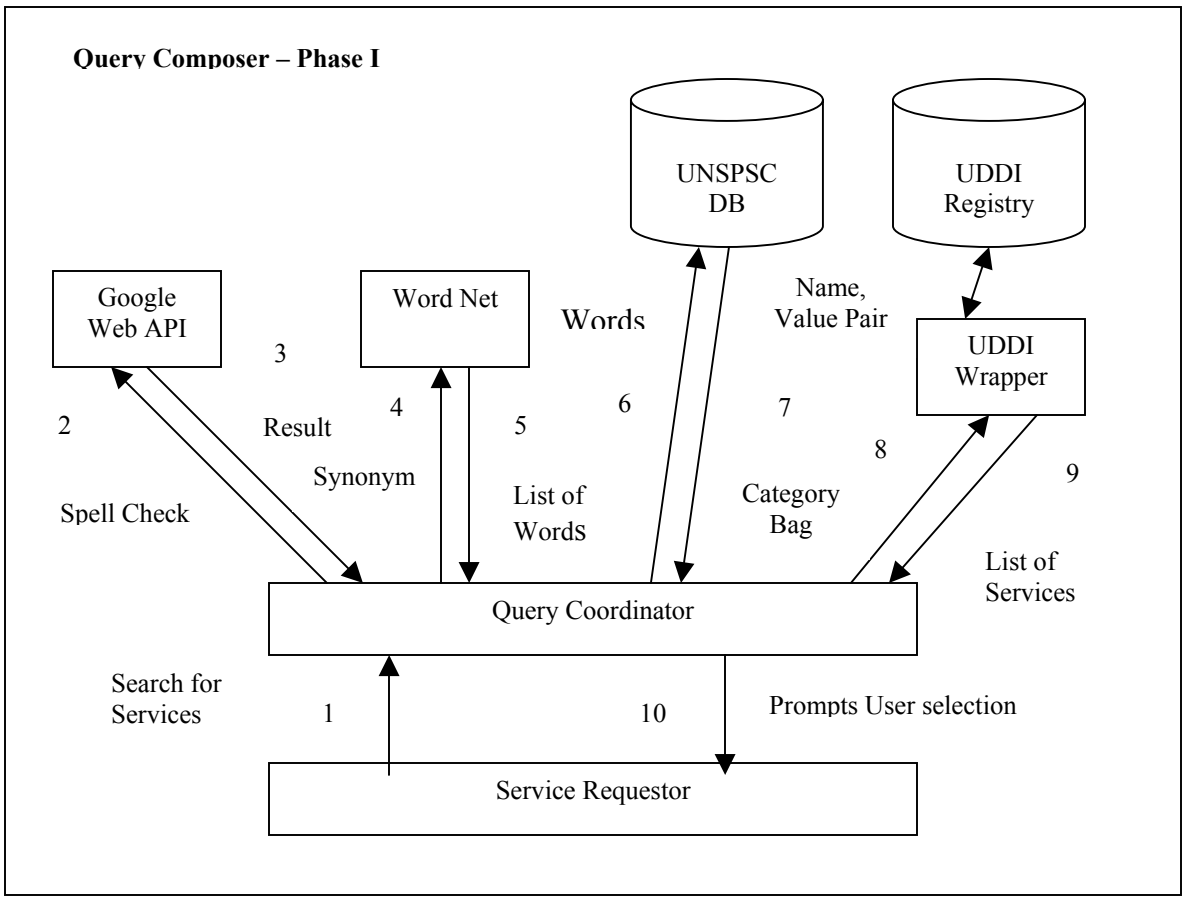

Figure 4.2 Query Composer Phase I

It is quite possible that the service requestor makes some spelling errors while trying to express his/her request. The query coordinator makes use of Google's Web service [Google] facility to correct the spelling mistakes if there are any. A composite service might be constructed based on different services of different service categories; the use of different terms to mean the same thing presents a very difficult problem. It is necessary to map the terms used by the service requestor to the dissimilar terms used in different service categories. Query composer solves the problem of semantic heterogeneity by using WordNet. It queries the WordNet with the terms given by the service requestor. Wordnet returns a list of words that closely matches with the service requestor's request.

UDDI provides a mechanism to include standard taxonomies that can be used to describe each entry using as many industry standard search terms as needed. Using categorization, the UDDI Inquiry API can quickly and efficiently connect businesses and services to customers who need them. In our work, we have made use of the UNSPSC categorization to find services that matches users' request. We have stored the UNSPSC categorization in our own database. This system that manages the database is called Persistent Object Manager (POM), which is the result of an earlier project of our research center. POM stores the UNSPSC categorization in the form of name value pair. The lists of words are mapped against the POM to find the matching categories. The matching categories are returned in the form of a category bag, which is a name value pair.. The Query Coordinator makes use of the UDDI Wrapper to obtain the list of services under each category. The service requestor is then prompted to select the services that s/he is interested in. This completes the Phase I in query composition.

The second phase in query composition is to let the service requestor to select the operations within a service and express his/her constraint information along with the execution order among the operations if s/he has any preference. Figure 4.3 shows phase 2 in query composer. Each service consists of various operations; however in a service the service requestor might be interested in a particular operation. The WSDL Parser Module parses the WSDL document and lets the requestor to select among the operations. The WSDL Parser gets the tModel key from the Query Composer, downloads the WSDL document, and parses it at runtime.

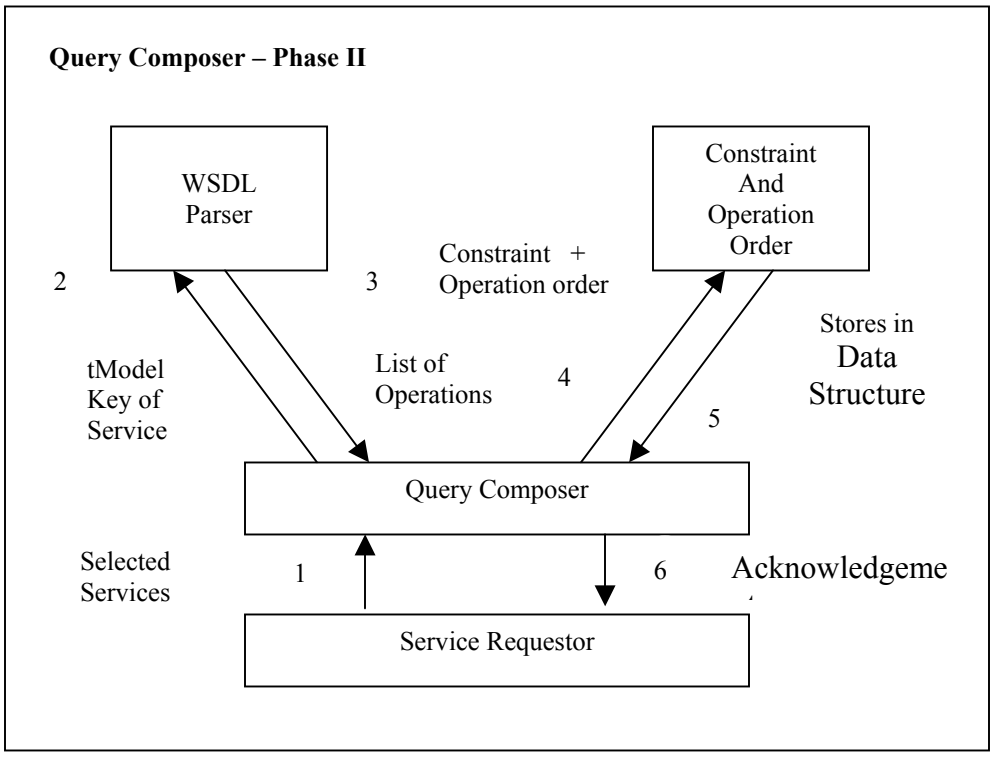

Figure 4.3 Query Composer Phase II

Once the requestor selects the list of operations, the Constraint and Operation Order Module lets the users express his/her constraint information for input and output entities of each operation. If the requestor has any preference in the execution order of the operations, it captures it. The constraint and operation order information is stored in a

data structure that needs to be further processed by the Service Composer to match against the Service Dependency Graph (SDG). The Service request is generated in the form of an XML document and given to the Service Composer. The details of the data structures are explained in Chapter 6.

4.3 UDDI and POM

In order to leverage the state of art technology, we have made use of UDDI enabled registries. In our work we have used IBM's UDDI Business Test Registry 2.0. IBM provides UDDI4J an API to use their Business Test registry. The UDDI API specification defines itself as a collection of XML requests and their responses. The API capabilities are briefly discussed below. The API is logically divided into two parts. They are the inquiry API and publishers API. Publishers API can be used by programmers to create tools that can directly interact with UDDI to publish their services. The inquiry API lets programmers to search and browse information stored in the UDDI registry. Some of the features of the API are discussed below:

Unique identifiers. When a service is registered with the UDDI registry, it is identified and referenced by a universally unique identifier, known as a UUID. UUID's are assigned when the data structure is first inserted into the UUID registry. They are hexadecimal strings whose structure and generation algorithm is defined by the ISO/IEC 11578:1996 standard. This standard virtually guarantees the generation of a unique identifier by concatenating the current time, hardware address, IP address, and random number in a specific fashion. The Inquiry API uses the UUID to request a particular structure on demand.

SOAP messaging. XML message passing in UDDI is achieved by using SOAP. SOAP uses XML and HTTP to deliver information using remote procedure mechanisms. SOAP

messages can be strongly protected through digital signatures and encryption. SOAP provides a convenient transport mechanism to link applications over the Internet. The usage of SOAP in UDDI specification ensures easy accessibility of applications.

Error handling. SOAP specification takes care of most of the errors effectively. UDDI specification defines error codes that are not handled by SOAP. SOAP has specific fault codes for fault reporting and handling invalid requests. In case of an application level error, a structure called a disposition report will be embedded inside a SOAP fault report. The UDDI 4J provides necessary functions to report the error to the user. Errors are detected prior to processing a request and it results in a minimum load on the registry at any time.

Security. UDDI provides different levels of security for searching and publishing Web services. Only authenticated users are allowed to use the service provider (publish) API. The UDDI4J API uses Java Secure Socket Extension (JSSE): a set of Java packages that enable secure Internet communications. It implements a Java version of SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication. These security features guarantee a secure transaction for service discovery and registration.

We now discuss the design principles of POM and advantages of using such a system. POM uses an Object-relational database called Cloudscape to store objects in a persistent fashion. It bridges the gap between object oriented paradigm and relational paradigm by providing an effective mapping between relational and object oriented mechanisms. It provides features of object storage, object retrieval, object updation and

object deletion. It also provides concurrency control and transaction management to enable data sharing among multiple users. POM provides the object to relational mapping capability by integrating the Java Programming Language, JDBC and the cloudscape DBMS as shown in the Figure 4.4.

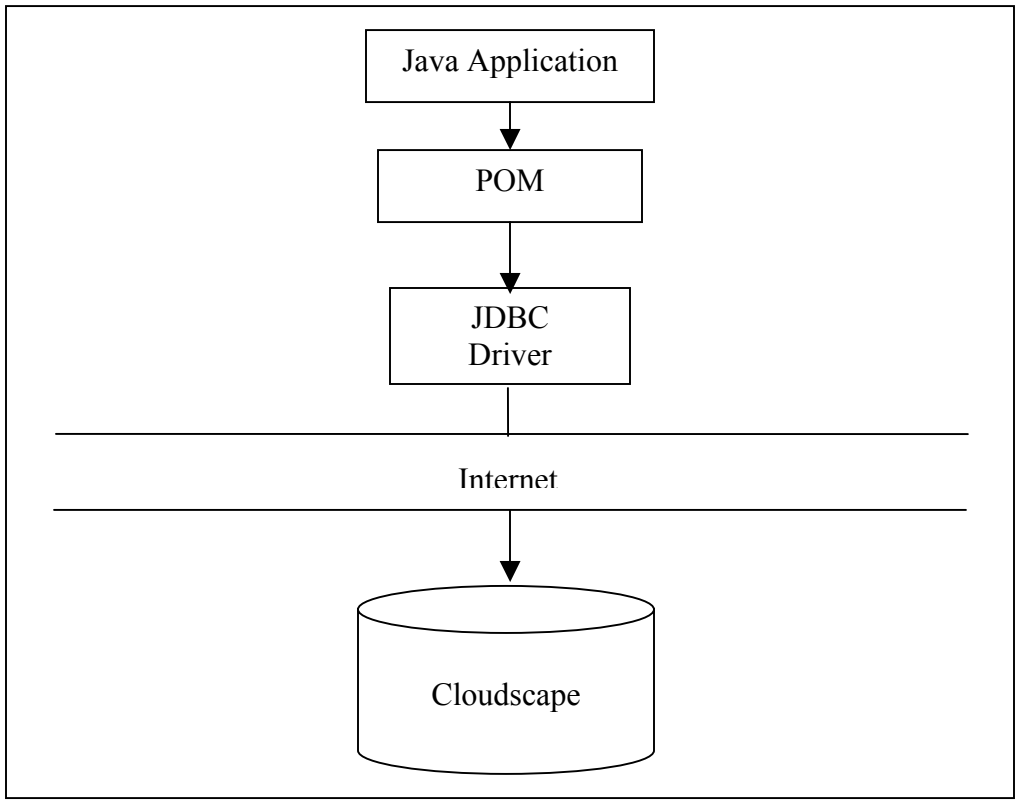

Figure 4.4 POM System Architecture

4.4 UNSPSC categorization and UDDI

The Universal Description Discovery and Integration (UDDI) specification provides a platform-independent way of describing services, discovering businesses, and integrating business services using the Internet. Taxonomies systems are crucial to UDDI. It is through categorization and identification that businesses are able to find each other and their appropriate services. The current UDDI registry is expected to grow in data size to millions of records in its database. Searching such a large data set with

"simple" search mechanisms (with few criteria) would result in a large result set. For example, searching for all the “car Rental” agencies would return a large number of car related rental and manufacturing companies. Taxonomies provide the most important mechanism in UDDI to help achieve more search-result occurrences for a service or business. When more classification is applied to a business or service, better search results will occur.

