A WEB SERVICES FLOW LANGUAGE (WSFL) ENGINE FOR THE ENACTMENT OF COMPOSITE WEB SERVICES

By

RAMAN N. CHIKKAMAGALUR

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

2002
I dedicate this thesis to my family
ACKNOWLEDGMENTS

I offer my highest gratitude to Dr. Herman Lam, chairman of my supervisory committee, for providing me with guidance and motivation to complete this thesis. My great appreciation goes to Dr. Stanley Y. W. Su, my supervisory committee member, for constantly providing me with valuable comments and suggestions during my thesis work. I would like to thank Dr. Manuel Enrique Bermudez, my supervisory committee member, for helping me learn good programming principles and for his precious time. I feel very proud for having people of this stature related to my work.

Thanks go to Sharon Grant for making the Database Center such a pleasant place to work. I would like to thank John Bowers and Nisi Caudle for providing me guidance and support during my graduate studies.

My wholehearted gratitude goes to my parents, Nagaraj Chikkamagalur and Sarala Nagaraj, for their unconditional love and continuous encouragement throughout my career. I heartily thank my brother, Ravikiran Chikkamagalur, and sister-in-law, Shubha Gupta, whose love, support and guidance helped me overcome many challenges during my graduate studies.

I thank Althea Liang, Seema Degwekar, Karthik Nagarajan, Lakshmi Chakrapani, Gilliean Lee and Seokwon Yang, who helped me learn many things and also made my stay in the Database Systems R&D Center enjoyable.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>Abstract</td>
<td>ix</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Survey of Enabling Technologies</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Web Services Technology</td>
<td>5</td>
</tr>
<tr>
<td>2.2 WSDL</td>
<td>8</td>
</tr>
<tr>
<td>2.3 SOAP</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Axis Toolkit</td>
<td>10</td>
</tr>
<tr>
<td>2.5 WSFL</td>
<td>11</td>
</tr>
<tr>
<td>2.5.1 Flow Model</td>
<td>12</td>
</tr>
<tr>
<td>2.5.2 Global Model</td>
<td>14</td>
</tr>
<tr>
<td>2.6 Dynaflow</td>
<td>15</td>
</tr>
<tr>
<td>3 A WSFL Engine for an Intelligent Registry</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Automated Web Service Composition in an Intelligent Registry</td>
<td>16</td>
</tr>
<tr>
<td>3.2 WSFL Engine Architecture</td>
<td>20</td>
</tr>
<tr>
<td>4 Runtime Workflow Structures Generator</td>
<td>23</td>
</tr>
<tr>
<td>4.1 A Travel Order example</td>
<td>23</td>
</tr>
<tr>
<td>4.2 Design and Implementation of the Runtime Workflow Structures Generator</td>
<td>27</td>
</tr>
<tr>
<td>4.2.1 Creating Begin Activity Structure</td>
<td>29</td>
</tr>
<tr>
<td>4.2.2 Creating Activity Info Structure</td>
<td>30</td>
</tr>
<tr>
<td>4.2.3 Creating Dataflow Structure</td>
<td>32</td>
</tr>
<tr>
<td>4.2.4 Creating Transition Structure</td>
<td>35</td>
</tr>
<tr>
<td>4.2.5 Creating End Activity Structure</td>
<td>38</td>
</tr>
<tr>
<td>4.2.5: Persisting generated Runtime Workflow Structures</td>
<td>39</td>
</tr>
</tbody>
</table>
5 ACTIVITY CODE GENERATOR AND WF EXECUTOR.........................................................40

5.1 Design and implementation of Activity Code Generator........................................40
  5.1.1 WSFL Parser ..................................................................................................42
  5.1.2 AXIS WSDL2Java .......................................................................................44
  5.1.3 Stub Code Invoker Generator ....................................................................45

5.2 Design and Implementation of the WF Executor....................................................48
  5.2.1 Implementation of Scheduler ....................................................................49
  5.2.2 Implementation of Activity Handler .........................................................51

6 SUMMARY AND CONCLUSION.......................................................................................55

APPENDIX STUB CODES GENERATED BY AXIS TOOL KIT.........................................59

  1 WSDL Document for EAirticketOrder web service .............................................59
  2 Stub Code EAirticketOrder class .......................................................................63
  3 Stub Code EAirticketOrderService ......................................................................63
  4 Stub Code EAirticketOrderServiceLocator class................................................64
  5 Stub Code for EAirticketOrderSoapBindingStub ................................................65

LIST OF REFERENCES ..................................................................................................71

BIOGRAPHICAL SKETCH.............................................................................................73
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Web Services Model</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Web Services stack</td>
<td>7</td>
</tr>
<tr>
<td>2.3 SOAP message structure</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Components involved in Dynamic and automatic web service composition</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Sequence of activities among the components</td>
<td>19</td>
</tr>
<tr>
<td>3.3 WSFL Engine Architecture</td>
<td>21</td>
</tr>
<tr>
<td>4.1 WSFL flow structure of “Travel Order” composite service</td>
<td>25</td>
</tr>
<tr>
<td>4.2 Architecture of Runtime Workflow Structures Generator</td>
<td>27</td>
</tr>
<tr>
<td>4.3 Flow structure of method processCodeGen</td>
<td>28</td>
</tr>
<tr>
<td>4.4 flowSource element for “Complete Travel Order” example</td>
<td>29</td>
</tr>
<tr>
<td>4.5 Begin Activity Structure for Travel Order example</td>
<td>30</td>
</tr>
<tr>
<td>4.6 Activity Element for BookAirTickets</td>
<td>31</td>
</tr>
<tr>
<td>4.7 Activity Info Structure for BookAirTickets activity</td>
<td>32</td>
</tr>
<tr>
<td>4.8 Explicit dataLink mapping between flowSource and activity BookAirTickets</td>
<td>34</td>
</tr>
<tr>
<td>4.9 Dataflow Structure for datalink0 of Travel Order example</td>
<td>34</td>
</tr>
<tr>
<td>4.10 Modified control structure for “Complete Travel Order” example</td>
<td>36</td>
</tr>
<tr>
<td>4.11 Attributes of Transition Structures</td>
<td>37</td>
</tr>
<tr>
<td>4.12 Transition structure for connector EndJoinConn</td>
<td>37</td>
</tr>
<tr>
<td>4.13 flowSink element of “Complete Travel Order” flow model</td>
<td>38</td>
</tr>
<tr>
<td>4.14 End Activity Structure for Travel Order example</td>
<td>39</td>
</tr>
</tbody>
</table>
5.1 Architecture of Activity Code Generator ................................................................. 42
5.2 Binding information for activity BookAirTickets ...................................................... 43
5.3 Code Generated to Invoke the Stub codes for Activity BookAirTickets .................. 48
5-4 Architecture of WF Executor .................................................................................. 49
5.5 SOAP request sent from activity code to EAirticketOrder web service .................. 53
5.6 SOAP response from EAirticketOrder web service to activity code ...................... 54
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 WEB SERVICES FLOW LANGUAGE (WSFL) ENGINE FOR THE ENACTMENT OF COMPOSITE WEB SERVICES

By
Raman N. Chikkamagalur

December 2002

Chair: Dr. Herman Lam
Major Department: Computer and Information Science and Engineering

Web services technology is envisioned to provide application-to-application interactions in such a way as to better support the sharing of application functionality and business processes among organizations that conduct collaborative e-business over the Internet. In order to offer higher value, end-to-end services, it should be possible to compose “composite” web services from “base” web services in a simple and flexible manner. Currently, there is an ongoing research project at the Database Systems Research and Development Center of the University of Florida to develop an Intelligent Registry that aims to produce composite web services in a dynamic and automatic manner. The composite service plan generated by the Intelligent Registry will be in the form of a Web Services Flow Language (WSFL) document.

The focus of this thesis is the design and implementation of a WSFL Engine to execute the composite service plan represented by a WSFL document. Using a “code generation” approach, the WSFL Engine generates (1) runtime workflow structures and
(2) activity codes. The runtime workflow structures determine the control and data flows of a process instance. The activity code contains information to bind the activity operations to the web services. The code generation approach results in an efficient and flexible WSFL Engine. It is efficient because for each invocation of a process model the workflow engine does not have to interpret the WSFL document in order to determine the flow and how to bind to the web services. It takes advantage of the performance of compiled classes. The separation of the flow information in the runtime workflow structures and the binding information in the activity code leads to loose coupling and better flexibility. Flexibility is maintained without sacrificing performance. The WSFL Engine itself is implemented as a web service. The WSDL document for the WSFL Engine could be published and thus the services of the WSFL Engine could be exposed over the Internet.
CHAPTER 1
INTRODUCTION

Web services technology is envisioned to provide application-to-application interactions in such a way as to better support the sharing of application functionality and business processes among organizations that conduct collaborative e-business over the Internet. Web services propose a service-oriented paradigm for computing in which distributed, loosely coupled services collaboratively provide business processes and can be accessed via the Internet by end customers. Web services are self-contained modular business applications that have open, Internet oriented, standards-based interfaces. This standards-based communication allows web services to be accessed by customers, suppliers and trading partners independent of hardware, operating system and even programming environment. The relevant standards for web services include Simple Object Access Protocol (SOAP) [1], Web Services Description Language (WSDL) [2] and Universal Description, Discovery and Integration (UDDI) [3].

Current web services techniques are limited to applications that operate independently. In order to offer higher value, end-to-end services, it should be possible to compose, customize and deploy web services in a very flexible and efficient way. This leads to the need for composing autonomous services to get new services that achieve new functionality. Service composition will considerably reduce development time, effort and cost for new applications. Recently, specifications like the Web Services Flow Language (WSFL) [4] from IBM have been proposed to describe composition of services in the form of a workflow using XML. A WSFL specification describes how different
web services should be invoked in terms of ordering and parallelism. Microsoft has made a similar proposal called XLANG [5] for web services composition.

Most of the current research work on web services composition relies on the composition tools to define a business model [6]. In an ongoing research project at the Database Systems Research and Development Center of the University of Florida, a system is being developed for dynamic and automatic web services composition. The purpose is to build an Intelligent Registry capable of providing dynamic and automatic web services composition, based on user requests and taking into consideration the constraint information provided by the service providers and service requestor.

The Intelligent Registry provides a Service Registration function for receiving and processing web services registration from a service provider. The service registration includes the conventional registration information, such as business name, business description, service name, service description, access point information, etc. It also includes the provider constraints on this service. The Intelligent Registry also provides a Query Processor to which a service requester can submit a query. The query may be a request for a simple web service or it may result in the invocation of a set of web services. In the later case, the Query Processor will invoke the Service Composer component to generate a composite service plan dynamically. The composite service plan is an execution sequence 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 among these services. Based on the composite service plan, a corresponding WSFL document is generated by a WSFL Generator. After the WSFL document is generated by the Intelligent Registry, it will invoke a WSFL Engine to execute it.
The focus of this thesis is the design and implementation of a WSFL Engine to support composite web services invocation as dictated by a WSFL document. Using a “code generation” approach, we generate (1) runtime workflow structures and (2) activity codes, based on the input given in the form of a WSFL document. The WSFL document is parsed using standard XML parsers to retrieve information related to activities, dataflow and controlflow.

The runtime workflow structures and the activity codes are used by the WSFL Engine to schedule and execute a workflow instance. The generated activity code is in the form of Java code. The WSFL Engine, when scheduling the activities for a process instance, loads the Java activity classes from a runtime repository and executes the code directly to perform specified tasks. The outputs of web services are mapped to input of corresponding web services as dictated by the WSFL document.

This code generation approach results in an efficient and flexible WSFL Engine. It is efficient because for each invocation of a process model, the workflow engine does not have to interpret the WSFL document in order to determine the flow and how to bind to the web services. It takes advantage of the performance of compiled classes. The separation of flow information in the runtime workflow structures and the binding information in the activity code leads to loose coupling and better flexibility. Changes to an activity (e.g., change of a service operation) can be made by modifying the WSFL global model. The code for the modified activity is re-generated and re-loaded using Java’s class reloading capability. In this manner, any change made to an activity specification will be reflected in the execution of the activity as long as the re-generated code is placed in the runtime repository before the execution of the activity. Thus,
flexibility is maintained without sacrificing performance. The WSFL Engine itself is implemented as a web service. The WSDL document for the WSFL Engine could be published and thus the services of the WSFL Engine could be exposed over the Internet.

The organization of this thesis is as follows. Chapter 2 presents a survey of the enabling technologies. Chapter 3 describes the overview of the Intelligent Registry and the architecture of the WSFL Engine. Chapter 4 provides the details of the design and implementation of the Runtime Workflow Structures Generator. Chapter 5 describes the details of design and implementation of the Activity Code Generator and the Workflow Execution Module. Chapter 6 provides a summary and conclusion of the thesis.
CHAPTER 2
SURVEY OF ENABLING TECHNOLOGIES

This chapter presents a survey of the standards, tools and software that have been used for the development of the WSFL Engine.

2.1 Web Services Technology

Web services technology could be viewed as a platform for distributed computing over the Web. The World Wide Web Consortium (W3C) defines web services as a network accessible interface to application functionality, built using standard Internet technologies. These web services are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. After a web service is deployed, other applications (and other web services) can discover and invoke the deployed service. The goal of web services includes universal operability, widespread adoption and ubiquitous accessibility of deployed services. It acts as an abstraction layer separating the platform and programming language specific details of how the application code is actually invoked. This standardized layer means that any language that supports web services can access the application’s functionality. This interoperability among applications written in different programming languages is one of the key benefits for implementing web services.

At the core of the web services model is the notion of a “Service,” which is defined as a collection of operations that carry out some type of task. Within the context of web services, a service can be described, discovered and invoked using standard XML technologies such as WSDL, SOAP and UDDI.
Web services are implemented and published by Service Providers. They are discovered and invoked by Service Requesters. Information about a web service may be kept within a Service Registry. Figure 2.1 provides an illustration as to how these three roles interact with each other.

Figure 2.1 Web Services Model

The three fundamental operations shown in Figure 2.1 can be described in the following ways:

- **Publish**--performed by the Service Provider to advertise the existence and capabilities of a service.
- **Find**--performed by the Service Requester to locate a service that meets a particular need.
- **Bind**--performed by the Service Requester to invoke the service being provided by the Service Provider.

The web services architecture is implemented through four types of technologies organized into layers that build upon one another as shown in Figure 2.2. The packaging, description and discovery layers in the web services stack are the layers essential for providing just-in-time integration capability and the necessary platform-neutral programming model.
The *discovery* layer provides the mechanism for clients to fetch the description of providers. One of the most widely recognized discovery mechanisms available is the Universal Description, Discovery and Integration (UDDI) project. IBM and Microsoft have jointly proposed an alternative to UDDI, called the Web Services Inspection (WS-Inspection).

When a web service is implemented, it must make decisions on every level as to which network, transport, and packaging protocols it will support. A *description* of that service represents those decisions in such a way that the service consumer can contact and use the service. The Web Services Description Language (WSDL) is the standard for providing those descriptions. Other less popular approaches include the use of the W3C’s Resource Description Framework (RDF)[7] and the DARPA Agent Markup Language (DAML) [8], both of which provide a much richer (but far more complex) capability of describing web services than WSDL.

For application data to be moved around the network by the transport layer, it must be *packaged* in a format that all parties can understand (other terms used for this process are “serialization” and “marshalling”). XML is the basis for most of the present web services packaging formats because it can be used to represent the meaning of the data being transferred, and because XML parsers are now ubiquitous. SOAP is a very common packaging format built on XML that has been proposed by W3C. The *transport layer* maps to the TCP and IP protocols of HTTP protocol.
The WSFL Engine makes use of WSDL documents to get the location and binding information for a web service. Section 2.2 gives a brief introduction of WSDL. The WSFL Engine sends SOAP requests and receives SOAP response from a web service. Section 2.3 gives a brief description of SOAP.

2.2 WSDL

WSDL stands for Web Services Description Language. WSDL is a proposal by Ariba, IBM and Microsoft for describing web services for the W3C XML Activity on XML Protocols. WSDL is a specification language defining how to describe web services in a common XML grammar. WSDL describes four critical pieces of data:

- Interface information describing all publicly available functions
- Data type information for all the message requests and message responses
- Binding information about the transport protocol to be used
- End-point address information for locating the specified service

Conceptually, WSDL represents a contract between the service requester and service provider. The main advantage of using WSDL is it is platform and language independent and is used primarily to describe SOAP services. WSDL represents a cornerstone of web services architecture, because it provides a common language for describing services and a platform for automatically integrating those services.

With the help of WSDL, a client can locate a web service and invoke any of its publicly available operations. Given a WSDL file it is possible for a user to write a client code to invoke the corresponding web service. But it is more convenient to automatically generate client stubs by using the WSDL document. Several toolkits like Microsoft SOAP Toolkit [9], SOAP::Lite[10], AXIS[11] provide these features. In our thesis, we make use of the Apache AXIS toolkit to generate client stubs.
2.3 SOAP

SOAP stands for “Simple Object Request Protocol”. SOAP is a simple XML based protocol to allow applications exchange information using HTTP. It is a communication protocol between applications, which defines formats for the messages that are exchanged between applications. SOAP is platform independent and language independent because it is based on XML. SOAP provides a way to communicate between applications running on different operating systems, with different technologies and programming languages. In the SOAP protocol, everything that goes across the wire is expressed in terms of HTTP or SMTP headers, MIME encoding, and special XML grammar as defined by SOAP specification [1] for encoding application data.

A SOAP message is an ordinary XML document. Its structure is depicted in Figure 2.3.

A SOAP message contains the following elements:

- A SOAP envelope, that defines the content of the message
- A SOAP header (optional), that contains header information
- A SOAP body, that contains call and response information

The **SOAP envelope** declaration is the outermost XML tag that delineates the boundaries of the SOAP document. The **SOAP header and body** elements are syntactically similar. SOAP 1.1 has no conventions for what is supposed to be in the header. It is simply a place to put directives to the SOAP processor that receives the message. The sending and receiving applications need to agree on which elements go there and what they mean. Higher- level protocols built on top of SOAP, such as ebXML, Message Service (MS), have formalized the use of the SOAP header by defining specific elements such as a `<MessageHeader>`, which contains such specific things as `<From>`, `<To>`, and `<MessageId>`. The SOAP body is intended for the actual data, or message payload, to be consumed and processed by the ultimate receiver. The `<Body>` element is
reserved purely for the method call and its parameters, and the <Header> is used for things targeted at the underlying infrastructure, such as a transaction ID, where transaction ID is not a part of the method signature. It is intended for the SOAP processor that receives the message, which could very well be a J2EE server with a transaction manager.

![SOAP message structure](image)

Figure 2.3 SOAP message structure

### 2.4 Axis Toolkit

AXIS [11] is a web services toolkit developed by Apache. This is a SOAP Engine that provides a framework for constructing SOAP processors such as clients, servers and gateways. The current version of Axis is written in Java. Axis provides features so that it acts as a server that plugs into a servlet engine such as Tomcat. It provides APIs that generate WSDL document from a Java class file and provides APIs to generate client stubs and server skeletons from a WSDL document. It also provides a tool for monitoring TCP/IP packets to view the structure of SOAP messages that are exchanged between applications.

AXIS has features that allow deployment of web services in a flexible way by making use of a Web Services Deployment Descriptor (WSDD). The AXIS API,
org.apache.axis.client.AdminClient is used to deploy and undeploy a web service. A deployment descriptor contains details that the AXIS engine should know to deploy a Java class file as a web service. It contains details regarding the Java class name for a given service, mapping QNAME information to Java classes for the purpose of de-serialization and serialization [12].

AXIS supports WSDL in three ways:

- After a web service is deployed in Axis, users may then access the web service's URL with a standard web browser and by appending "?wsdl" to the end of the URL. They will obtain an automatically generated WSDL document, which describes the deployed web service.

- AXIS provides a "WSDL2Java" tool, which will generate Java proxies and skeletons for services with WSDL descriptions.

- AXIS provides a "Java2WSDL" tool, which will build WSDL from Java classes.