The UDDI data structure used to describe taxonomies is called a tModel. This is because a tModel is used both to define a service's technical interface and as a taxonomy, or namespace, which specifies the categorization or identification scheme. This is also because tModel structures are referenced, unlike other structures which hold containment relationships, amongst themselves. When describing how a Web service is to interact with its clients, the specification information is stored in the tModel structure. The tModel is used as a service type in this role.

For instance, in Figure 4.5, a tModel is registered for an airline reservation. Notice that the tModel refers to a WSDL document, in the overviewURL, to describe the details of the protocol, or service type. In UDDI, only the metadata is stored about businesses, services, and taxonomies, but never the actual data. In this case, the tModel points to the WSDL, rather than storing the WSDL itself.

Also note that the UDDI registry assigns a unique UUID (Universally Unique Identifier) to a tModel stored and is shown as tModelKey. The UUID of the credit check report tModel appears as a attribute to the tModel node (UUID:AAAAAAA....). In Figure 4.5, notice that a tModelKey is being referenced in the categoryBag structure of the credit-check tModel. This depicts the other usage of tModel, as an abstract namespace or

taxonomy. The keyedReference structure holds a tModelKey, which contains a keyName and a keyValue. This refers to a value in a given namespace defined by the tModel, which is represented by its key.

```
<tModel xmlns="urn:uddi-org:api"
tModelKey="UUID:AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAAAA">
<name>myairline:airlineReservation</name>
<description xml:lang="en">AirLine Reservation System</description>
 <overviewDoc>
<overviewURL>http://www.myairline.com/reserve.wsdl</overviewURL>
</overviewDoc>
<categoryBag>
<keyedReference
tModelKey="UUID:1F8F4B50-9D53-11D7-A596-000629DC0A53 "
keyName="Travel facilitation"
keyValue="90.12.00.00.00"/>
</categoryBag>
</tModel>
```

Figure 4.5 An Airline Reservation Service type specification.

In Figure 4.5, notice that a tModelKey is being referenced in the categoryBag structure of the credit-check tModel. This depicts the other usage of tModel, as an abstract namespace or taxonomy. The keyedReference structure holds a tModelKey, which contains a keyName and a keyValue. This refers to a value in a given namespace defined by the tModel, which is represented by its key. The various kinds of taxonomies could be used for identification or classification of all the data types in UDDI, as mentioned in the UDDI data model. In our work, we have made use of UNSPSC categorization as it is widely adoped. As mentioned earlier, our POM database stores all UNSPSC categorization with their name and code. The structure of the databases is shown in Figure 4.6. The details of it are self explanatory.

We also support browsing the category so that the service requestor can choose the particular category s/he is interested. Once the service requestor selects the list of

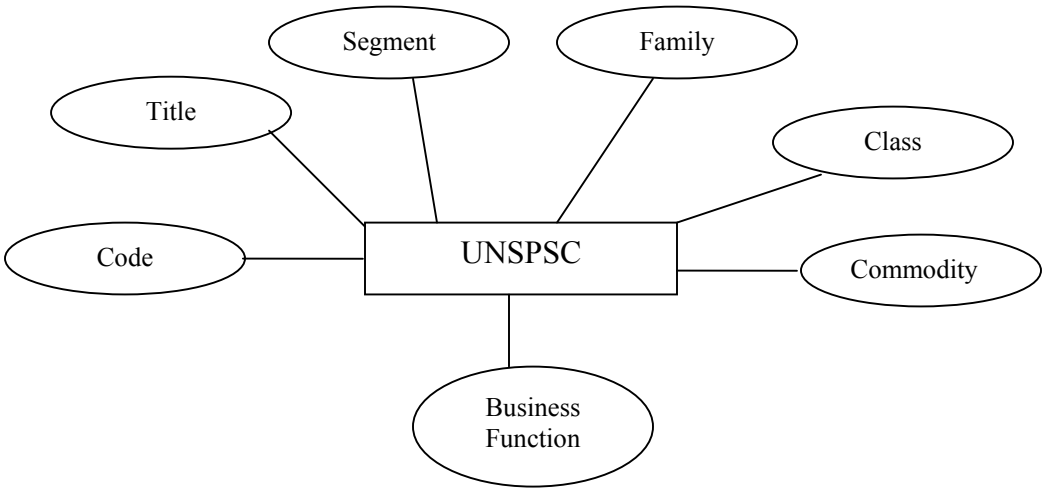

Figure 4.6 UNSPSC Database Schema

categories s/he is interested in. We form a category bag to query the UDDI registry to get the list of registered services under each category. The structure of the category bag is shown in Figure 4.7.

```
<CategoryBag>
<name>Travel Facilitation</name>
<code>90.12.00.00.00</code>
</CategoryBag>
```

Figure 4.7 Category Bag

CHAPTER 5
COMPOSITE SERVICE SPECIFICATION (WSFL) GENERATOR

After the Service Composer generates a composite service template satisfying a requestor's request, the next step is to generate a corresponding Composite Service Specification document. In our work, we have used IBM's WSFL (Web Service Flow Language) as our composite service specification language. In our system, the discovered composite service template is used by the Intelligent Registry to automatically generate a Composite Service Specification document in WSFL, which is then stored in its persistent storage. The service operation nodes in the template are mapped to activities in a Web service process model. In this chapter, the design and implementation of the Composite Service Specification Generator is explained in detail with an example.

5.1 A Travel Order Example

In our example, we consider a traveler looking for a Web service to satisfy her/his travel needs. We assume that s/he is looking for the following services that include air ticket reservation, car rental reservation and hotel room reservation. The traveler makes a request to the intelligent registry. After selecting the desired services and giving constraint information, the Service Composer automatically finds any needed additional services to satisfy the request. In this example, we assume that the Service Composer automatically finds a Hotel Matching service needed to reserve hotel. Then, it generates a composite service plan in the form of a Composite Service Template. The Composite Service Template is given as input to the CSS generator. In Figure 5.1, we give the structure of the Composite Service Template for the Travel example.

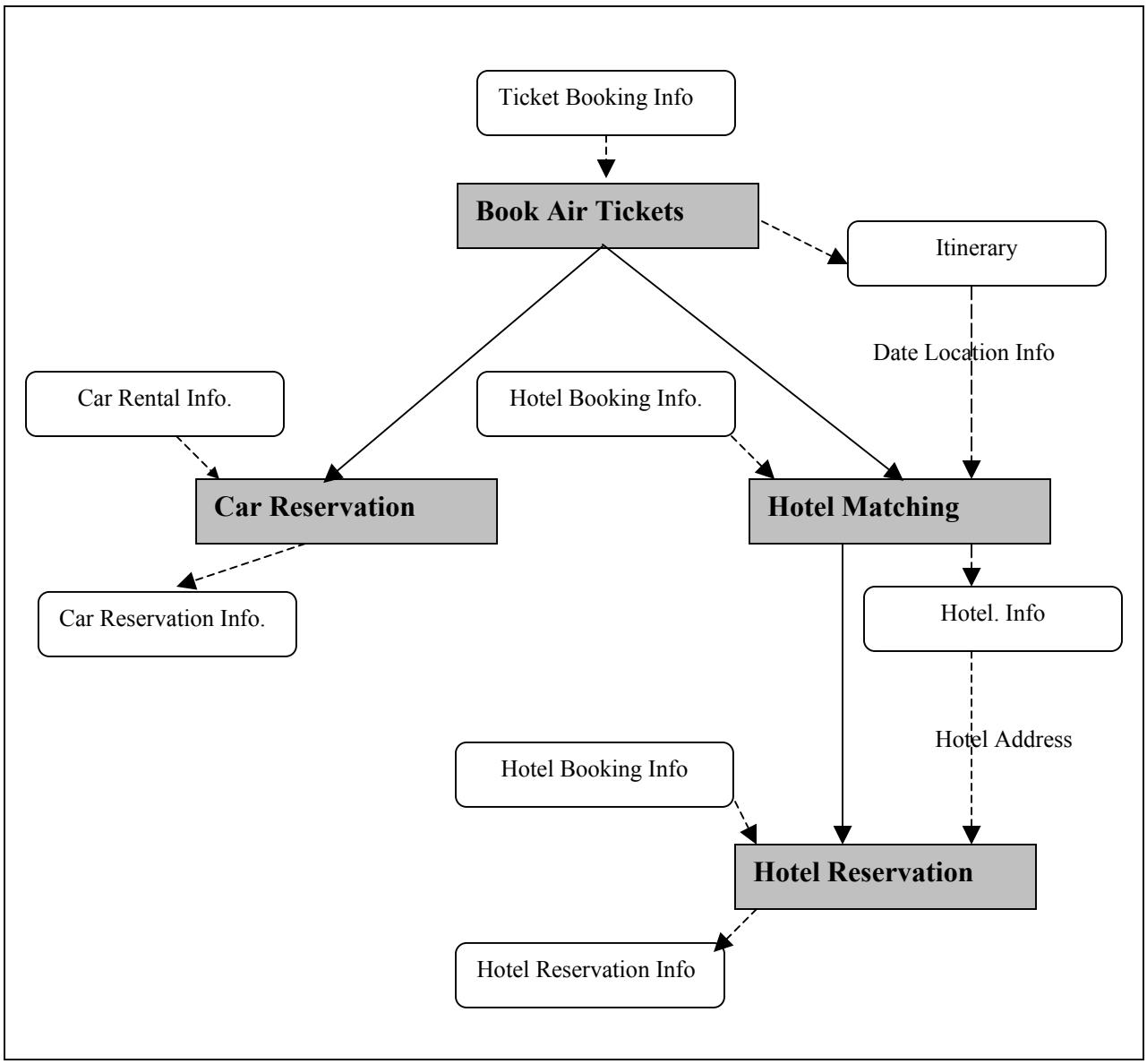

Figure 5.1 Composite Service Template

In the above figure, the dotted lines represent the datalinks, mapping the input and output attributes with their corresponding operations and the dark lines represent the operation execution sequences. The shaded boxes represent the service operations that need to be invoked. The details of the input and output parameters are as follows:

1. Ticket Booking Information: This parameter contains details regarding traveler's information, number of passengers, travel date and time, destination and arrival place.

2. Car Rental Information: This parameter contains details regarding Traveler Information, number of days to be booked, pick up address and pick up date.

3. Hotel Booking Information: This parameter contains details regarding Traveler's room preference, the number of days s/he wants to reserve and the number of rooms.

4. Itinerary: This parameter has information like ticket number, flight details and cost of the ticket.

5. Hotel Reservation Information: This parameter has information regarding hotel address, start and end time, confirmation number and cost of booking.

6. Car Reservation Information: This parameter has information regarding car rental company name, reservation number and cost for renting the car.

7. Date Location Information: It is an entity of itinerary and it is given as input to Hotel Matching as the hotel should be reserved after the airline booking has been done.

8. Hotel Address: It is an output entity of Hotel Matching. It has information regarding the hotel that needs to be booked and is given as input to the Hotel Reservation operation.

As mentioned earlier, a Composite Service Template is an acyclic graph. Each operation and data node (input and output parameter) is mapped as a "Vertic" with the type attribute of the Vertic denoting whether it is an operation or a data entity node. Each datalink is represented as an edge between a data node vertic and an operation node vertic. Each control link is represented as a separate data structure. The details of the data structure of the graph are explained in Chapter 6.

5.2 Design and Implementation of CSS (WSFL) Generator

The main components of the Composite Service Specification Generator and their interactions are shown in Figure 5.2

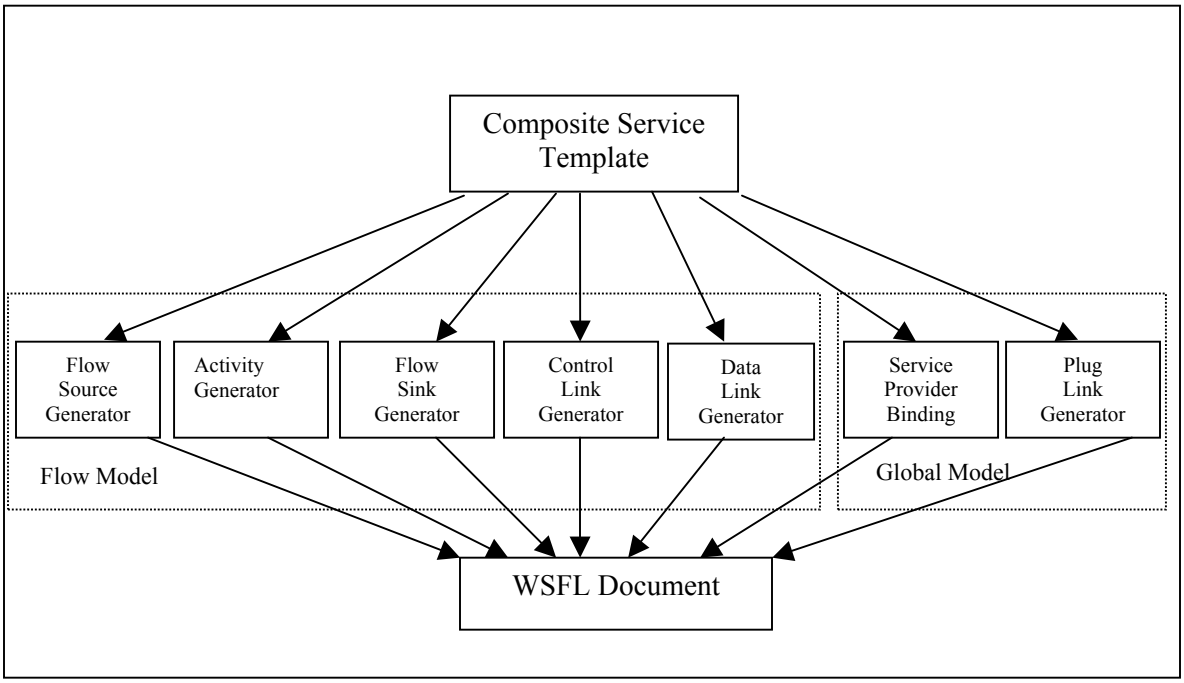

Figure 5.2 Architecture of CSS Generator

Composite Service Specification Generator is initiated by instantiating the Composite Service Template. The CSS Generator is implemented by using Java API for XML Processing (JAXP) that creates a WSFL document. WSFL document consist of two types of Web service compositions: flow model and global model. The flow model is constructed first, and then the global model is constructed. Figure 5.3 gives the sequence of activities in the WSFL construction process.

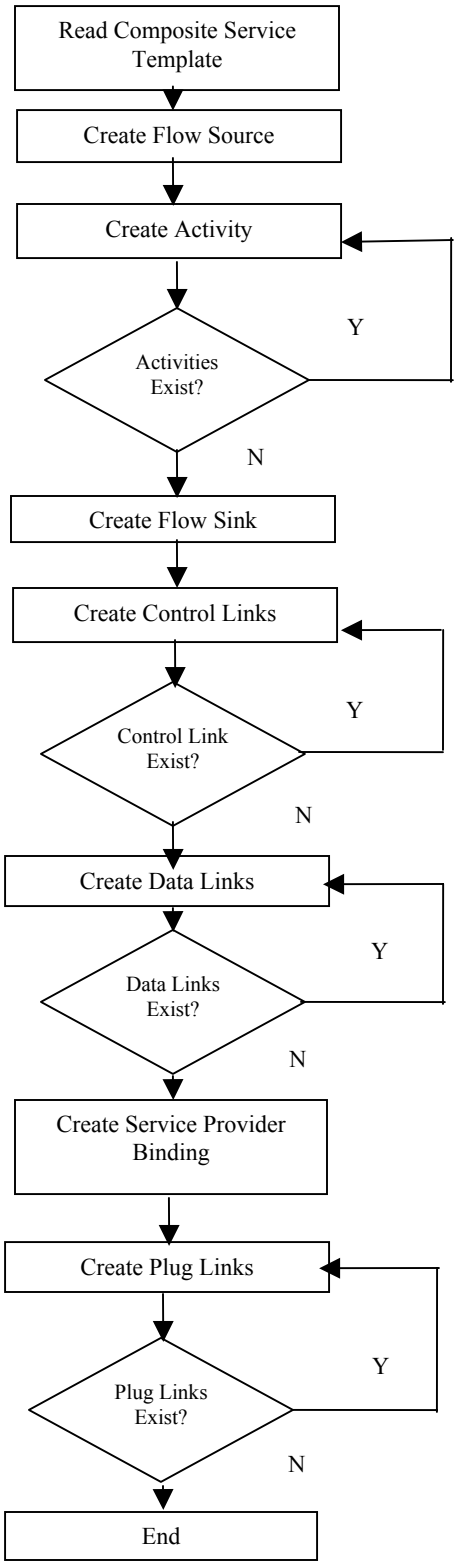

Figure 5.3 Sequence of activities in WSFL Construction

5.3 Flow Model Generator

The WSFL document consists of definition tag which points to the default namespace. It consists of an optional import element that points to the definition document. The definition document has information regarding the complex types used in the WSFL document. In WSFL, each operation is performed by a business partner and is captured by the concept of service provider. A service provider type is a set of named port types. The service provider type is mapped to the corresponding real service in the Global document. Figure 5.4 shows the service provider and import elements created by CSS generator.

```
<definitions targetNameSpace="http://www.cise.ufl.edu/">
<import
location="http://taipei:8080/examples/demo2wsflFiles/TravelOrder/TravelBookingDefs.wsdl"
        namespace="http://www.cise.ufl.edu/"/>
<flowModel name="TravelBooking" serviceProviderType="TravelBookingFlow">
        <serviceProvider name="rentACarService" type="rentACarType"/>
        <serviceProvider name="findBestHotelService" type="findBestHotelType"/>
        <serviceProvider name="orderAirTicketService" type="orderAirTicketType"/>
        <serviceProvider name="reserveRoomsService" type="reserveRoomsType"/>
```

Figure 5.4 Import and Service provider elements for “Travel Example”

Flow Model consist of the following components

- Flow source
- Activity
- Flow sink
- Control Links
- Data Links

The following sections will explain how these structures are generated.

5.3.1 Flow Source Generator

Flow Source in a WSFL document corresponds to the input parameters of all operations that can be obtained from the service requestor. In our Travel example, Ticket Booking Information, Car rental Information and Hotel Reservation Information

constitute the flow source. The flow source element for “Travel Order” example is shown in Figure 5.5. It also shows the message and its input and output part names.

```
<flowSource name="travelflowSource">
      <output name="TravelOrderflowSource
                message="TravelOrderRequest"/>
</flowSource>

<message name = "TravelOrderRequest">
     <part name = "customerData" type = "CustomerInfo"/>
    <part name = "ticketData" type ="TravelInfo"/>
    <part name = "hotelData" type = "HotelBookingInfo"/>
    <part name = "carData" type = "CarBookingInfo"/>
</message>
```

Figure 5.5 FlowSource element for “Complete Travel Order Example”

The Flow Source Generator scans through the Composite Service Template and finds all flow source elements and generates the document elements corresponding to Figure 5.3. The type attribute refers to the type of the message. It can be a simple/complex attribute. If it is a complex attribute, its corresponding definition will be included in the WSFL definition document. In our example, all attributes are of complex type. The complex type definition of the “CustomerInfo” is given in Figure 5.6.

```
<complexType name="CustomerInfo">
        <sequence>
                <element name="customerName" nillable="true" type="xsd:string" />
                  <element name="address" nillable="true" type="Address" />
                  <element name="creditCardNumber" type="xsd:int" />
                  <element name="contactNumber" nillable="true" type="xsd:string" />
        </sequence>
</complexType>
```

Figure 5.6 Definition of the Complex Type Customer Info

5.3.2 Activity Generator

An activity represents a business task to be performed as a single step within the context of a business process. Operations in the composite service template are the

concrete implementations of the abstract activities to be performed. Activities corresponds to a vertic's whose type is an operation in the composite service template. Each activity has a signature that is related to the signature of the operation used as the implementation of the activity. The input and output messages on an activity correspond to the input and output parameters of the operation. Figure 5.7 shows an example of an activity for the travel order example.

An activity is implemented by interacting with the service provider of the corresponding operation. The service provider is specified by a nested <performedBy> element. The implementing operation is specified through a nested <implement> element, which has an <export> element pointing to the corresponding WSDL operation name and port type. The request and response messages of the activity points to the input and output attribute names of the operation as shown in Figure 5.8.

```
<activity name = "BookAirTickets">
        <input name = "InBookAirTickets"
                message = "BookAirTicketsRequest"/>
        <output name = "OutBookAirTickets"
                message = "BookAirTicketsResponse"/>
        <performedBy serviceProvider = "EAirticketOrderService"/>
        <implement>
                <export>
                        <target portType = "CompleteTravelOrderPT"
                                operation = "orderFlightTickets"/>
                </export>
        </implement>
</activity>
```

Figure 5.7 Activity Structure Example

```
<message name="BookAirTicketsRequest">
        <part name="cust1" type="CustomerInfo"/>
        <part name="departure1" type="xsd:string"/>
        <part name="destination1" type="xsd:string"/>
        <part name="noOfPassengers1" type="xsd:int"/>
        <part name="depTime1" type="DateAndTimeInfo"/>
</message>

<message name="BookAirTicketsResponse">
        <part name="flightReservationNo" type="xsd:string"/>
        <part name="flightSchedule" type="FlightSchedule"/>
        <part name="arrivalDate" type="xsd:dateTime"/>
</message>
```

Figure 5.8 Request and Response Messages in WSFL

5.3.3 Flow Sink Generator

The Flow Sink in the WSFL document corresponds to the data that is returned once the activity is completed. The input of flow sink can be linked to the output of the activities in the flow model through dataLinks to indicate that the result of an activity contributes to the result of the overall flow. The Flow Sink Generator finds the output elements of the activities and groups them to a vector to form a Flow sink element. The structure of the Flow sink element for the travel order example is shown in Figure 5.9.

```
<flowSink name = "travelFlowSink">
        <input name = "InTravelFlowSink"   message = "TravelOrderResponse"/>
</flowSink>

<message name="TravelOrderResponse">
        <part name="flightResNum" type="xsd:string" />
        <part name="flightSchedule1" type="FlightSchedule"/>

        <part name="hotelName4" type="xsd:string"/>
        <part name="hotelAddress1" type="Address" />
        <part name="hotelResNumber" type="xsd:string"/>

        <part name="carPickAddr" type="Address" />
        <part name="carRetTime" type="DateAndTimeInfo" />
        <part name="carCompInfo" type="CarReservationInfo" />
</message>
```

Figure 5.9 Flow Sink Element Structure

5.3.4 Control Link Generator

Control links represent edges in the Composite Service Template that correspond to the flow of control in the business process. A control link is a directed edge pointing from its source activity to the destination activity. In a <controlLink> element, mandatory source and target attributes are used to name the linked activities. The Composite Service Template has a data structure to represent the control links, which is discussed in Chapter 6. The control structure of the flowModel is acyclic and the Composite Service Template does not allow a loop in its flow structure. The control structure as captured by the Composite Service Template is shown in Figure 5.10 and the corresponding control link element generated by the Control Link Generator is shown in Figure 5.11.

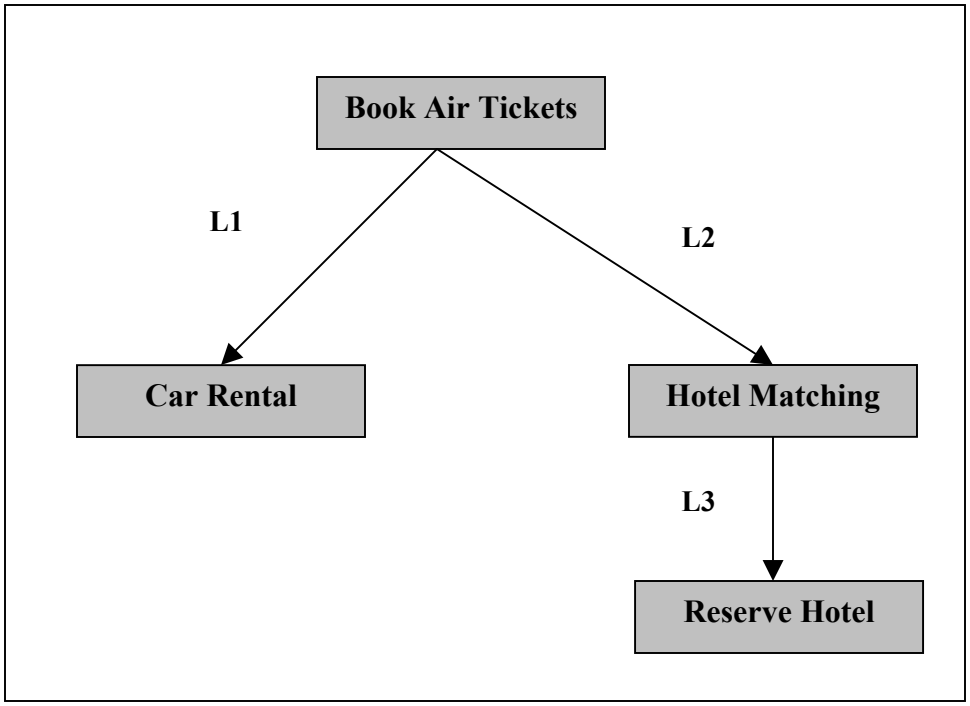

Figure 5.10 Control Structure for Travel Example

```
<controlLink name = "L1"
                source = "BookAirTickets"
                target = "DoHotelMatching"/>

<controlLink name = "L2"
                source = "BookAirTickets"
                target = "RentCar"/>

<controlLink name = "L3"
                source = "DoHotelMatching"
                 target = "ReserveHotel"/>
```

Figure 5.11 Control Links generated by Control Link Generator

5.3.5 Data Link Generator

A <dataLink> element represents the exchange of information between activities. In the Composite Service Template, a data link is represented by as an edge between a data vertic and operation vertic or vice versa. The Data Link Generator generates the <dataLink> elements in three phases. First, it generates the data links for all edges from the flow Source to activities. Second, it generates the data links for all edges between activities. Finally, it generates the data links for all edges from the activities to the flow sink. The attributes source and target represent the linked activities. For every existing data link between two activities, there must be a control path between them. The mapping between the source and target elements is defined by using a <map> element as per WSFL specifications. Figure 5.13 shows a data link element created for the “Travel” example scenario.

```
<dataLink name = "dataLink3" source = "travelFlowSource"
                                    target = "ReserveHotel">
<map sourceMessage= "TravelOrderFlowSource"
        targetMessage = "InReserveHotel"
        sourcePart = "customerData"
        targetPart = "customerInformation"/>

<map sourceMessage= "TravelOrderFlowSource"
        targetMessage = "InReserveHotel"
        sourcePart = "numOfRooms"
        targetPart = "roomQuantity2"/>

<map sourceMessage= "TravelOrderFlowSource"
        targetMessage = "InReserveHotel"
        sourcePart = "numOfNights"
        targetPart = "nightQuantity2"/>

<map sourceMessage= "TravelOrderFlowSource"
        targetMessage = "InReserveHotel"
        sourcePart = "roomType"
        targetPart = "roomType2"/>
</dataLink>
```

Figure 5.12 Data Links generated by Data Link Generator

5.4 Global Model generator

Global Model binds the "proxy" service providers and activities in the flow model to service providers and operations that are actually used to realize these proxies. Global Model Generator consist of two components

- Service Provider Binder
- Plug Link Generator

5.4.1 Service Provider Binder

The <serviceProvider> element binds the service providers with their corresponding locations. This is done by using the <locator> element. The locator type element supports static, local, uddi and mobility binding. We have used static binding in our example. In static binding, the service implementation is specified via the service

attribute, which references a WSDL or WSFL definition of a service. Figure 5.13 shows the service provider bindings used in our example.

```
<serviceProvider name = "EAirticketOrderService"
                        type = "TravelTicketOrderType">
<locator type = "static"
          service = "http://taipei:8080/examples/trav-wsdls/site1/EAirticketOrder.wsdl"/>
</serviceProvider>

<serviceProvider name = "PerfectHotelMatchingService"
                        type = "HotelMatchingType">
<locator type = "static"
          service = "http://taipei:8080/examples/trav-wsdls/site2/PerfectHotelMatching.wsdl"/>
</serviceProvider>
```

Figure 5.13 Service Provider Bindings

5.4.2 Plug Link Generator

The plugLink element binds an operation from service provider's interface in the flow model to an operation implemented by the service provider. Figure 5.14 shows plugLink elements of Travel order example.

```
<plugLink>
          <source serviceProvider = "TravelBooking"
                    portType = "CompleteTravelOrderPT"
                    operation = "orderFlightTickets"/>
          <target serviceProvider = "EAirticketOrderService"
                    portType = "EAirticketOrder"
                     operation = "orderAirTicket"/>
</plugLink>
```

Figure 5.14 Plug Link Elements

In the above example, the "proxy" port type and operation names used in the flow model are mapped to the real port type and operation names of the corresponding WSDL document. This is achieved by parsing the WSDL document at runtime.