2.5 WSFL

As explained earlier, the goal of web services is to enable seamless application integration over the Internet regardless of programming languages or operating environments. The goal of 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 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)[4]. WSFL is a tool to model a workflow using an XML syntax that can be read by both humans and machines.
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.

2.5.1 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 metamodel 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.

2.5.2 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.

### 2.6 Dynaflow

Dynaflow [13], a research project that was successfully carried out at the Database Systems Research and Development Center at University of Florida, implemented a dynamic workflow engine. The workflow engine is dynamic because it facilitated the service composer to modify the structure and properties of the workflow model even after it was constructed. The implementation of Dynaflow’s workflow engine followed a code generation approach. The code generator of workflow engine interpreted a workflow expressed in terms of the Workflow Process Definition Language (WPDL) [14] from Workflow Management Coalition and generated runtime structures and activity codes. The workflow scheduler would later use these generated runtime workflow structures and activity codes to enact a workflow instance.

Since, the concept of building and executing workflow and composite service are very similar, we have followed the approach taken in Dynaflow’s workflow engine. The WSFL Engine described in this thesis follows the same code generation methodology to generate runtime structures and activity code. The WSFL Engine parses WSFL document and populates runtime workflow structures that were used by Dynaflow. The WF Executor of the WSFL Engine is implemented using Dynaflow workflow scheduler by modifying the activity handler and making minor modifications for the scheduler code.
CHAPTER 3
A WSFL ENGINE FOR AN INTELLIGENT REGISTRY

Most of the current research work on service composition relies on the composition tools to define business model [6]. As an ongoing research project at the Database Systems Research and Development Center of the University of Florida, a system is being developed for dynamic and automatic web service composition. The purpose is to build an Intelligent Registry capable of providing dynamic and automatic web service composition, based on the end users request and taking into consideration the constraint information provided by the service providers and service requestor.

This chapter provides a brief description of the Intelligent Registry and describes the architecture of a WSFL Engine, which forms a key component of this system.

3.1 Automated Web Service Composition in an Intelligent Registry

Figure 3.1 gives an architecture diagram for the Intelligent Registry and shows how a WSFL Engine interacts with it. The Intelligent Registry and WSFL Engine are deployed as web services and therefore they will be accessible to service providers and service requestors over the Internet. The Intelligent Registry consists of the following components:

Service registration. This component is responsible for receiving and processing web service registration from a service provider. The service registration make use of a UDDI registry to register conventional registration information, such as business name, business description, business classification, service name, service description, access point information and the category to which the service belongs. It also registers the
constraints defined by the service provider on this service such as constraints on service attributes, constraints on operation attributes and constraints on input/output parameter. The WSDL document used for the service registration has been extended to capture the constraint information in XML format [15].

**UDDI registry.** The *UDDI Registry* component is a conventional UDDI registry such as IBM UDDI registry [16] or Microsoft UDDI registry [17]. This component is capable of all brokering activities, such as publish, save and find a web service as specified in UDDI specification [3].

**Query processor.** A service requester can submit a query to the *Intelligent Registry*. The query is constructed using request description language that follows XML schema. The query may be a request for a simple web service or it may result in the invocation of a set of web services. In the latter case, the *Query Processor* will invoke the *Service Composer* component.

**Service composer.** Based on the user request and the constraints on the request the *Service Composer* component generates a *composite service plan* dynamically. The *composite service plan* is a directed acyclic graph that represents a set of web services connected in a planned order to serve the request of a *Service Requestor*. From the execution viewpoint, a *composite service plan* is an execution sequence 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. Interaction between the *Service Composer* component and *Service Requestor* may be necessary to get an optimal *composite service plan*. 
**WSFL generator.** The *WSFL Generator* generates a WSFL document by making use of the *composite service plan*. The generated WSFL document will capture the sequence of execution of the web services and the dataflow mapping between the web services in XML format. The WSFL document will also contain the endpoint information of individual web services that form part of the composite service plan. The endpoint service information is represented as an URL to the WSDL implementation document of the web service.

![Diagram](image)

**Figure 3.1** Components involved in Dynamic and automatic web service composition

After the WSFL document is generated by the *Intelligent Registry*, it will invoke the *WSFL Engine* to generate runtime workflow structures and activity codes by providing the *WSFL Engine* an URL to the WSFL document. After the runtime workflow structures and activity codes for a WSFL document are successfully generated, the *Intelligent Registry* will provide the endpoint information of the *WSFL Engine* to the
Service Requestor along with the process model name. Since the WSFL Engine is deployed as a web service, the endpoint information of WSFL Engine could be retrieved using its WSDL document.

Figure 3.2 Sequence of activities among the components

The service requestor can then invoke the WSFL Engine to instantiate and execute the instance of service composition that was created on its request. The WSFL Engine will execute independent web services in a choreographed manner as specified by the runtime workflow structures and returns the end result to the Service Requestor. Figure 3.2 summarizes the sequence of activities among the components.
3.2 WSFL Engine Architecture

The architecture of the WSFL Engine is show in Figure 3.3. It is deployed as a web service, which has the advantage of ubiquitous accessibility and loose coupling with the Intelligent Registry. In describing the WSFL Engine architecture, we will use the following definitions. A process model is used to model a business process and consists of a network of activities and their control/data flow of these activities. A process model will be instantiated as a process instance and executed by the WSFL Engine. WSFL Engine exposes four interfaces to a WSFL Engine client:

- **GenerateWFStructures**: This interface will be invoked by the Intelligent Registry to generate runtime workflow structures which correspond to the composite service plan represented by WSFL flow model document.

- **GenerateActivityCode**: This interface is invoked by the Intelligent Registry to generate code for the activities in a process model.

- **DeleteProcessModel**: The Intelligent Registry invokes this interface to delete a process model when it is no longer in use. After the process model is deleted, a process instance for this model cannot be instantiated.

- **InvokeWFInstance**: On invocation of this interface, a process instance for a specified process model is instantiated, scheduled and executed. This interface will return the final output to the Service Requestor after all the activities in the process model are processed.

As shown in Figure 3.3, the WSFL Engine consists of three main components.

- Runtime Workflow Structures Generator
- Activity Code Generator
- WF Executor

When the GenerateWFStructures API is called, it invokes the Runtime Structures Generator to generate runtime workflow structures according to the WSFL flow model document. The generated runtime workflow structures will be written to a Java serialization file and stored in a runtime repository that will be later used by the WF
Executor to execute the process instance. The design and implementation of Runtime Structures Generator are presented in chapter 4.

Figure 3.3 WSFL Engine Architecture

When the GenerateActivityCode API is called, it invokes Activity Code Generator to generate Java classes for the activities defined in the WSFL document. The generated activity code consists of a set of client stubs and code to invoke these stubs. The Activity Code Generator retrieves the WSDL URL from the WSFL global model document and uses the AXIS toolkit’s WSDLToJava API to generate client stubs. It then generates Java classes that invoke the generated client stubs to bind to the remote web services. The design and implementation of Activity Code Generation is presented in Chapter 5.

When the InvokeWFInstance API is called, it invokes WF Executor to instantiate and execute a process instance. The WF Executor loads the runtime workflow structures
and activity codes for this process model. The activity code is scheduled by the WF Executor as per the sequence dictated by the runtime workflow structures. It also does the input-output parameter mapping between activities as specified by runtime workflow structures. WF Executor manages the execution of a service composition until completion by assigning activities to each specified service provider at the specified time. The design and implementation of WF Executor is also presented in Chapter 5.
CHAPTER 4
RUNTIME WORKFLOW STRUCTURES GENERATOR

As shown in the WSFL Engine architecture in Figure 3.3, the Runtime Workflow Structures Generator generates the runtime workflow structures used by the WF Executor to schedule and execute a process instance. These structures determine the control and data flows of a process instance. Changes can be made to the control and data flows of a process instance by modifying the corresponding WSFL document, which can be used to generate new workflow structures. However, changes made to runtime structures will not affect the generated activity code generated by the Activity Code Generator, which contain binding information to the web services. This leads to better flexibility and loose coupling between flow structure generation and web service binding generation. The design and implementation of the Runtime Workflow Structures Generator will be explained using an example. The flow structure for the example will be presented in section 4.1. Section 4.2 gives the design of Runtime Workflow Structures Generator and implementation details of each of its modules.

4.1 A Travel Order example

In this example, we assume that a traveler is looking for a web service that can address his travel needs that include booking air tickets, reserving a hotel room for his stay and making a car rental reservation. The traveler makes a request to the Intelligent Registry (see section 3.1 and Figure 3.1), to provide this composite web service. First, based on user request and the associated constraints and also the constraints specified by the service providers, the Service Composer component of the Intelligent Registry will
generate a composite service plan that caters to the needs of the traveler. The composite service plan is inputted to the \textit{WSFL Generator}. The \textit{WSFL Generator} component of \textit{Intelligent Registry} will generate a WSFL document specifying the activities, data and control flows of the composite web service.

Figure 4.1 gives a pictorial description on how the flow model for the example \textit{Travel Order} composite service is captured in the WSFL document. In this figure, the dotted lines represent \textit{dataLink}, mapping the dataflow between activities and the dark lines represent \textit{controlLink}. The shaded boxes represent \textit{activities} that actually contain service operations that need to be invoked.

A \textit{flowSource} element of WSFL flow model contains information regarding the list of input parameters required to initiate the workflow. The details of input parameters for the \textit{flowSource} are as follows:

- Customer Information: This parameter contains details regarding the name, address and credit card information of the traveler.
- Ticket Booking Information: This parameter contains details regarding traveler’s destination and arrival place, number of passengers and travel date and time.
- Hotel Booking Information: This parameter contains details regarding traveler’s choice of room type, number of rooms he wants to reserve and the number of days of stay.
- Car Rental Information: This parameter contains details regarding how many days the traveler needs the car, pick up address and pick up date.

An \textit{Activity} element contains information regarding a web service operation that will be used to bind to a particular service provider. In this example, there are four activities:

- BookAirTickets.
- RentCar.
- DoHotelMatching.
- ReserveHotel.
The functions of these activities are self-explanatory.

Figure 4.1 WSFL flow structure of “Travel Order” composite service

A `dataLink` element maps parameters between output of one activity to input of another activity. 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 `DoHotelMatching` activity.
- **datalink2** maps `DateLocationInfo`, which is one of the output of the activity `BookAirTickets`, to the activity `DoHotelMatching`.