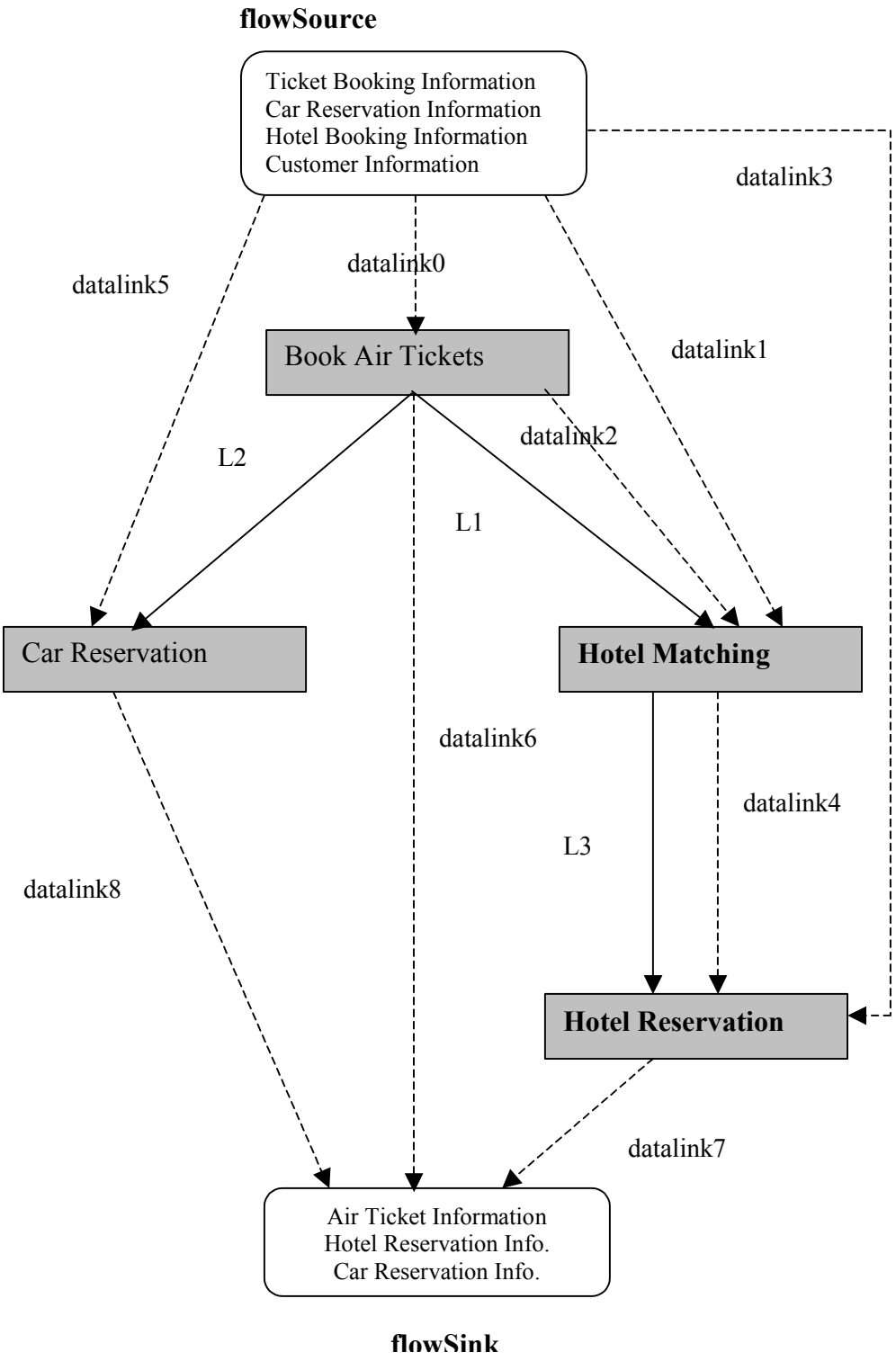

Figure 5.15 WSFL Flow structure of Travel Order Example

The plug link indicates that the TravelBooking service provider's orderFlightTickets operation is bound to the EAirticketOrderService service provider's orderAirTicket operation. This binds the flow model's public interface to the orderAirTicket operation.

Figure 5.15 gives a pictorial description of the flow model for the Travel Order composite service as captured by the WSFL document. In the figure, the dotted lines represent dataLinks and the dark lines represent controlLinks. The shaded boxes represent activities that actually contain service operations that need to be invoked.

A flowSource contains a list of output parameters that a composite service will generate.

The dataflow details for this example is listed below:

- datalink0 maps customer information and ticket booking information parameters from flowSource to the BookAirTickets activity .
- datalink1 maps CustomerInfo and HotelBookingInfo from flowSource to the HotelMatching activity.
- datalink2 maps DateLocationInfo, which is one of the output of the activity BookAirTickets, to the activity HotelMatching.
- datalink3 maps CustomerInfo and HotelBookingInfo from flowSource to HotelReservation.
- datalink4 maps HotelInformation from the HotelMatching activity to the HotelReservation activity.
- datalink5 maps CustomerInfo and CarRentalInfo parameters from flowSource to the CarReservation activity.
- datalink6 maps output parameter, Itinerary of the BookAirTickets activity to flowSink.
- datalink7 maps output parameter, HotelReservationInfo of the HotelReservation activity to flowSink.

- datalink8 maps output parameter, CarRentalReservationInfo of the CarReservation activity to flowSink.

A controlLink is a directed edge that describes the order in which activities will have to be performed. All activities are connected together through control links. Details regarding control links present in this example are as follows:

- controlLink L1 specifies that activity HotelMatching should be scheduled only after activity BookAirTickets is completed.
- controlLink L2 specifies that activity CarReservation should be scheduled only after activity BookAirTickets is completed
- controlLink L3 specifies that activity HotelReservation should be scheduled only after activity HotelMatching is completed.

A flowSink contains a list of output parameters that a composite service will generate.

CHAPTER 6
IMPLEMENTATION DETAILS

The components of the Intelligent Registry described in this thesis have been implemented using JDK 1.4 on a windows 2000 platform. The Web server used is Tomcat 4.0.4. The component databases have been implemented using the Persistent Object Manager (POM). We have made use of IBM Web services toolkit 3.3 [15] to establish the connection with the UDDI registry. This chapter is organized into three sections. Section 6.1 gives the data structures used for our system. Section 6.2 gives the screen shots of our system.

6.1 Data Structures and Class Definitions

As we have explained in Chapter 4, we store the UNSPSC data in the Persistent Object Manager (POM). POM maps objects to relational tables. We show the class structure of the UNSPSC object in Figure 6.1.

```
public class Unspsc
{
        public String code;
        public String title;
        public int segment;
        public int family;
        public int clas;
        public int commodity;
        public int businessfunction;
}
```

Figure 6.1 Class Definition for UNSPSC Object

The variable details of the class are as follows:

1. code: This variable represents the unique UNSPSC code for the product. This code is used to form the category bag while querying the UDDI registry.
2. title: This variable represents the title of the product.
3. segment: A two character numerical field used to represent the logical aggregation of families. The division of the code into five hierarchical levels allows easy browsing of the category and gives better performance.
4. family: It is a two-character field that specifies inter- related commodities within a segment.
5. class: This two-character numerical field represents commonly used commodities.
6. commodity: It presents a group of substitutable products.
7. business function: It represents the function performed by an organization in support of the commodity.

The corresponding "create table" command for the UNSPSC object is shown in Figure 6.2.

```
CREATE TABLE UNSPSC
  (
    OID INT NOT NULL,
    code VARCHAR(255) ,
    title VARCHAR(255) ,
    segment INT ,
    family INT ,
    clas INT ,
    commodity INT ,
    businessfunction INT
  )
```

Figure 6.2 Create Table command for UNSPSC Object

Each table is assigned a primary key attribute when it is created. This is called the Object identifier (OID) column. This key can be used to update or delete object once it is created.

Each Web service has many related operations. Each operation has input and output parameters. We store the various services, operations, constraints and

operation execution sequence selected by the service requestor using composition. It is explained in the following sections.

6.1.1 WSDLService Object

This object represents the selected Web service and its details. The class definition is shown in Figure 6.3. The variables are as given below:

```
public class WSDLService
{
        String serviceIntfName;
        String serviceIntfKey;
        Vector operations;
}
```

Figure 6.3 Class Definition for WSDLService Object

1. serviceIntfKey: This variable gives the key of the service interface that this particular service implements. This is a Universal Unique Identifier (UUID) generated by the UDDI registry at the time service is registered with the UDDI registry. This is obtained by using the UDDI find mechanism.

2. serviceIntfName: This variable represents the interface name for the corresponding service interface key.

3. operations: This vector contains the list of operations that belongs to this service.

The detail of the operation structure is explained in Section 6.1.2

6.1.2 WSDL Operations Object

WSDL Operations class encapsulates all the details about an operation. It includes details about the operation name, input and output parameters of an operation. The class definition is shown in Figure 6.4. It contains the following variables.

```
public class WSDLOperations
{
        String operationName;
        Vector inputs;//vector of DataEntity
        Vector outputs;//vector of DataEntity
}
```

Figure 6.4 Class Definition for WSDLOperations Object

4. operationName: This variable stores the operation name.

5. inputs: This variable is a vector. It stores a list of all the input parameters to the operation in the form of a data entity. The structure of the data entity is explained in the next section.

6. outputs: This variable is a vector. It also stores the output parameter in the form of data entity.

6.1.3 WSDLDataEntity Object

This object stores the operation's input and output parameter information. If the input or output parameter is of a complex type, it also stores information regarding it. The class structure is shown in Figure 6.5. The variable details are as follows:

```
public class WSDLDataEntity
{
        String dataEntityName;
        String dataEntityType;
        Constraint constraint;
        boolean complex;
        WSDLDataEntity seqlist;
}
```

Figure 6.5 Class Definition for WSDLDataEntity Object

7. dataEntityName: It represents the input or output parameter name.

8. dataEntityType: It represents the type of the data entity. If the data Entity is of a compex type, this will have the null value. Otherwise, it will have the corresponding type information such as string, int, etc.

9. constraint: It contains constraint information regarding the attribute as specified by the service requestor. The details of it is explained in Section 6.1.4.

10. complex: It is of Boolean type. It is set to true if the attribute is of a complex type.

11. seqlist: It will have the null value if the attribute is of a simple type. If the attribute is of a complex type, it points to the simple attributes. Basically, it is a linked list structure.

6.1.4 AttributeConstraint Object:

This object is used to specify the attribute constraints. It can be used for constraint specification of input, output, operation and service attributes. The class definition is shown in Figure 6.6. It consists of the following variables:

```
public class AttributeConstraint implements java.io.Serializable
{
        public String name;
        public String type;
        public String keyword;
        public boolean negotiable;
        public Vector valueList;
        public int priority;
        public boolean nomatch;
}
```

Figure 6.6 Class Definition of the AttributeConstraint Object

12. name: This variable gives the name of the particular attribute of the service, operation, input or output attribute that this object is a constraint specification of.

13. type: This variable gives the data type of the attribute. The data types can be string, integer, float, date, time, datetime, or duration.

14. keyword: This variable gives the keyword associated with each attribute. The keyword helps in identifying the kind of constraint being specified. The constraints can be specified on a range of values, on an enumeration of values or on a single value along with comparison operators. So, the keywords can be range, enumeration, =, !=, >, <, >=, and <=.

15. valueList: This variable gives the list of values that a particular attribute can take. The form of this list depends on the keyword specified. If the keyword is range, this list, implemented as an object of the class java.util.Vector, contains two elements that denote the start and the end of the range respectively. If the keyword is enumeration, this list contains all the valid attribute values. If the keyword is one of the six comparison operators, the list just contains a single value.

16. negotiable: This variable determines whether a particular attribute constraint is flexible or not. If a constraint is flagged as non-negotiable and it is not satisfied, constraint matching terminates and the Constraint Satisfaction Processor reports a constraint mismatch.

17. priority: In case an attribute constraint is negotiable, this variable determines its priority, and hence its order of evaluation. As stated above, non-negotiable attributes are evaluated first followed by attributes in the order of decreasing priority.

18. nomatch: This variable is used to differentiate those attributes for which there is a constraint specification from those for which there is none. When specifying constraints for an operation, for example, a service provider or a service requestor may not want to specify constraints on some attributes. This variable is set to true in these cases.

6.1.5 Operationsorder Object

This object stores information regarding the order of operation execution given by the service requestor. The class declaration is shown in Figure 6.7. The variable details are as follows:

```
public class OperationsOrder
{
        String name;
        Vector succeedingOperations;
}
```

Figure 6.7 Class Definition for Operations Order

The variable “name” stores the variable name and the “succeeding operations” variable is a vector, which stores the list of operations that follows.