- **datalink3** maps `CustomerInfo` and `HotelBookingInfo` from `flowSource` to `ReserveHotel`.

- **datalink4** maps `HotelInformation` from the `DoHotelMatching` activity to the `ReserveHotel` activity.

- **datalink5** maps `CustomerInfo` and `CarRentalInfo` parameters from `flowSource` to the `RentCar` activity.

- **datalink6** maps output parameter, `Itinerary` of the `BookAirTickets` activity to `flowSink`.

- **datalink7** maps output parameter, `HotelReservationInfo` of the `ReserveHotel` activity to `flowSink`.

- **datalink8** maps output parameter, `CarRentalReservationInfo` of the `RentCar` 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 T1** specifies that activity `DoHotelMatching` should be scheduled only after activity `BookAirTickets` is completed.

- **controlLink T2** specifies that activity `RentCar` should be scheduled only after activity `BookAirTickets` is completed

- **controlLink T3** specifies that activity `ReserveHotel` should be scheduled only after activity `DoHotelMatching` is completed.

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

The following outputs are generated for this example process model:

- **Itinerary**: This output is generated from the `BookAirTickets` activity. This parameter has information like start and arrival date, start and arrival time, ticket number, airline company name and cost of the ticket.

- **HotelReservationInfo**: This output is generated by the `ReserveHotel` activity. This parameter has information regarding hotel address, confirmation number and billed amount.
- **CarReservationInfo**: This output is generated by the *RentCar* activity. This parameter has information regarding car rental company, reservation number and cost for the renting the car.

### 4.2 Design and Implementation of the Runtime Workflow Structures Generator

The main components of the *Runtime Workflow Structures Generator* and their interactions are shown in Figure 4.2.

**Figure 4.2 Architecture of Runtime Workflow Structures Generator**

*Runtime Workflow Structures* generation is initiated by invoking the *WSFL Parser*. This component takes the URL of a WSFL flow model document as its input. The *WSFL Parser* is implemented using the Apache Xerces XML parser that creates a Document Object Model (DOM) tree. The *WSFL Parser* parses the WSFL flow model document and retrieves the *flowSource* element, *Activity* elements, *dataLink* elements, *controlLink* elements and *flowSink* element. These elements are used as input to appropriate workflow structure generators to generate the corresponding workflow structures. Figure 4.3 gives the sequence of the workflow structure generation process. Using standard Java
Serialization APIs, the generated workflow structures are serialized and stored in a runtime repository. These persisted runtime workflow structures are later used by \textit{WF Executor} to enact a process instance.

There are basically five types of runtime structures generated. They are:

- Begin Activity Structure.
- Dataflow Structure.
- Activity Info Structure.
- Transition Structure.
- End Activity Structure.
The following sections will explain how these structures are generated.

4.2.1 Creating Begin Activity Structure

The Begin Activity structure is intended to contain input parameters of a process model that are specified in the flowSource of a WSFL document. The Begin Activity structure will be the first entity structure that will be loaded by the WF Executor. A Begin Activity structure is constructed using the flowSource element present in the WSFL document. Generation of the Begin Activity structure is done by the method `createBeginActivity` of the Runtime Workflow Structures Generator.

The method `createBeginActivity` will parse the flowSource element to retrieve relevant information to populate the Begin Activity runtime structure. The flowSource element for “Travel Order” example is depicted in Figure 4.4. The relevant information includes input parameter names and input parameter types. This information is stored as the Parameter List member variable of the Begin Activity structure. Figure 4.4 also gives the message definitions to which the flowSource element refers.

```xml
<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 4.4 flowSource element for “Complete Travel Order” example
To illustrate, Figure 4.5 shows the *Begin Activity* structure constructed using the *flowSource* element of “Travel Order” example. The *flowSource* element is parsed to retrieve the *output* message name. In this example the output message name is *TravelOrderRequest*. This message name is used to retrieve the parts name and parts type information from the *message* element. In this example we retrieve *customerData*, *ticketData*, *hotelData* and *carData* as parts name. These values are used to populate the *Parameter Name* variable of *Parameter List* hashTable. We retrieve *CustomerInfo*, *TicketInfo*, *HotelBookingInfo* and *CarBookingInfo* as parts type. These values are used to populate the *Parameter Type* variable of *Parameter List* hashTable. The *Parameter Value* part of *Parameter List* will be populated when the process instance is enacted.

**4.2.2 Creating Activity Info Structure**

The *Activity Info* structures are constructed using the *activity* elements present in a WSFL flow model document. This is done in the method *createActivityStructures*. As an example, the *activity* element for *BookAirTickets* activity and its corresponding input and output messages are shown in Figure 4.6.
Figure 4.6 Activity Element for BookAirTickets

The name of the activity, input parameters of activity and output parameters of the activity are parsed and retrieved from activity element and stored in the corresponding Activity Info structure. The Activity Info structure for BookAirTickets activity of “Travel Order” example is depicted in Figure 4.7. To obtain the input and output parameter information, the activity element is parsed to retrieve input and output message names. Input and output message names are used to retrieve part name and part type for the parts of corresponding messages. The parts name and parts type information of input message are stored in input parameter list of Activity Info structure. The parts name and parts type information of output message is stored in output parameter list of Activity Info structure.
Figure 4.7 Activity Info Structure for BookAirTickets activity

4.2.3 Creating Dataflow Structure

The generated dataflow structures are intended for the WF Executor to map data outputted by entities that have successfully completed, to entities that need these data to proceed. An entity can be a Begin Activity, Activitiy or End Activity. Dataflow structures inform the WF Executor where the data comes from, and where it should go by providing mapping information between parameters from different entities. Dataflow structures are generated by the method createDataflows in the Runtime Workflow Structures Generator, using the dataLink elements specified in WSFL flow model.

In a WSFL flow model we can represent two kinds of dataflow mapping. The first one is implicit dataflow mapping and the second one is explicit dataflow mapping. In implicit dataflow mapping, the dataLink element only specifies the source and target message names. In this case, there will be a one-to-one mapping of parts contained in the source and target messages. This functionality is handled by the method, doImplicitDataflowMapping.

In explicit dataflow mapping, the mapping information between parts is specified using the map element in the WSFL document. Explicit dataflow mapping is performed by the method doExplicitDataflowMapping. This method will parse the dataLink
element and retrieves the source entity and destination entity names. It also retrieves source and target message names. Using the entity names and message names, it retrieves the message types of corresponding messages. The message type name is then used to retrieve the part’s name and part’s type from the WSFL definition element. By following this procedure we can find that, the type name for part cust1 (see Figure 4.8) is CustomerInfo. The Parameter Mapping List of dataflow structure is populated using the map element information present in the WSFL document. When the dataflow mapping is processed, care has been taken to ensure that the source and target parts are of the same type. If the source and target part types are different, an exception will be thrown.

To illustrate, Figure 4.8 depicts how the dataLink element datalink0 maps parameters from the Begin Activity to the activity BookAirTickets. The first map element specifies that input parameter cust1 of the BookAirTickets activity is mapped from the parameter customerInfo of flowSource. The second map element specifies that the input parameter travelData of the BookAirTickets activity is mapped from the parameter ticketData of flowSource. Figure 4.9 presents the resultant dataflow structure for datalink0.
<dataLink name = "dataLink0"
   source = "travelflowSource"
   target = "BookAirTickets">
   <map sourceMessage= "TravelOrderflowSource"
       targetMessage = "InBookAirTickets"
       sourcePart = "customerData"
       targetPart = "cust1"/>
   <map sourceMessage= "TravelOrderflowSource"
       targetMessage = "InBookAirTickets"
       sourcePart = "ticketData"
       targetPart = "travelData"/>
</dataLink>

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

<message name = "BookAirTicketsRequest">
   <part name = "cust1" type = "CustomerInfo"/>
   <part name = "travelData" type = "TravelInfo"/>
</message>

Figure 4.8 Explicit dataLink mapping between flowSource and activity BookAirTickets

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Variable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Model Name</td>
<td>Travel Order</td>
</tr>
<tr>
<td>DataLink Name</td>
<td>Datalink0</td>
</tr>
<tr>
<td>Source Entity Name</td>
<td>Begin Activity</td>
</tr>
<tr>
<td>Target Entity Name</td>
<td>BookAir Tickets</td>
</tr>
<tr>
<td>Parameter Mapping List</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Parameter</th>
<th>Target Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>customerData</td>
<td>Cust1</td>
</tr>
<tr>
<td>ticketData</td>
<td>TravelData</td>
</tr>
</tbody>
</table>

Figure 4.9 Dataflow Structure for datalink0 of Travel Order example
4.2.4 Creating Transition Structure

Transition structures capture control dependencies among entities of a process model. A control structure is generated for each of the following entities of a process model: Begin Activity, Activity, Connector, and End Activity. During the execution of a workflow instance, the WF Executor determines whether the next entity can be scheduled according to its corresponding Transition structure.

The generated runtime workflow structures are based on Dynaflow’s workflow structure, which was explained in section 2.2.5. These structures make use of entities called Connectors to handle join and split transitions. But there is no such concept in WSFL specifications. To convert WSFL a flow model to Dynaflow’s workflow structures, it is necessary to generate these Connector entities.

Connector entities are generated based on the controlLink information present in a WSFL document. If an activity name is present as the target activity in more than one controlLink definition in the WSFL document, then we have a join connector. A name will be synthesized for the join connector based on the target join activity name. If an activity name is present as the source activity in more than one controlLink definition in the WSFL document, then we will have a split connector. A name will be synthesized for the split connector based on the source activity name. The generation of transition structures is handled by the method createTransitions. Transitions for join connectors are handled in the method generateTransForJoin. Transitions for split connector are handled by the method generateTransForSplit. If the transition is of a simple type, it is handled by the method generateTransForSimpleCL.

The WSFL document does not specify transitions from the flowSource to a start activity and last activity to flowSink. These transitions will have to be generated, as they
are required for Dynaflow’s workflow structures. Figure 4.10 shows the modified control structure corresponding to the WSFL flow model depicted in Figure 4.1.

The generation of transition structures is much more involved than generation of other entity structures. First, a list of control links specified in a WSFL flow model is obtained and all target entities are determined. For each target entity, a transition structure is generated. The transition structure name is obtained by getting the target entity’s name.

Figure 4.10 Modified control structure for “Complete Travel Order” example
Figure 4.11 Attributes of Transition Structures
As shown in Figure 4.11, a Transition structure consists of three attributes:

- **Transition structure name**: The name of the target entity corresponding to the transition structure.
- **Aggregation property**: If the entity is a JOIN connector, the aggregation property of the transition structure is the same as the aggregation property of the JOIN connector (AND, OR, or XOR). Otherwise, the aggregation property is SIMPLE. This information is retrieved from join activity of WSFL document.
- **Transition(s)**: A transition structure may contain one or more transitions. If the entity type is a JOIN connector, there are multiple transitions in this transition structure. Otherwise, there is only one transition in the structure. The Transition(s) variable of transition structure is a hash table. The key of the hash table is the name of the source entity of a transition, and the value stored in the hash table is an object of type TransCondPair. The TransCondPair object has details about source and target entity names, transition name and condition of that transition. The Transition name is the name of controlLink obtained by WSFL document.

For example, in the process model “Travel Order” (Figure 4.10), the transitions ReserveHotelEndConnTrans and RentCarEndConnTrans point to the same entity: connector EndJoinConn. The transition structure generated for EndJoinConn contains two transitions, namely, ReserveHotelEndConnTrans and RentCarEndConnTrans. The transition structure for EndJoinConn connector is as depicted in Figure 4.12.
4.2.5 Creating *End Activity* Structure

An *End Activity* structure is intended to contain output parameters of a process model. This will be the last entity structure that will be loaded by the *WF Executor*. The *WF Executor* will end the execution of a process instance once the *End Activity* is loaded. The *End Activity* structure is constructed using the *flowSink* element present in the WSFL document.

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

<message name = "TravelOrderResponse">
    <part name = "airTicket" type = "Itinerary"/>
    <part name = "hotelConfirmation" type = "HotelReservationInfo"/>
    <part name = "carConfirmation" type = "CarReservationInfo"/>
</message>
```

![Figure 4.13 flowSink element of “Complete Travel Order” flow model](image)

This structure is generated by the method *createEndActivity*. The *createEndActivity* method parses the *flowSink* element to retrieve relevant information required to populate the *End Activity* structure. Relevant information includes output parameter types and output parameter names. This information is stored in the *Parameter List* member variable of the *End Activity* structure.

The *flowSink* element for “Travel Order” example is given in Figure 4.13. Figure 4.14 illustrates the *End Activity* structure for this example. The WF Executor will populate the parameter value field of parameter list after executing the process instance. The values stored in the *Parameter List of End Activity* will be the final output of the composite web service.
4.2.5: Persisting generated Runtime Workflow Structures

After the runtime workflow structures for the WSFL flow model are generated completely, it is persisted using Java serialization using the method `persistModel`. The name of the file in which the serialized structures are stored is the name of flow model with an extension “.ser”. In our example, the flow model name is `travel order` and hence the file name is `travelOrder.ser`. These persisted serialized runtime workflow structures are later loaded by `WF Executor` to instantiate and enact the workflow instance.
CHAPTER 5
ACTIVITY CODE GENERATOR AND WF EXECUTOR

As shown in the WSFL Engine architecture in Figure 3.3, the Runtime Workflow Structures Generator generates the runtime workflow structures (described in Chapter 4). The Activity Code Generator generates the code necessary to bind the operation in the activities to web services. The runtime workflow structures and the generated activity code are used by the WF Executor to schedule and execute a process instance. This chapter describes the design and implementation of Activity Code Generator and WF Executor in detail. The organization of the remainder of this chapter is as follows. The design and implementation of Activity Code Generator is described in Section 5.1. The design and implementation of WF Executor is presented in section 5.2.

5.1 Design and implementation of Activity Code Generator

Activities are fundamental building blocks of a business process model. In WSFL specifications, activity information is encapsulated by specifying an activity’s input and output parameters. The key activity information is its binding to a web service operation. This information is extracted from the WSFL global model element in the form of a URL to the WSDL implementation document for the corresponding web service. Within an activity, there is no control information such as split or join constructs. Thus, the code generated for an activity contains no control flow information. Also, the Begin Activity (which is the entry point of a process model) and the End Activity (which is the exit point of a business process model) do not contain any tasks. Thus, no code is generated for these entities.
The activity code generated fall into two categories:

- Stub codes generated by AXIS WSDL2Java API
- Code generated to invoke the stub codes

The stub codes generated by AXIS WSDL2Java API contains binding information (serialization and de-serialization details) and location information (endpoint address) for the remote web service. This stub code generates the SOAP message necessary to invoke the required web service and also interprets the SOAP message returned from the web service.

The second part is the code generated to invoke these stub codes. This code will invoke the appropriate method in the stub codes by making use of the service name, port name and operation name specified in the WSFL global model.

Figure 5.1 presents the architecture of Activity Code Generator. It contains a WSFL Parser that will parse the input WSFL document. The implementation of parser makes use of the Apache Xerces XML Parser [18]. The parser retrieves the WSDL documents for all the activities specified in the process model. These WSDL files are inputted to the Apache AXIS WSDL2Java API [11] to generate client stub codes. These stub codes contains binding information for remote web services. The Stub Code Invoker Generator receives input from the WSFL Parser and the WSDL Parser to generated code to invoke the stub codes. The stub codes and code generated to invoke these stub codes is stored in a runtime repository. The WF Executor will make use of these generated activity codes while executing a process instance.

The details of these components are described in the following sections.
Figure 5.1 Architecture of Activity Code Generator