6.1.6 Csn Object

This object encapsulates the composite service graph. As we mentioned in Chapter 5, the operations and data links are expressed in the form of “vertic” and edges

respectively. The class definition of Csn is shown in Figure 6.8. Information necessary to construct WSFL document is encapsulated in this class. The variable details are as follows:

```
public class Csn
{
        public VerticHashtable vertices;
        PairVector edgeVector;
}
```

Figure 6.8 Class Definition for Csn

19. vertices: Vertices contains an hash table of data nodes and operations. All data nodes and operations are kept in a hash table for easy lookup. “Vertices” has a hash table of “Vertic”s. Its data structure is explained in Section 6.1.7. The class VerticHashtable extends the hash table and provides all functionalities of a hash table.

20. edgeVector: EdgeVector is a vector, which contains all the edges in the composite service graph. They represent data links between a data node and an operation or vice versa. PairVector extends vector and provides all functionalities of vector. All edges in the graph are stored in this edgeVector. The data structure of an edge is explained in Section 6.1.8.

6.1.7 Vertic Object

Vertic object represents both data nodes and operations in the composite service graph. It has a type attribute, which denotes whether Vertic is a data node or an operation node. The variable details are as follows:

```
public class Vertic
{
        public static int DATAENTITY = 0;
        public static int OPERATION = 1;
        public int type;
        public String name;
        public Edge headInEdge;
        public Edge headOutEdge;
        String tModelKey;//key for this service
        String interfaceKey;
}
```

Figure 6.9 Class Definition for Vertic Object

21. type: The type variable denotes whether the “vertic” is a data node or an operation node. ‘1’ represents an operation and ‘0’ represents a data entity node.

22. name: “Name” variable gives the name of the data entity or operation.

23. headInEdge: HeadInEdge represents an edge which is a linked list structure. It stores all the incoming edges of the node.

24. headOutEdge: HeadOutEdge represents an edge which is a linked list structure. It stores all the incoming edges of the node. The data structure of “edge” is explained in Section 6.1.8.

25. tModelKey: It gives the tModel implementation key for the node.

26. interfaceKey: This variable has the interface key for the node. It is a UUID and is unique.

6.1.8 Edge Object

The Edge Object represents edges in the composite service graph. It consists of the names of the two corresponding nodes. Edges represent data links in a composite service graph. Figure 6.10 shows the class definition.

```
public class Edge
        {
        boolean visited;
        protected String start;
        protected String end;
        }
```

Figure 6.10 Class Definition for Edge Object

The variable details are as follows:

27. visited: This variable has a Boolean value. It is used while parsing the data structure and marked true or false based on whether it has been visited or not.

28. start: “Start” variable has the name of the start vertic name of edge. This can be either a Data Node or an Operation node.

29. end: “End” variable has the name of the end vertic name of the edge. This can be either a Data Node or an Operation node.

6.1.9 MapConcept Object

This class is used to store all the input and output attributes of all operations during WSFL generation. It is used by the WSFL generator while generating the request and response messages for the flow model. The MapConcept Object is stores as a hash table for faster retrieval. Figure 6.11 shows the class definition.

```
public class MapConcept
        {
        String operationName;
        String partType;
        String partName;
        }
```

Figure 6.11 Class Definition for MapConcept Object

The variable details are as follows:

30. operationName: This variable stores the operation name of the attribute.
31. partType: This variable stores the type of the attribute.
32. partName: This variable stores the name of the attribute.

6.1.10 WSFLDocNDefCreator Object

This class uses JAXP (Java API for XML processing) [11] to construct the WSFL document and its definitions. It has various methods to parse the composite service template and to construct WSFL document from it. The generated WSFL document is shown in the appendix.

6.2 Screen Shots

We show step-by-step screen shots to show how the service requestor interacts with the query composer. As shown in Figure 6.12, the service requestor uses a general

term to express his/her request. For example, in the figure the service requestor uses the word “Travel”.

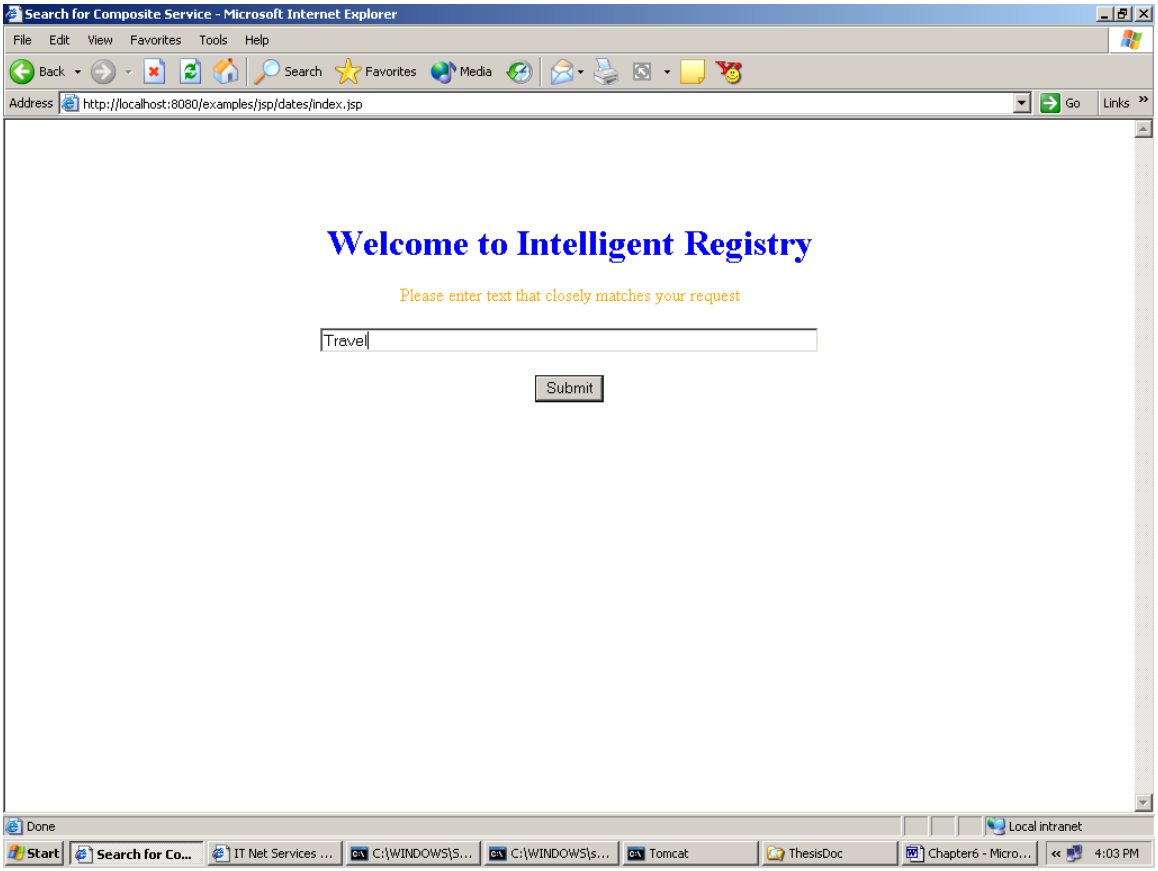

Figure 6.12 Screenshot showing Service Requestor expressing request

The service requestor can also enter a combination of words to express his/her request. Figure 6.13 shows the screen shot, which shows the matching category and the words generated by Jwordnet.

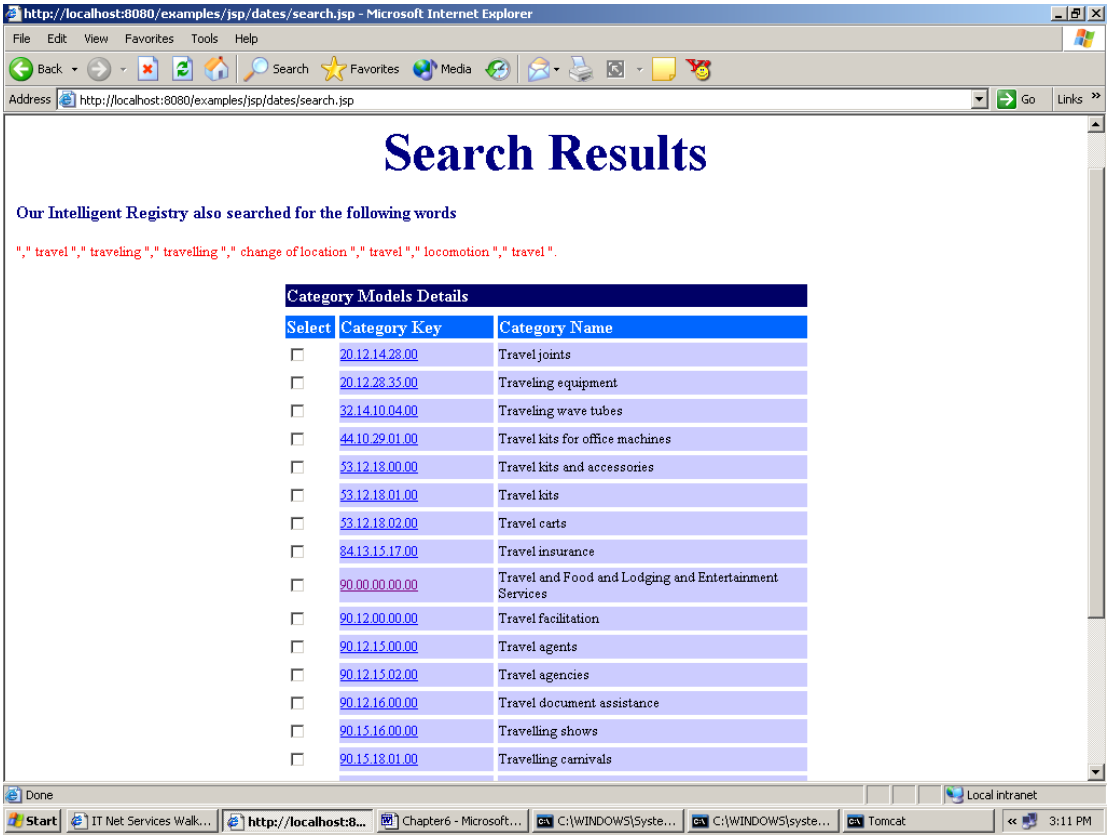

Figure 6.13 Screenshot showing matching UNSPSC

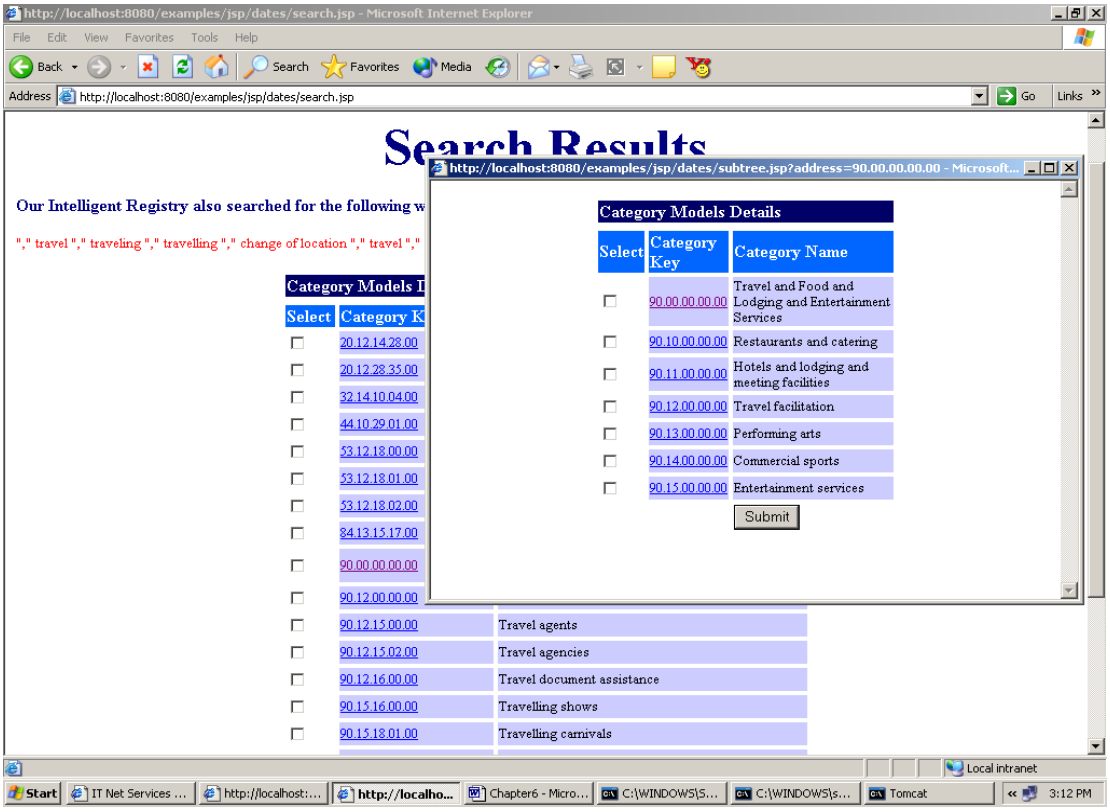

Figure 6.14 Screenshot showing Service Requestor browsing UNSPSC category

Figure 6.14 shows the service requestor browsing the sub-categories to find the sub- categories s/he is interested in. Once the service requestor selects the category s/he is interested in, the UDDI wrapper module contacts the UDDI registry to find all the services registered under the category. The query composer displays the tModelKey, tModel name and the WSDL URL of the service. The service requestor can then select the tmodel name and select among the operations offered by the service. Figure 6.16 shows the service requestor selecting the operations s/he interested in. Similarly the service requestor can select operations offered by various services.