5.1.1 WSFL Parser

The input for WSFL Parser is a WSFL document that has both flow and global model information. The WSFL Parser component parses the WSFL flow model and global model elements to retrieve the URLs for the WSDL implementation documents that contain binding and endpoint information for the corresponding web services. It also retrieves the service names, portType names and operation names for all web services. The output WSDL URLs is used by the Apache AXIS WSDL2Java API to generate the web service client stub codes. The output service name, portType name and operation name for a web service is used by the Stub Code Invoker Generator to generate code that is used to invoke corresponding method of the generated stub codes.
The WSFL flow model element is parsed to retrieve all activity names, their corresponding service providers’ name and corresponding activity operation names. The service provider name is used to fetch the WSDL implementation document URL from the WSFL global model. The activity operation name is used to find the actual endpoint web service operation name from the pluglink element.

```xml
<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>

<serviceProvider name = "EAirticketOrderService"
    type = "TravelTicketOrderType">
    <locator type = "static"
        service = "http://local:8080/axis/services/EAirticketOrder?wsdl"/>
</serviceProvider>

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

Figure 5.2 Binding information for activity BookAirTickets

To illustrate the working of Activity Code Generator, we make use of the example presented in section 4.1. The activity specification and binding information for the BookAirTicket activity in a WSFL document is presented in Figure 5.2. In this example,
the service provider for the activity BookAirTickets is EAirticketOrderService. This name is used to retrieve the WSDL implementation document URL,  
http://localhost:8080/axis/services/EAirticketOrder?wsdl from global model element. The activity operation orderFlightTickets is mapped to the web service operation orderAirTicket by parsing the plugLink element.

Following this process we have found that the activity BookAirTickets can be realized by invoking the orderAirTicket operation on the stub code generated by using the WSDL URL http://localhost:8080/axis/services/EAirticketOrder?wsdl

The service name EAirTicketOrderService, portType name EAirTicketOrder and operation name orderAirTicket are extracted and passed to the Stub Code Invoker Generator component.

5.1.2 AXIS WSDL2Java

The input to the AXIS WSDL2Java component is a WSDL implementation document of a web service. The purpose of this component is to generate the client stub codes to a web service. The stub codes generated by the AXIS WSDL2Java API contains binding information (serialization and de-serialization details) and location information (endpoint address) for the remote web service. The WF Executor later loads and executes this stub codes to bind the activity operation to the remote web service operation. This API is provided by the Apache AXIS toolkit in the class org.apache.axis.wsdl.WSDL2Java. Table 5.1 gives a list of files generated by the WSDL2Java API. This table shows information regarding the type of Java class that is generated for each element type present in the WSDL document. The third column of this table has a list of classes generated for the example EAirticketOrder activity. A detailed explanation of WSDL2Java and client stub codes generation is provided in the AXIS user
guide document [11]. Appendix A, has listed the stub codes generated by the AXIS WSDL2Java API for the WSDL document of EAirticketOrder web service.

<table>
<thead>
<tr>
<th>WSDL Clause</th>
<th>Java Class generated</th>
<th>Classes generated for EAirticketOrder activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>For each entry in the Type section</td>
<td>A Java class is generated</td>
<td>CustomerInfo, Itinerary, DataLocationInfo, Address</td>
</tr>
<tr>
<td></td>
<td>A holder class if this type is used as an</td>
<td>DateLocationInfoHolder, ItineraryHolder</td>
</tr>
<tr>
<td></td>
<td>inout/out parameter</td>
<td></td>
</tr>
<tr>
<td>For each port-Type</td>
<td>A Java interface</td>
<td>EAirticketOrder</td>
</tr>
<tr>
<td>For each binding</td>
<td>A stub class</td>
<td>EAirticketOrderSoapBindingStub</td>
</tr>
<tr>
<td>For each service</td>
<td>A service interface</td>
<td>EAirticketOrderService</td>
</tr>
<tr>
<td></td>
<td>A service implementation (the locator)</td>
<td>EAirticketOrderServiceLocator</td>
</tr>
</tbody>
</table>

### 5.1.3 Stub Code Invoker Generator

The activity code generated for each activity contains the stub codes and the code to invoke the stub codes. The code to invoke the stub codes is generated by the *Stub Code Invoker Generator* component. This component gets the *operation* name that need to be invoked and its corresponding *service* name and *portType* name as its input from the WSFL Parser. It also gets the *operation* name, *service* name and *portType* name from the WSDL document using WSDL Parser. This is done to verify whether the *operation* name, *service* name and *portType* name that are specified in the WSFL document is consistent with those specified in the WSDL document. If there is inconsistency in these names, an exception will be thrown. The output code generated by the *Stub Code Invoker Generator* component is loaded by the WF Executor to invoke the appropriate stub codes method that binds to the remote web service.
The WSDL Parser component is an implementation of the IBM WSDL4J parser [19]. This component takes a WSDL implementation document as its input as mentioned above. The parser is used to retrieve the operation name, service name and portType name of a web service operation from its WSDL document.

This generated code will be in the form of a Java class (.java file). The generated activity code is compiled and stored in a runtime repository. These compiled classes are later loaded by WF Executor while enacting the process instance. While generating activity code all variable names and their type information are synthesized by parsing the details provided in the WSFL document. The generated activity code will invoke the stub classes generated by WSDL2Java API (described in section 5.1.2).

To illustrate, Figure 5.3 depicts the activity code generated to invoke the stub codes for the example BookAirTicket activity. The name of the class is generated by concatenating the process instance name with “$$” and activity name. Process instance name is same as the global model name in the WSFL document. In this example, the process instance name is ETravelBooking and activity name is BookAirTicket. Therefore the class name is ETravelBooking$$BookAirTicket. The activity code generated to invoke the stub codes will have a method called activate. This method takes a vector as input. This input vector contains input parameter values required to invoke the web service. The activate method returns a vector as output. The output vector contains values generated by the remote web service. To invoke a web service we have to instantiate a service locator object. The name of the service locator object is the service name of the remote web service concatenated to the string “Locator”. From Figure 5.2, we find that the service name for the remote web service is EAirTicketOrderService. Using this
information an object of type \textit{EAirticketOrderServiceLocator} is instantiated. Similarly a \textit{portType} object is instantiated using the service locator object. The name of the portType in Figure 5.2 is \textit{EAirTicketOrder}. With this information an object of type \textit{EAirTicketOrder} is instantiated. \textit{Holder} classes are generated if there is more than one parts name in an output message element. From Figure 4.6 we find that output message \textit{BookAirTicketResponse} has two output parts of type, \textit{Itinerary} and \textit{DateLocationInfo}. Therefore two holder objects of type \textit{ItineraryHolder} and \textit{DateLocationInfoHolder} are instantiated. From Figure 5.2 we find that the remote web service operation name is \textit{orderAirTicket}. Therefore code is generated to invoke operation \textit{orderAirTicket}. The output values are retrieved from the holder classes and populated into the output vector after the web service is invoked. As explained in section 5.1.1, the \textit{service} name, \textit{portType} name and \textit{operation} name are retrieved using WSFL Parser. The \textit{activate} method throws an exception if there is any problem in binding to the remote web service.
Figure 5.3 Code Generated to Invoke the Stub codes for Activity BookAirTickets

5.2 Design and Implementation of the WF Executor

The execution of a process instance is performed by the *WF Executor*, which forms the core of the runtime environment. The *WF Executor* uses a multi-thread architecture to manage the execution of process instances. This multi-threaded architecture allows concurrent execution of multiple process instances in a very efficient way.

Figure 5.4 depicts the components of the *WF Executor* and interaction among these components. The *WF Executor* consists of two main parts:

- Scheduler
- Activity Handler

The Scheduler schedules the activities as dictated by the transition structures present in the process model. The Activity Handler loads the generated activity codes and executes them to make web service invocation. The design and implementation of the Scheduler component is presented in section 5.2.1. The design and implementation of Activity Handler component is presented in section 5.2.2.

Figure 5-4 Architecture of WF Executor

5.2.1 Implementation of Scheduler

The Scheduler component of the WF Executor schedules the activities as dictated by the transition structures. The input to this component is a process instance name and a set of input values required for executing the process instance.

For each process instance, a Scheduler thread is created. The Scheduler loads the runtime workflow structures for a process instance from the runtime repository using the
process instance name. It uses these runtime workflow structures to enforce inter-activity dependencies during activity scheduling. There are two kinds of inter-activity dependency: control dependency and data dependency. Transition structures are used to enforce control dependencies, and dataflow structures are used to enforce the data dependencies. The Scheduler evaluates the transitions and dataflow mapping between activity execution and thus controls the execution and data mapping of the process instance. The evaluation of transition structures and parameter mapping is triggered upon completion of a source entity in the transition structures and dataflow structures. An activity is completed when all the task items inside the activity are finished. The Connector, Begin Activity and End Activity entities are completed as soon as they are scheduled.

After an entity is completed, the Scheduler will take the following steps to schedule the execution of next entity:

1. **Dataflow mapping**: If the completed entity is an activity, the scheduler maps its output data to the input parameters of the appropriate entities. The dataflow structures present in the runtime workflow structures provide specific information for this data mapping.

2. **Transition structure evaluation**: The Scheduler retrieves the transition structures, which contains a transition with the completed entity as the source entity, from the hash table of the transition structures and evaluates them. If the aggregation property of a transition structure is SIMPLE, there is only one transition in the transition structure. In this case, the evaluation of the transition structure is actually the evaluation of the condition on this transition. If there is no condition on the transition, the result of the
evaluation is always true. The target entity of the transition structure is not scheduled until the condition for the transition is evaluated to false.

If the aggregation property is not SIMPLE, it means that the target entity must be a JOIN connector. In this case, after the transition whose source is the completed entity is evaluated to true, the Scheduler also needs to evaluate the aggregation expression, which is formed by the transitions and the aggregation operator (AND, OR, or XOR). For example, in figure 4.10, to initiate EndJoinConn, which is a join connector with AND aggregation property, both ReserveHotelEndConnTrans and RentCarEndConnTrans should be completed.

3. Entity Scheduling and execution: If the scheduled entity is an activity, the Scheduler creates an Activity Handler thread and delegates the execution of the activity to it. At the end of the execution, the Activity Handler thread passes the output data of the activity to the Scheduler to do the dataflow mapping. It also notifies the Scheduler about the completion of its execution to trigger the transition structure evaluation and then terminates. If the scheduled entity is a connector, the scheduler sets the status of the connector to “complete” and does nothing. If the scheduled entity is the End Activity of the process model, the process instance is completed.

The Scheduler repeats steps 1 to 3 until End Activity structure is loaded.

5.2.2 Implementation of Activity Handler

When an activity is scheduled, the Scheduler creates an Activity Handler thread to load the activity code, and execute it. The input to this component is a set of input values required to invoke a web service operation. At the end of execution, the Activity Handler thread passes the output data of the activity to the Scheduler to do the dataflow mapping. The output data is the set of output values generated by the invoked web service.
Since each Activity Handler is executed as a thread, the Scheduler can continue on and all parallel activities in a process model can be executed concurrently by multiple, concurrent Activity Handler threads. The scheduled Activity Handler will load the corresponding activity code from the runtime repository. The activity code consists of client stub codes and code to invoke these stub codes. The Activity Handler loads appropriate code to invoke the stub codes from the runtime repository. It passes the input parameter values to loaded code to perform web service invocation. The loaded class instantiates an object of stub code using the stub classes present in the runtime repository. The stub codes contain binding information and location information to the remote web services. The stub codes builds a SOAP message containing the request and sends it to the remote web service. It later receives a SOAP response containing returned data. The stub codes use APIs provided by AXIS tool kit to de-serialize and serialize Java objects to and from XML, that are be embedded in SOAP messages. At the end of execution, the Activity Handler thread passes the output data of the activity to the Scheduler. It also notifies the Scheduler about the completion of its execution to trigger transition structure evaluation and then terminates.

An example SOAP message sent by the generated stub codes corresponding to activity BookAirTickets activity to the EAirTicketOrder web service is given in Figure 5.5. Figure 5.6 shows the response SOAP message sent from the EAirTicketOrder web service to the generated stub code for the activity BookAirTickets.
Figure 5.5 SOAP request sent from activity code to EAirticketOrder web service
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Date: Tue, 03 Sep 2002 14:46:23 GMT
Server: Apache Tomcat/4.0.4 (HTTP/1.1 Connector)
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns1:orderAirTicketResponse soapenv:encodingStyle= "http://schemas.xmlsoap.org/soap/encoding/">
      <out1 href="#id0"/>
      <out2 href="#id1"/>
      <multiRef id="id0" soapenc:root="0" soapenv:encodingStyle= "http://schemas.xmlsoap.org/soap/encoding/">
        <price xsi:type="xsd:int">536</price>
        <startLocation xsi:type="xsd:string">Gainesville, Florida</startLocation>
        <travelCompany xsi:type="xsd:string">Lufthansa</travelCompany>
        <startTime xsi:type="xsd:string">10:00 PM</startTime>
        <destinationLocation xsi:type="xsd:string">Atlanta</destinationLocation>
        <arrivalTime xsi:type="xsd:string">5:10 AM</arrivalTime>
        <ticketNumber xsi:type="xsd:string">504448</ticketNumber>
        <contactPhoneNumber xsi:type="xsd:string">Samantha Roberts, Customer Service, Phone: (852) 039-8962</contactPhoneNumber>
        <additionalInfo xsi:type="xsd:string">As our valuable customer we are giving your one ticket for free</additionalInfo>
        <departingDate xsi:type="xsd:string">July 23, 2002</departingDate>
      </multiRef>
      <multiRef id="id1" soapenc:root="0" soapenv:encodingStyle= "http://schemas.xmlsoap.org/soap/encoding/">
        <time xsi:type="xsd:string">5:10 AM</time>
        <date xsi:type="xsd:string">July 23, 2002</date>
        <location xsi:type="xsd:string">Atlanta</location>
      </multiRef>
    </ns1:orderAirTicketResponse>
  </soapenv:Body>
</soapenv:Envelope>

Figure 5.6 SOAP response from EAirticketOrder web service to activity code
CHAPTER 6
SUMMARY AND CONCLUSION

In this thesis, we have described the design and implementation of a WSFL Engine to support the development of an Intelligent Registry. The objective of the Intelligent Registry is to dynamically and automatically generate a composite web service plan, based on end users request and taking into consideration the requestor and service providers’ constraints. The composite service plan is outputted in the form of a WSFL document. After the WSFL document is generated, the WSFL Engine is invoked to execute the composite service plan.

The WSFL Engine itself is implemented as a web service. Thus, although it was designed to support the Intelligent Registry, the WSFL Engine can be invoked by any web service client that requires the enactment of a workflow instance that is represented by a WSFL document.

The implementation of the WSFL Engine followed the code generation approach that was used in the implementation of Dynaflow [13]. It consists of three main components:

- Runtime Workflow Structures Generator
- Activity Code Generator
- WF Executor

Based on the specification for a composite service defined using WSFL document, the Runtime Workflow Structures Generator generates the runtime workflow structures used by the WF Executor to schedule and execute a process instance. These structures determine the control and data flows of a process instance. The generated runtime
workflow structures consist of entity, transition, and dataflow structures. These are persisted in a runtime repository in form of a Java serialized structure.

The *Activity Code Generator* generates the code necessary to bind to web services. The generate activity code consists of client stubs and the code to invoke these stubs. The stub code generated using the AXIS WSDL2Java API, contains binding information (serialization and de-serialization details) and location information (endpoint address) for the remote web service. This stub code generates and interprets SOAP messages. The generated activity code is compiled and stored in a runtime repository.

The runtime workflow structures and the generated activity code are used by the *WF Executor* to schedule and execute a process instance. The *WF Executor* uses runtime workflow structures to enforce inter-activity dependencies during activity scheduling. When scheduling the activities for a process instance, it simply loads the Java activity classes from the run time repository and executes the code directly to perform specified tasks. *WF Executor* uses multi-thread architecture to manage the execution of process instances. This multi-threaded architecture allows concurrent execution of multiple process instances in a very efficient way.

The code generation approach results in a lightweight and flexible WSFL Engine. It is lightweight because for each invocation of a process model the workflow engine need not have to interpret the WSFL document in order to determine the flow and on how to bind to the web services. Taking advantage of the performance of compiled classes, the *WSFL Engine* is efficient. The separation of the flow information in the runtime workflow structures and the binding information in the activity code leads to loose coupling and better flexibility. Changes to an activity can be made by modifying the
WSFL global model. The code for the modified activity is re-generated and re-loaded using Java’s class reloading capability. In this manner, any change made to an activity specification will not affect the flow structure, but will be reflected in the execution of the activity as long as the re-generated code is placed in the run-time repository before the execution of the activity. Thus, flexibility is maintained without sacrificing performance.

One issue with this WSFL Engine is that at present it cannot handle subflows. In WSFL subflows are treated in the same way as activities. A subflow is realized by invoking a composite web service that is executed on a workflow engine. A WSDL document for the composite web service has to be provided in the global model of the WSFL document to invoke the web service operation of the composite service. In our case, the WSDL document for the composite web service is the WSDL document of the WSFL Engine, which has an interface InvokeWFInstance that takes all the input parameters required to invoke the composite service in the form of a vector. As a result this WSDL document does not have any details regarding the number and type of parameters required to invoke the composite service. This information is present only in the WSFL document. The WSFL Engine makes use of the AXIS WSDL2Java toolkit to generate client stub code. Since, we do not have enough information regarding the input and output parameter types in the WSDL document of the composite web service, the WSDL2Java API cannot generate proper client stub codes. Because of this the WSFL Engine cannot invoke a composite web service. This problem could be solved by generating a WSDL document for the composite web service using its WSFL document. This WSDL document will have all information regarding the input and output parameter types required to invoke the composite web service. This WSDL document could be used
by the AXIS Java2WSDL toolkit to generate client stub codes that are used to bind to the composite web service. This could be done by developing a special component to generate WSDL document from WSFL document.
APPENDIX
STUB CODES GENERATED BY AXIS TOOL KIT

This appendix presents the web service client stub codes generated by AXIS toolkit WSDL2Java API for the WSDL document of the web service EAirticketOrder. The WSDL document for this EAirticketOrder web service is presented in section 1 of this chapter. Section 2 presents the code for EAirticketOrder class. Section 3 presents the code for EAirticketOrderService class. Section 4 presents the code for EAirticketOrderServiceLocator. Section 5 presents the code for EAirticketOrderSoapBindingStub.

1 WSDL Document for EAirticketOrder web service

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions encoding="UTF-8"/>
<wsdl:definitions

xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://localhost:8080/axis/services/EAirticketOrder/axis/services/EAirticketOrder-impl"
xmlns:intf="http://localhost:8080/axis/services/EAirticketOrder/axis/services/EAirticketOrder"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/
xmlns:tns1="urn:CustomerInfo"
xmlns:tns2="http://TravelTicketOrderType.completeTravelBookingDemo.myprograms2"
xmlns:tns3="urn:TravelInfo"
xmlns:tns4="urn:Itinerary"
xmlns:tns5="urn:DateLocationInfo"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<wsdl:types>
<schema targetNamespace="urn:CustomerInfo"
xmlns="http://www.w3.org/2001/XMLSchema">
<import namespace="http://schemas.xmlsoap.org/soap/encoding/
<complexType name="CustomerInfo">
<sequence>
<element name="customerName" nillable="true" type="xsd:string" />
<element name="address" nillable="true" type="tns2:Address" />
<element name="creditCardNumber" nillable="true" type="xsd:int" />
<element name="contactNumber" nillable="true" type="xsd:string" />
</sequence>
</complexType>
```
<element name="CustomerInfo" nillable="true" type="tns1:CustomerInfo" />
</schema>
<schema targetNamespace="http://TravelTicketOrderType.completeTravelBookingDemo.myprogs2" xmlns="http://www.w3.org/2001/XMLSchema">
<import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
<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>
</schema>
<schema targetNamespace="urn:TravelInfo" xmlns="http://www.w3.org/2001/XMLSchema">
<import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
<complexType name="TravelInfo">
<sequence>
<element name="departingDate" nillable="true" type="xsd:string" />
<element name="departingTime" nillable="true" type="xsd:string" />
<element name="startLocation" nillable="true" type="xsd:string" />
<element name="numberOfPassengers" type="xsd:int" />
<element name="destination" nillable="true" type="xsd:string" />
</sequence>
</complexType>
<element name="TravelInfo" nillable="true" type="tns3:TravelInfo" />
</schema>
<schema targetNamespace="urn:Itinerary" xmlns="http://www.w3.org/2001/XMLSchema">
<import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
<complexType name="Itinerary">
<sequence>
<element name="price" type="xsd:int" />
<element name="startLocation" nillable="true" type="xsd:string" />
<element name="travelCompany" nillable="true" type="xsd:string" />
<element name="startTime" nillable="true" type="xsd:string" />
<element name="destinationLocation" nillable="true" type="xsd:string" />
<element name="arrivalTime" nillable="true" type="xsd:string" />
<element name="ticketNumber" nillable="true" type="xsd:string" />
<element name="contactPhoneNumber" nillable="true" type="xsd:string" />
<element name="additionalInfo" nillable="true" type="xsd:string" />
<element name="departingDate" nillable="true" type="xsd:string" />
</sequence>
</complexType>
<element name="Itinerary" nillable="true" type="tns4:Itinerary" />
</schema>
<schema targetNamespace="urn:DateLocationInfo" xmlns="http://www.w3.org/2001/XMLSchema">
<import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
<complexType name="DateLocationInfo">
<sequence>
<element name="time" nillable="true" type="xsd:string" />
<element name="date" nillable="true" type="xsd:string" />
</sequence>
</complexType>
</schema>