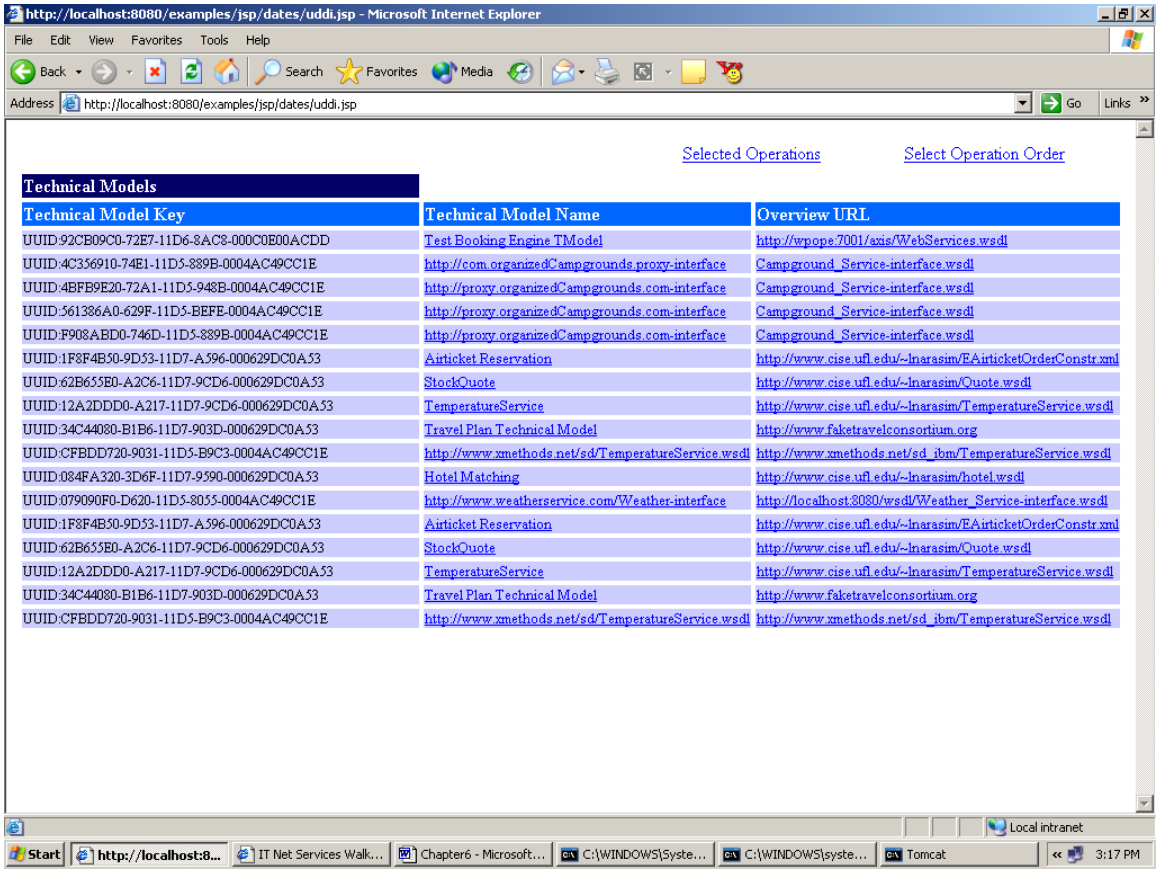

Figure 6.15 Screen shot showing list of Services.

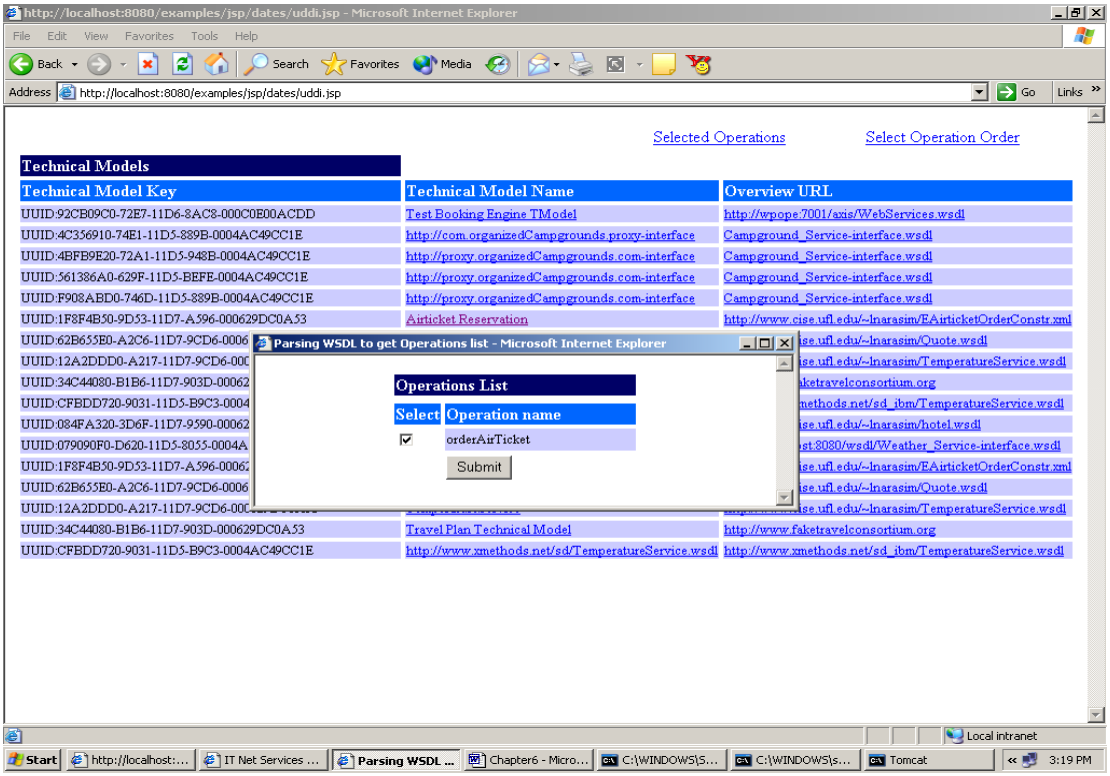

Figure 6.16 Screenshot showing Service Requestor selecting Operations

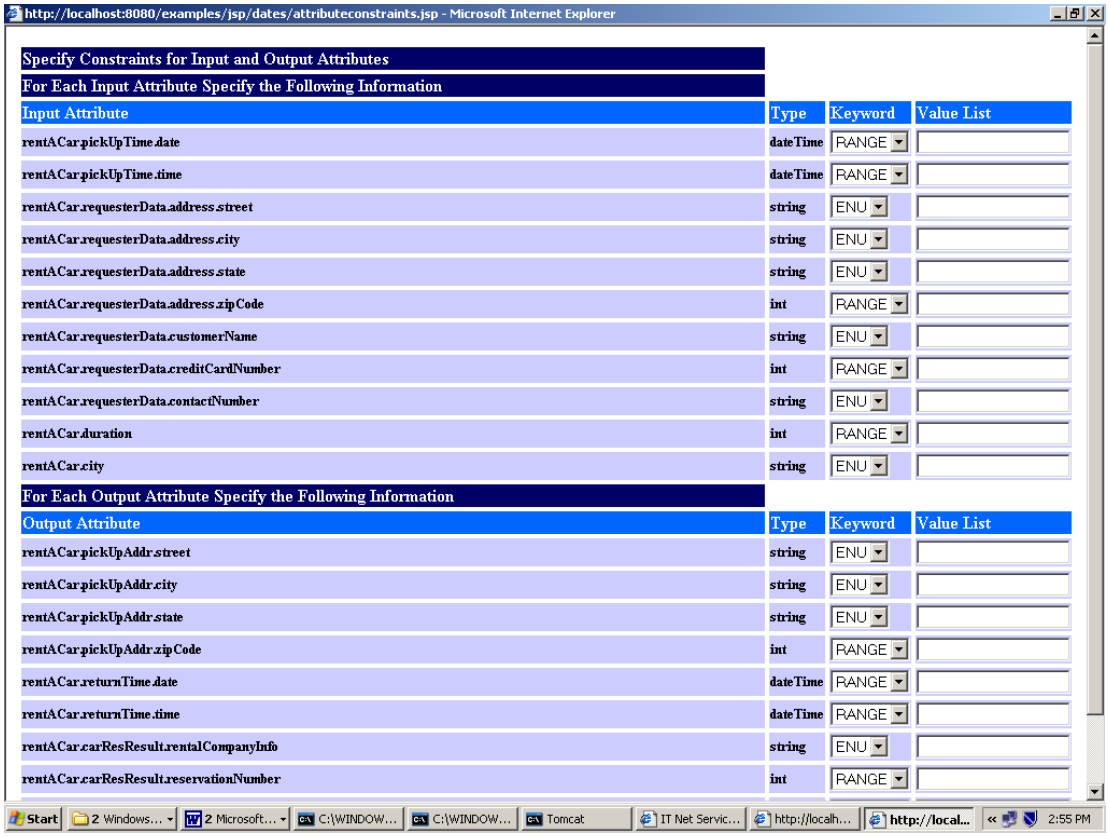

Figure 6.17 Screenshot showing Service Requestor specifying Attribute Constraints

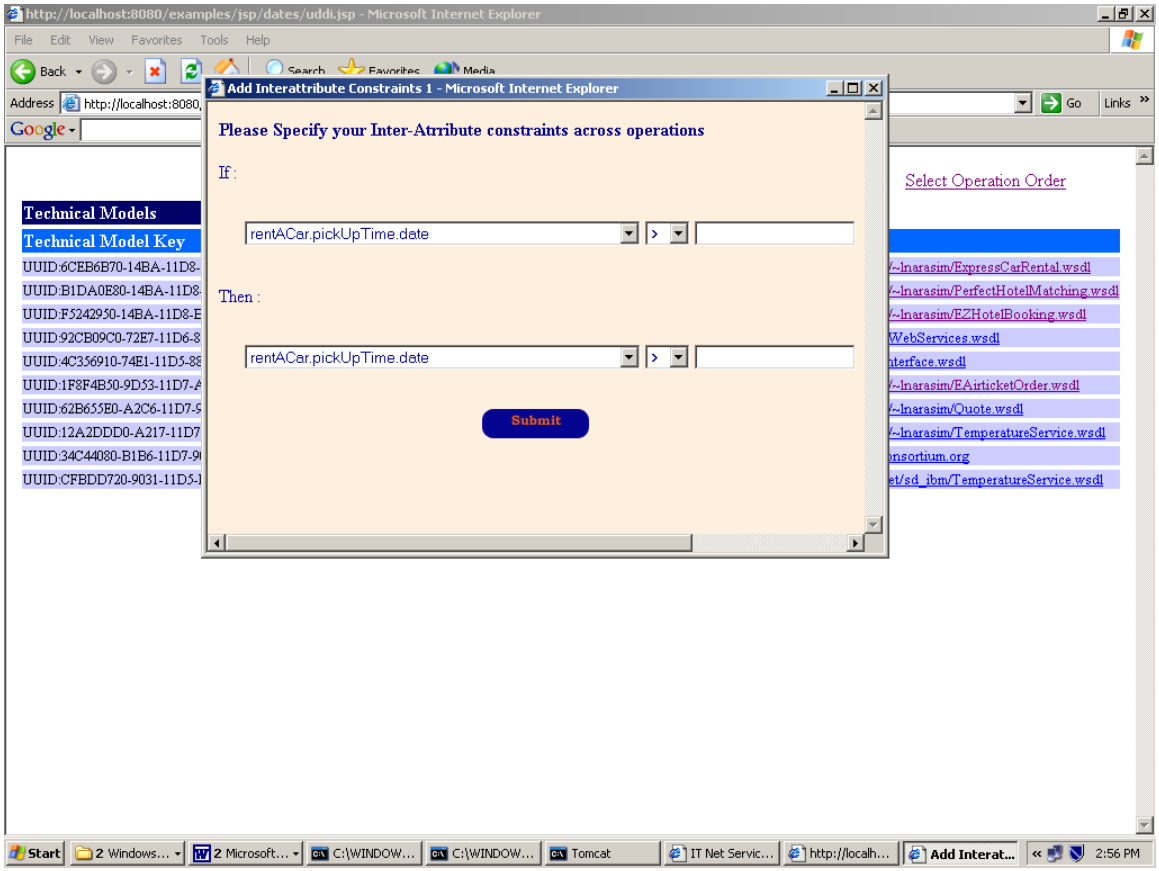

Figure 6.18 Screenshot showing Service Requestor specifying Inter-Attribute Constraints

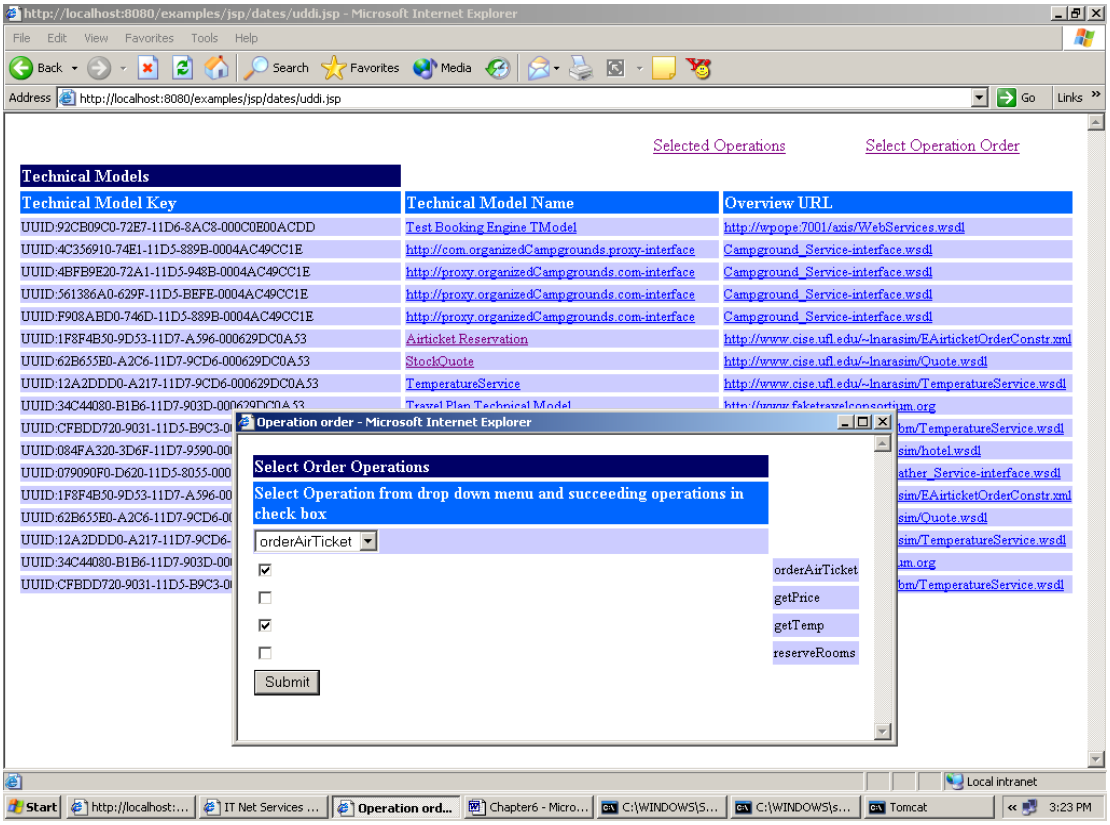

Figure 6.19: Screen shot showing Service Requestor expressing Operation Sequence

The service requestor can also give the execution sequence among operations. Figure 6.18 shows the service requestor expressing the operation execution sequence. The generated WSFL document is shown in Appendix. In this chapter, we have looked at the implementation details of our system along with some screen shots. In the next chapter we conclude our work.

CHAPTER 7
CONCLUSION

Web service composition can save an organization or user a lot of time and money because some Web services needed by the organization or user can be composed of registered services so that new Web services will not have to be developed. The current Web service model does not support dynamic Web service composition. Nor does it offer a mechanism to assist a service requestor to formulate his/her queries for composite services through user-system interaction. Such a mechanism will be very useful for service requestors to discover composite Web services.

In this work, we have described the design and implementation of a Query Composer and Composite Service Specification Generator to achieve dynamic Web service composition. The components of the Query Composer and Composite Service Specification (CSS) Generator were explained in detail. We have used the implemented system and services related to travel as the application domain to evaluate our query composition and CSS generation. We have found that the use of Wordnet and UNSPSC categorization is an effective way to deal with different terms that service requestors may use to search for service categories. We have also developed a user friendly interface for service requestors to specify constraint information and operation execution order. There are several composite service specification languages like WSFL, XLANG[16] and BPEL[17]. But none of them have been standardized by the W3C committee. We have made use of IBM's Web Service Flow Language as our composite service specification language. We believe that dynamic Web service discovery is a significant improvement

over existing Web service model. The grid services are becoming popular in e-business arena. We are exploring the usage of our Intelligent Registry and its components in grid service composition. We are also looking into grid service composition mechanisms.

APPENDIX
SERVICE REQUEST AND WSFL DOCUMENTS

This appendix presents the Service Request document generated by the Query Composer and the WSFL document generated by the Composite Service Specification Generator. The Service Request document is shown in Section 1. Section 2 presents the WSFL document.

A.1 Service Request document for Travel Booking

```
<?xml version="1.0" encoding="UTF-8"?>
<serviceRequest
xmlns="http://www.dbcenter.cise.ufl.edu/ServiceRequestSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~qliang/ServiceRequestSchem
a/ServiceRequest.xsd">
<complexType name="FlightSchedule">
<sequence>
<element name="depTime" nillable="true"type="DateAndTimeInfo"/>
<element name="arrTime" nillable="true" type="DateAndTimeInfo"/>
<element name="flightNo" nillable="true" type="xsd:string"/>
<element name="destination" nillable="true" type="xsd:string"/>
</sequence>
</complexType>
<complexType name="CustomerInfo">
<sequence>
<element name="address" nillable="true" type="Address"/>
<element name="customerName" nillable="true" type="xsd:string"/>
<element name="creditCardNumber" type="xsd:int"/>
<element name="contactNumber" nillable="true" type="xsd:string"/>
</sequence>
</complexType>
<complexType name="CarReservationInfo">
<sequence>
<element name="rentalCompanyInfo" nillable="true" type="xsd:string"/>
<element name="reservationNumber" type="xsd:int"/>
<element name="cost" type="xsd:int"/>
</sequence>
</complexType>
<complexType name="DateAndTimeInfo">
<sequence><element name="time" nillable="true" type="xsd:dateTime"/>
<element name="date" nillable="true" type="xsd:dateTime"/>
</sequence>
</complexType>
<complexType name="Address">
```

```
<sequence>
<element name="street" nillable="true" type="xsd:string"/>
<element name="city" nillable="true" type="xsd:string"/>
<element name="state" nillable="true" type="xsd:string"/>
<element name="zipCode" type="xsd:int"/>
</sequence>
</complexType>
<categorization categorySystem="UNSPSC">
<categoryCode>90.12.00.00.00</categoryCode>
<categoryCode>90.00.00.00.00</categoryCode>
<categoryCode>90.00.00.00.00</categoryCode>
</categorization>
<inputs>
<input>
<name>depTime1</name>
<ontologyID>depTime1</ontologyID>
<type>DateAndTimeInfo</type>
</input>
<input>
<name>noOfPassengers1</name>
<ontologyID>noOfPassengers1</ontologyID>
<type>int</type></input>
<input>
<name>destination1</name>
<ontologyID>destination1</ontologyID>
<type>string</type>
</input>
<input>
<name>departure1</name>
<ontologyID>departure1</ontologyID>
<type>string</type>
</input>
<input>
<name>cust1</name>
<ontologyID>cust1</ontologyID>
<type>CustomerInfo</type>
</input>
<input>
<name>pickUpTime</name>
<ontologyID>PickUpWhen_DateAndTimeInfo</ontologyID>
<type>DateAndTimeInfo</type>
</input>
<input>
<name>requesterData</name>
<ontologyID>cust1</ontologyID>
<type>CustomerInfo</type>
</input>
<input>
<name>duration</name>
<ontologyID>duration</ontologyID>
<type>int</type>
</input>
<input>
<name>city</name>
<ontologyID>PickUpCity_City</ontologyID>
<type>string</type>
</input>
```

```
<input>
<name>nightQuantity2</name>
<ontologyID>NumOfNights</ontologyID>
<type>int</type>
</input>
<input>
<name>roomQuantity2</name>
<ontologyID>NumOfRooms</ontologyID>
<type>int</type>
</input>
<input>
<name>roomType2</name>
<ontologyID>RoomType</ontologyID>
<type>string</type>
</input>
<input>
<name>date1</name>
<ontologyID>arrivalDate</ontologyID>
<type>dateTime</type>
</input>
<input>
<name>hotelName2</name>
<ontologyID>HotelName</ontologyID>
<type>string</type>
</input>
<input>
<name>customerInformation</name>
<ontologyID>cust1</ontologyID>
<type>CustomerInfo</type>
</input>
</inputs>
<outputs>
<output>
<name>arrivalDate</name>
<ontologyID>arrivalDate</ontologyID>
<type>dateTime</type>
</output>
<output>
<name>flightSchedule</name>
<ontologyID>flightSchedule</ontologyID>
<type>FlightSchedule</type>
</output>
<output>
<name>flightReservationNo</name>
<ontologyID>flightReservationNo</ontologyID>
<type>string</type>
</output>
<output>
<name>pickUpAddr</name>
<ontologyID>PickUpAddress_Address</ontologyID>
<type>Address</type>
</output>
<output>
<name>returnTime</name>
<ontologyID>ReturnWhen_DateAndTimeInfo</ontologyID>
<type>DateAndTimeInfo</type>
</output>
```

```
<output>
<name>carResResult</name>
<ontologyID>carResResult</ontologyID>
<type>CarReservationInfo</type>
</output>
<output>
<name>hotelReservationNo</name>
<ontologyID>HotelReservationNo</ontologyID>
<type>string</type>
</output>
<output>
<name>hotelName3</name>
<ontologyID>hotelName3</ontologyID>
<type>string</type>
</output>
</outputs>
<operations>
<operation ontologyID="orderAirTicket" operationName="orderAirTicket"
tModelKey="UUID:1F8F4B50-9D53-11D7-A596-000629DC0A53"/>
<operation ontologyID="rentACar" operationName="rentACar"
tModelKey="UUID:6CEB6B70-14BA-11D8-B936-000629DC0A53"/>
<operation ontologyID="reserveRooms" operationName="reserveRooms"
tModelKey="UUID:F5242950-14BA-11D8-B936-000629DC0A53"/>
</operations>
<flow name="T1" source="orderAirTicket" target="rentACar"/>
<flow name="T2" source="orderAirTicket" target="reserveRooms"/>
<constraints>
<constraint>
<name>orderAirTicket.arrivalDate</name>
<on>orderAirTicket.arrivalDate</on>
<type>dateTime</type>
<keyword><=</keyword>
<valueList>2003-12-2003</valueList>
</constraint>
<constraint>
<name>orderAirTicket.depTime1.time</name>
<on>orderAirTicket.depTime1.time</on>
<type>dateTime</type>
<keyword><</keyword>
<valueList>2003-12-20</valueList>
</constraint>
<interattribute_constraints>
<interattribute_constraint>
<name>IAC0</name>
<if>orderAirTicket.flightSchedule.depTime.date>=1.0718964E12</if>
<then>orderAirTicket.departure1=Orlando</then>
</interattribute_constraint>
</interattribute_constraints>
</constraints>
</serviceRequest>
```

A.2 WSFL document generated for Travel Booking Example Scenario

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNameSpace="http://www.cise.ufl.edu/">
```

```
<import location="http://taipei:8080/examples/demo2-
wsflFiles/TravelOrder/TravelBookingDefs.wsdl"
namespace="http://www.cise.ufl.edu/"/>
<flowModel name="TravelBooking"
serviceProviderType="TravelBookingFlow">
<serviceProvider name="rentACarService" type="rentACarType"/>
<serviceProvider name="findBestHotelService" type="findBestHotelType"/>
<serviceProvider name="orderAirTicketService"
type="orderAirTicketType"/>
<serviceProvider name="reserveRoomsService" type="reserveRoomsType"/>
<flowSource name="TravelBookingSource"><output
message="TravelBookingRequest" name="TravelBookingOutput"/>
</flowSource>
<activity name="rentACarActivity">
<input message="rentACarRequest" name="InrentACar"/>
<output message="rentACarResponse" name="OutrentACar"/>
<performedBy serviceProvider="rentACarService"/>
<implement>
<export>
<target operation="rentACarOperation" portType="rentACarPT"/>
</export>
</implement>
</activity>
<activity name="findBestHotelActivity">
<input message="findBestHotelRequest" name="InfindBestHotel"/>
<output message="findBestHotelResponse" name="OutfindBestHotel"/>
<performedBy serviceProvider="findBestHotelService"/>
<implement>
<export>
<target operation="findBestHotelOperation" portType="findBestHotelPT"/>
</export>
</implement>
</activity>
<activity name="orderAirTicketActivity">
<input message="orderAirTicketRequest" name="InorderAirTicket"/>
<output message="orderAirTicketResponse" name="OutorderAirTicket"/>
<performedBy serviceProvider="orderAirTicketService"/>
<implement>
<export>
<target operation="orderAirTicketOperation"
portType="orderAirTicketPT"/>
</export>
</implement>
</activity>
<activity name="reserveRoomsActivity">
<input message="reserveRoomsRequest" name="InreserveRooms"/>
<output message="reserveRoomsResponse" name="OutreserveRooms"/>
<performedBy serviceProvider="reserveRoomsService"/>
<implement>
<export>
<target operation="reserveRoomsOperation" portType="reserveRoomsPT"/>
</export>
</implement>
</activity>
<flowSink name="TravelBookingSink"><input
message="TravelBookingResponse" name="TravelBookingSink"/>
</flowSink>
```

```
<controlLink name="T1" source="orderAirTicketActivity"
target="findBestHotelActivity"/>
<controlLink name="T2" source="orderAirTicketActivity"
target="rentACarActivity"/>
<controlLink name="T3" source="findBestHotelActivity"
target="reserveRoomsActivity"/>
<dataLink name="dataLink0" source="TravelBookingSource"
target="rentACarActivity">
<map sourceMessage="TravelBookingSource" sourcePart="cust1"
targetMessage="InrentACar" targetPart="requesterData"/>
<map sourceMessage="TravelBookingSource" sourcePart="Duration"
targetMessage="InrentACar" targetPart="Duration"/>
<map sourceMessage="TravelBookingSource"
sourcePart="PickUpWhen_DateAndTimeInfo" targetMessage="InrentACar"
targetPart="pickUpTime"/>
<map sourceMessage="TravelBookingSource" sourcePart="PickUpCity_City"
targetMessage="InrentACar" targetPart="city"/>
</dataLink>
<dataLink name="dataLink1" source="TravelBookingSource"
target="findBestHotelActivity">
<map sourceMessage="TravelBookingSource" sourcePart="NumOfRooms"
targetMessage="InfindBestHotel" targetPart="roomQuantity1"/>
<map sourceMessage="TravelBookingSource" sourcePart="NumOfNights"
targetMessage="InfindBestHotel" targetPart="nightQuantity1"/>
<map sourceMessage="TravelBookingSource" sourcePart="RoomType"
targetMessage="InfindBestHotel" targetPart="roomType1"/>
<map sourceMessage="TravelBookingSource" sourcePart="place"
targetMessage="InfindBestHotel" targetPart="place"/>
</dataLink>
<dataLink name="dataLink2" source="TravelBookingSource"
target="orderAirTicketActivity">
<map sourceMessage="TravelBookingSource" sourcePart="cust1"
targetMessage="InorderAirTicket" targetPart="cust1"/>
<map sourceMessage="TravelBookingSource" sourcePart="departure1"
targetMessage="InorderAirTicket" targetPart="departure1"/>
<map sourceMessage="TravelBookingSource" sourcePart="destination1"
targetMessage="InorderAirTicket" targetPart="destination1"/>
<map sourceMessage="TravelBookingSource" sourcePart="noOfPassengers1"
targetMessage="InorderAirTicket" targetPart="noOfPassengers1"/>
<map sourceMessage="TravelBookingSource" sourcePart="depTime1"
targetMessage="InorderAirTicket" targetPart="depTime1"/>
</dataLink>
<dataLink name="dataLink3" source="TravelBookingSource"
target="reserveRoomsActivity">
<map sourceMessage="TravelBookingSource" sourcePart="cust1"
targetMessage="InreserveRooms" targetPart="customerInformation"/>
<map sourceMessage="TravelBookingSource" sourcePart="NumOfRooms"
targetMessage="InreserveRooms" targetPart="roomQuantity2"/>
<map sourceMessage="TravelBookingSource" sourcePart="NumOfNights"
targetMessage="InreserveRooms" targetPart="nightQuantity2"/>
<map sourceMessage="TravelBookingSource" sourcePart="RoomType"
targetMessage="InreserveRooms" targetPart="roomType2"/>
</dataLink>
<dataLink name="dataLink4" source="findBestHotelActivity"
target="reserveRoomsActivity">
<map sourceMessage="OutfindBestHotel" sourcePart="hotelName1"
targetMessage="InreserveRooms" targetPart="hotelName2"/>
```