<element name="location" nillable="true" type="xsd:string" />
</sequence>
</complexType>
<element name="DateLocationInfo" nillable="true" type="tns5:DateLocationInfo" />
</schema>
<schema targetNamespace="http://schemas.xmlsoap.org/soap/encoding/"
xmlns="http://www.w3.org/2001/XMLSchema">
<import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
<element name="Array" nillable="true" type="soapenc:Array" />
</schema>
</wsdl:types>
<wsdl:message name="getOperationDescsResponse">
<wsdl:part name="return" type="soapenc:Array" />
</wsdl:message>
<wsdl:message name="getOperationDescsRequest" />
<wsdl:message name="orderAirTicketRequest">
<wsdl:part name="in0" type="tns1:CustomerInfo" />
<wsdl:part name="in1" type="tns3:TravelInfo" />
</wsdl:message>
<wsdl:portType name="EAirticketOrder">
<wsdl:operation name="orderAirTicket" parameterOrder="in0 in1 out1 out2">
<wsdl:input message="intf:orderAirTicketRequest" name="orderAirTicketRequest" />
<wsdl:output message="intf:orderAirTicketResponse" name="orderAirTicketResponse" />
</wsdl:operation>
<wsdl:operation name="getOperationDescs">
<wsdl:input message="intf:getOperationDescsRequest" name="getOperationDescsRequest" />
<wsdl:output message="intf:getOperationDescsResponse" name="getOperationDescsResponse" />
</wsdl:operation>
<wsdl:operation name="getOperationDescByName" parameterOrder="in0">
<wsdl:input message="intf:getOperationDescByNameRequest" name="getOperationDescByNameRequest" />
<wsdl:output message="intf:getOperationDescByNameResponse" name="getOperationDescByNameResponse" />
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="EAirticketOrderSoapBinding" type="intf:EAirticketOrder">
<wsdl:soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
<wsdl:operation name="orderAirTicket" />
<wsdl:soap:operation soapAction="" />
<wsdl:input name="orderAirTicketRequest">
<wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
namespace="http://localhost:8080/axis/services/EAirticketOrder/axis/services/EAirticketOrder" use="encoded" />
</wsdl:input>
</wsdl:operation>
<wsdl:output name="orderAirTicketResponse">
<wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
namespace="http://localhost:8080/axis/services/EAirticketOrder/axis/services/EAirticketOrder" use="encoded" />
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="getOperationDescs">
<wsdlsoap:operation soapAction="" />
<wsdl:input name="getOperationDescsRequest">
<wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
namespace="http://localhost:8080/axis/services/EAirticketOrder/axis/services/EAirticketOrder" use="encoded" />
</wsdl:input>
<wsdl:output name="getOperationDescsResponse">
<wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
namespace="http://localhost:8080/axis/services/EAirticketOrder/axis/services/EAirticketOrder" use="encoded" />
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="getOperationDescByObjectName">
<wsdlsoap:operation soapAction="" />
<wsdl:input name="getOperationDescByObjectNameRequest">
<wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
namespace="http://localhost:8080/axis/services/EAirticketOrder/axis/services/EAirticketOrder" use="encoded" />
</wsdl:input>
<wsdl:output name="getOperationDescByObjectNameResponse">
<wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
namespace="http://localhost:8080/axis/services/EAirticketOrder/axis/services/EAirticketOrder" use="encoded" />
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="EAirticketOrderService">
<wsdlport binding="intf:EAirticketOrderSoapBinding" name="EAirticketOrder">
<wsdlsoap:address
location="http://localhost:8080/axis/services/EAirticketOrder" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
2 Stub Code EAirticketOrder class