```
</dataLink>
<dataLink name="dataLink5" source="orderAirTicketActivity"
target="findBestHotelActivity">
<map sourceMessage="OutorderAirTicket" sourcePart="arrivalDate"
targetMessage="InfindBestHotel" targetPart="requestDate"/>
</dataLink>
<dataLink name="dataLink6" source="orderAirTicketActivity"
target="reserveRoomsActivity">
<map sourceMessage="OutorderAirTicket" sourcePart="arrivalDate"
targetMessage="InreserveRooms" targetPart="date1"/>
</dataLink>
<dataLink name="dataLink7" source="rentACarActivity"
target="TravelBookingSink">
<map sourceMessage="TravelBookingSink"
sourcePart="PickUpAddress_Address" targetMessage="OutrentACar"
targetPart="pickUpAddr"/>
<map sourceMessage="TravelBookingSink"
sourcePart="ReturnWhen_DateAndTimeInfo" targetMessage="OutrentACar"
targetPart="returnTime"/>
<map sourceMessage="TravelBookingSink" sourcePart="CarResResult"
targetMessage="OutrentACar" targetPart="CarResResult"/>
</dataLink>
<dataLink name="dataLink8" source="findBestHotelActivity"
target="TravelBookingSink">
<map sourceMessage="TravelBookingSink"
sourcePart="HotelAddress_Address" targetMessage="OutfindBestHotel"
targetPart="hotelAddress"/>
</dataLink>
<dataLink name="dataLink9" source="orderAirTicketActivity"
target="TravelBookingSink">
<map sourceMessage="TravelBookingSink" sourcePart="flightSchedule"
targetMessage="OutorderAirTicket" targetPart="flightSchedule"/><map
sourceMessage="TravelBookingSink" sourcePart="flightReservationNo"
targetMessage="OutorderAirTicket" targetPart="flightReservationNo"/>
</dataLink>
<dataLink name="dataLink10" source="reserveRoomsActivity"
target="TravelBookingSink">
<map sourceMessage="TravelBookingSink" sourcePart="HotelReservationNo"
targetMessage="OutreserveRooms" targetPart="hotelReservationNo"/>
</dataLink>
</flowModel>
<globalModel name="ETravelBooking" serviceProviderType="TravelBooking">
<serviceProvider name="rentACarService" type="rentACarType"><locator
service="http://www.cise.ufl.edu/~lnarasim/ExpressCarRental.wsdl"
type="static"/>
</serviceProvider>
<serviceProvider name="findBestHotelService" type="findBestHotelType">
<locator
service="http://www.cise.ufl.edu/~lnarasim/PerfectHotelMatching.wsdl"
type="static"/>
</serviceProvider>
<serviceProvider name="orderAirTicketService"
type="orderAirTicketType">
<locator
service="http://www.cise.ufl.edu/~lnarasim/EAirticketOrder.wsdl"
type="static"/>
</serviceProvider>
```

```
<serviceProvider name="reserveRoomsService" type="reserveRoomsType">
<locator
service="http://www.cise.ufl.edu/~lnarasim/EZHotelBooking.wsdl"
type="static"/>
</serviceProvider>
<plugLink>
<source operation="rentACarOperation" portType="rentACarPT"
serviceProvider="TravelBooking"/>
<target operation="rentACar" portType="ExpressCarRental"
serviceProvider="rentACarService"/>
</plugLink>
<plugLink>
<source operation="findBestHotelOperation" portType="findBestHotelPT"
serviceProvider="TravelBooking"/>
<target operation="findBestHotel" portType="PerfectHotelMatching"
serviceProvider="findBestHotelService"/>
</plugLink>
<plugLink>
<source operation="orderAirTicketOperation" portType="orderAirTicketPT"
serviceProvider="TravelBooking"/>
<target operation="orderAirTicket" portType="EAirticketOrder"
serviceProvider="orderAirTicketService"/>
</plugLink>
<plugLink>
<source operation="reserveRoomsOperation" portType="reserveRoomsPT"
serviceProvider="TravelBooking"/>
<target operation="reserveRooms" portType="EZHotelBooking"
serviceProvider="reserveRoomsService"/>
</plugLink>
</globalModel>
</definitions>

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="travelOrderDefs" targetNameSpace="" xmlns=""
xmlns:xsd="http://www.w3c.org/2002/XMLSchema">
<complexType name="FlightSchedule">
<sequence>
<element name="depTime" nillable="true" type="DateAndTimeInfo"/>
<element name="arrTime" nillable="true" type="DateAndTimeInfo"/>
<element name="flightNo" nillable="true" type="xsd:string"/><element
name="destination" nillable="true" type="xsd:string"/>
</sequence>
</complexType>
<complexType name="CustomerInfo">
<sequence>
<element name="address" nillable="true" type="Address"/>
<element name="customerName" nillable="true" type="xsd:string"/>
<element name="creditCardNumber" type="xsd:int"/>
<element name="contactNumber" nillable="true" type="xsd:string"/>
</sequence>
</complexType>
<complexType name="CarReservationInfo">
<sequence>
<element name="rentalCompanyInfo" nillable="true" type="xsd:string"/>
<element name="reservationNumber" type="xsd:int"/>
<element name="cost" type="xsd:int"/>
</sequence>
```

```
</complexType>
<complexType name="DateAndTimeInfo">
<sequence>
<element name="date" nillable="true" type="xsd:dateTime"/>
<element name="time" nillable="true" type="xsd:dateTime"/>
</sequence>
</complexType>
<complexType name="Address">
<sequence>
<element name="street" nillable="true" type="xsd:string"/>
<element name="city" nillable="true" type="xsd:string"/>
<element name="state" nillable="true" type="xsd:string"/>
<element name="zipCode" type="xsd:int"/>
</sequence>
</complexType>
<message name="TravelOrderRequest">
<part name="cust1" type="CustomerInfo"/>
<part name="Duration" type="xsd:int"/>
<part name="PickUpWhen_DateAndTimeInfo" type="DateAndTimeInfo"/>
<part name="PickUpCity_City" type="xsd:string"/>
<part name="NumOfRooms" type="xsd:int"/>
<part name="NumOfNights" type="xsd:int"/>
<part name="RoomType" type="xsd:string"/>
<part name="place" type="xsd:string"/>
<part name="departure1" type="xsd:string"/>
<part name="destination1" type="xsd:string"/>
<part name="noOfPassengers1" type="xsd:int"/>
<part name="depTime1" type="DateAndTimeInfo"/>
</message>
<message name="rentACarRequest">
<part name="pickUpTime" type="DateAndTimeInfo"/>
<part name="requesterData" type="CustomerInfo"/>
<part name="duration" type="xsd:int"/>
<part name="city" type="xsd:string"/>
</message>
<message name="rentACarResponse">
<part name="pickUpAddr" type="Address"/>
<part name="returnTime" type="DateAndTimeInfo"/>
<part name="carResResult" type="CarReservationInfo"/>
</message>
<message name="findBestHotelRequest">
<part name="roomType1" type="xsd:string"/>
<part name="place" type="xsd:string"/>
<part name="roomQuantity1" type="xsd:int"/>
<part name="nightQuantity1" type="xsd:int"/>
<part name="requestDate" type="xsd:dateTime"/>
</message>
<message name="findBestHotelResponse">
<part name="hotelName1" type="xsd:string"/>
<part name="hotelAddress" type="Address"/>
</message>
<message name="orderAirTicketRequest">
<part name="depTime1" type="DateAndTimeInfo"/>
<part name="noOfPassengers1" type="xsd:int"/>
<part name="destination1" type="xsd:string"/>
<part name="departure1" type="xsd:string"/>
<part name="cust1" type="CustomerInfo"/>
```

```
</message>
<message name="orderAirTicketResponse">
<part name="arrivalDate" type="xsd:dateTime"/>
<part name="flightSchedule" type="FlightSchedule"/>
<part name="flightReservationNo" type="xsd:string"/>
</message>
<message name="reserveRoomsRequest">
<part name="nightQuantity2" type="xsd:int"/>
<part name="roomQuantity2" type="xsd:int"/>
<part name="roomType2" type="xsd:string"/>
<part name="date1" type="xsd:dateTime"/>
<part name="hotelName2" type="xsd:string"/>
<part name="customerInformation" type="CustomerInfo"/>
</message>
<message name="reserveRoomsResponse">
<part name="hotelReservationNo" type="xsd:string"/>
<part name="hotelName3" type="xsd:string"/>
</message>
<message name="TravelOrderResponse">
<part name="PickUpAddress_Address" type="Address"/>
<part name="ReturnWhen_DateAndTimeInfo" type="DateAndTimeInfo"/>
<part name="CarResResult" type="CarReservationInfo"/>
<part name="HotelAddress_Address" type="Address"/>
<part name="flightSchedule" type="FlightSchedule"/>
<part name="flightReservationNo" type="xsd:int"/>
<part name="HotelReservationNo" type="xsd:int"/>
</message>
</definitions>
```

LIST OF REFERENCES

[1] Box D., Ehnebuske D., Kakivaya G., Layman A., Mendelsohn N., Nielsen H., Thatte S, Winer D., "Simple Object Access Protocol (SOAP) 1.1," May 2000, http://www.w3.org/TR/SOAP/, Accessed on 02/23/2003.

[2] Christensen E., Curbera F., Meredith G., Weerawarana S., "Web Services Description Language (WSDL) 1.1," March 2001, http://www.w3.org/TR/wsdl, Accessed on 02/23/2003.

[3] David Booth, Michael Champion, Chris Ferris, Francis McCabe, Eric Newcomer, and David Orchard, "Web Services Architecture (WSCA 1.0)", W3C Working Draft,May 2003, http://www.w3.org/TR/ws-arch/ Accessed on 02/23/2003.

[4] Bellwood T., Clement L., Ehnebuske D., Hatley A., Hondo M., Husband Y., Januszewski K., Lee S., McKee B., Munter J., Riegen C., "UDDI Version 3.0," July 2002, http://www.uddi.org/pubs/uddi-v3.00-published-20020719.htm, Accessed on 02/23/2003.

[5] IBM, "UDDI Business Registry Vesion 2," https://uddi.ibm.com/ubr/registry.html, Accessed on 09/23/2003.

[6] Degwekar S., "Constraint-based Brokering for Publishing and Discovery of Web Services," Master's Thesis, Department of Computer and Information Science and Engineering, University of Florida, 2002.

[7] Leymann F., "Web Services Flow Language (WSFL 1.0)," May 2001, www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, Accessed on: 09/23/2003.

[8] Chikkamagalur R., "A WSFL Engine for the Enactment of Composite Web Services," Master's Thesis, Department of Computer and Information Science and Engineering, University of Florida, 2002.

[9] Jwordnet, URL http://sourceforge.net/projects/jwn/, Accessed on: 07/07/2003

[10] "United Nations Standard Products and Services Code" [UNSPSC], URL http://www.unspsc.org/AdminFolder/Documents/UNSPSC_White_Paper.doc Accessed on 03/07/20003

[11] "Java API for XML Processing", Available from URL: http://java.sun.com/xml/jaxp/ Accessed on: 07/07/2003

[12] Shenoy A., “A Persistent Object Manager for Java Applications”, M.S. thesis, Department of Computer and Information Science and Engineering, University of Florida, 2001.

[13] “The Tomcat 4 Servlet/JSP Container,” Available from URL: http://jakarta.apache.org/tomcat/tomcat-4.1-doc/index.html. Accessed on: 07/07/2003.

[14] “Google web API”, Available from URL: http://www.google.com/apis/, Accessed on 09/09/2003.

[15] “IBM Web services toolkit”, Available from URL: http://www-106.ibm.com/developerworks/, Accessed on: 07/07/2003.

[16] Thatte S., “XLANG Web Services for Business Process Design.” http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, Accessed on: 09/23/2002.

[17] Andrews T., Business Process Execution Language, May 2003, http://www-106.ibm.com/developerworks/library/ws-bpel/, Accessed on: 09/23/2002

BIOGRAPHICAL SKETCH

Lakshmi N. Chakarapani is a native of Chennai, India. He earned his high school certificate from Vellayan Chettiar Higher Secondary School, Chennai. He earned his Bachelor of Engineering in the field of computer science at the University of Madras, Chennai, India.

In August 2001, he joined the University of Florida to pursue a master's degree in computer science and engineering. He has been a teaching assistant and a graduate research assistant during his studies at the University of Florida. His research interests include database systems, information integration, grid computing and Web services.