/**
 * EAirticketOrder.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package proxy;
public interface EAirticketOrder extends java.rmi.Remote {

    public void orderAirTicket(proxy.CustomerInfo in0,
                                proxy.TravelInfo in1,
                                proxy.holders.ItineraryHolder out1,
                                proxy.holders.DateLocationInfoHolder out2)
                                throws java.rmi.RemoteException;

    public java.lang.Object[] getOperationDescByName(
                                java.lang.String in0)
                                throws java.rmi.RemoteException;

    public java.lang.Object[] getOperationDescs()
                                throws java.rmi.RemoteException;
}

3 Stub Code EAirticketOrderService

/**
 * EAirticketOrderService.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package proxy;
public interface EAirticketOrderService extends 
                javax.xml.rpc.Service {

    public String getEAirticketOrderAddress();

    public proxy.EAirticketOrder getEAirticketOrder()
                                throws javax.xml.rpc.ServiceException;

    public proxy.EAirticketOrder getEAirticketOrder(
                                java.net.URL portAddress)
                                throws javax.xml.rpc.ServiceException;
}

** 4 Stub Code EAirticketOrderServiceLocator class

/**
 * EAirticketOrderServiceLocator.java
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package proxy;

public class EAirticketOrderServiceLocator extends org.apache.axis.client.Service implements proxy.EAirticketOrderService {

    // Use to get a proxy class for EAirticketOrder
    private final java.lang.String EAirticketOrder_address = "http://localhost:8080/axis/services/EAirticketOrder";

    public String getEAirticketOrderAddress() {
        return EAirticketOrder_address;
    }

    public proxy.EAirticketOrder getEAirticketOrder() throws javax.xml.rpc.ServiceException {
        java.net.URL endpoint;
        try {
            endpoint = new java.net.URL(EAirticketOrder_address);
        }
        catch (java.net.MalformedURLException e) {
            return null;
        }
        return getEAirticketOrder(endpoint);
    }

    public proxy.EAirticketOrder getEAirticketOrder(java.net.URL portAddress) throws javax.xml.rpc.ServiceException {
        try {
            return new proxy.EAirticketOrderSoapBindingStub(portAddress, this);
        }
        catch (org.apache.axis.AxisFault e) {
            return null; // ???
        }
    }

    /**
     * For the given interface, get the stub implementation.
     * If this service has no port for the given interface,
     * then ServiceException is thrown.
     */
    public java.rmi.Remote getPort(Class serviceEndpointInterface) throws javax.xml.rpc.ServiceException {
        try {
            return null;
        }
    }
}
if (proxy.EAirticketOrder.class.isAssignableFrom(serviceEndpointInterface)) {
    return new proxy.EAirticketOrderSoapBindingStub(new java.net.URL(EAirticketOrder_address), this);
}

catch (Throwable t) {
    throw new javax.xml.rpc.ServiceException(t);
}

throw new javax.xml.rpc.ServiceException("There is no stub implementation for the interface: " + (serviceEndpointInterface == null ? "null" : serviceEndpointInterface.getName()));
}

5 Stub Code for EAirticketOrderSoapBindingStub

/**
 * EAirticketOrderSoapBindingStub.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package proxy;

public class EAirticketOrderSoapBindingStub extends org.apache.axis.client.Stub implements proxy.EAirticketOrder {
    private java.util.Vector cachedSerClasses = new java.util.Vector();
    private java.util.Vector cachedSerQNames = new java.util.Vector();
    private java.util.Vector cachedSerFactories = new java.util.Vector();
    private java.util.Vector cachedDeserFactories = new java.util.Vector();

    public EAirticketOrderSoapBindingStub() throws org.apache.axis.AxisFault {
        this(null);
    }

    public EAirticketOrderSoapBindingStub(java.net.URL endpointURL, javax.xml.rpc.Service service) throws org.apache.axis.AxisFault {
        this(service);
        super.cachedEndpoint = endpointURL;
    }

    public EAirticketOrderSoapBindingStub(javax.xml.rpc.Service service) throws org.apache.axis.AxisFault {
        try {
            if (service == null) {
                super.service = new org.apache.axis.client.Service();
            } else {
                // Handle service != null here
            }
        } catch (Throwable t) {
            throw new javax.xml.rpc.ServiceException(t);
        }
    }

    // Implement methods for EAirticketOrder
}
super.service = service;
}
Class cls;
javax.xml.namespace.QName qName;
Class beansf = org.apache.axis.encoding.ser.BeanSerializerFactory.class;
Class beandf = org.apache.axis.encoding.ser.BeanDeserializerFactory.class;
Class enumsf = org.apache.axis.encoding.ser.EnumSerializerFactory.class;
Class enumdf = org.apache.axis.encoding.ser.EnumDeserializerFactory.class;
Class arraysf = org.apache.axis.encoding.ser.ArraySerializerFactory.class;
Class arraydf = org.apache.axis.encoding.ser.ArrayDeserializerFactory.class;
Class simplesf = org.apache.axis.encoding.ser.SimpleNonPrimitiveSerializerFactory.class;
Class simplesdf = org.apache.axis.encoding.ser.SimpleDeserializerFactory.class;
qName = new javax.xml.namespace.QName("urn:CustomerInfo", "CustomerInfo");
cachedSerQNames.add(qName);
cls = proxy.CustomerInfo.class;
cachedSerClasses.add(cls);
cachedSerFactories.add(beansf);
cachedDeserFactories.add(beandf);
qName = new javax.xml.namespace.QName("urn:TravelInfo", "TravelInfo");
cachedSerQNames.add(qName);
cls = proxy.TravelInfo.class;
cachedSerClasses.add(cls);
cachedSerFactories.add(beansf);
cachedDeserFactories.add(beandf);
qName = new javax.xml.namespace.QName("urn:DateLocationInfo", "DateLocationInfo");
cachedSerQNames.add(qName);
cls = proxy.DateLocationInfo.class;
cachedSerClasses.add(cls);
cachedSerFactories.add(beansf);
cachedDeserFactories.add(beandf);
qName = new javax.xml.namespace.QName("urn:Itinerary", "Itinerary");
cachedSerQNames.add(qName);
cls = proxy.Itinerary.class;
cachedSerClasses.add(cls);
cachedSerFactories.add(beansf);
cachedDeserFactories.add(beandf);
qName = new javax.xml.namespace.QName("http://TravelTicketOrderType.completeTravelBookingDemo.myprogs2", "Address");
cachedSerQNames.add(qName);
cls = proxy.Address.class;
cachedSerClasses.add(cls);
cachedSerFactories.add(beansf);
cachedDeserFactories.add(beandf);
}
catch(java.lang.Exception t) {
    throw org.apache.axis.AxisFault.makeFault(t);
}
}

private org.apache.axis.client.Call createCall() throws java.rmi.RemoteException {
    try {
        org.apache.axis.client.Call call =
            (org.apache.axis.client.Call)
                super.service.createCall();
        if (super.maintainSessionSet) {
            call.setMaintainSession(super.maintainSession);
        }
        if (super.cachedUsername != null) {
            call.setUsername(super.cachedUsername);
        }
        if (super.cachedPassword != null) {
            call.setPassword(super.cachedPassword);
        }
        if (super.cachedEndpoint != null) {
            call.setTargetEndpointAddress(super.cachedEndpoint);
        }
        if (super.cachedTimeout != null) {
            call.setTimeout(super.cachedTimeout);
        }
        java.util.Enumeration keys =
            super.cachedProperties.keys();
        while (keys.hasMoreElements()) {
            String key = (String) keys.nextElement();
            if (call.isPropertySupported(key))
                call.setProperty(key,
                    super.cachedProperties.get(key));
            else
                call.setScopedProperty(key,
                    super.cachedProperties.get(key));
        }
        // All the type mapping information is registered
        // when the first call is made.
        // The type mapping information is actually registered
        in
        // the TypeMappingRegistry of the service, which
        // is the reason why registration is only needed for
        the first call.
        synchronized (this) {
            if (firstCall()) {
                // must set encoding style before registering
                serializers
                call.setEncodingStyle(
                    org.apache.axis.Constants.URI_SOAP11_ENC);
            }
for (int i = 0; i < cachedSerFactories.size(); ++i) {
    Class cls = (Class) cachedSerClasses.get(i);
    javax.xml.namespace.QName qName = (javax.xml.namespace.QName) cachedSerQNames.get(i);
    Class sf = (Class) cachedSerFactories.get(i);
    Class df = (Class) cachedDeserFactories.get(i);
    call.registerTypeMapping(cls, qName, sf, df, false);
}
return call;
}

public void orderAirTicket(proxy.CustomerInfo in0, proxy.TravelInfo in1, proxy.holders.ItineraryHolder out1, proxy.holders.DateLocationInfoHolder out2) throws java.rmi.RemoteException{
if (super.cachedEndpoint == null) {
    throw new org.apache.axis.NoEndPointException();
}
org.apache.axis.client.Call call = createCall();
javax.xml.namespace.QName p0QName = new javax.xml.namespace.QName("", "in0");
call.addParameter(p0QName, new proxy.CustomerInfo.class, javax.xml.rpc.ParameterMode.IN);
javax.xml.namespace.QName p1QName = new javax.xml.namespace.QName("", "in1");
call.addParameter(p1QName, new proxy.TravelInfo.class, javax.xml.rpc.ParameterMode.IN);
javax.xml.namespace.QName p2QName = new javax.xml.namespace.QName("", "out1");
call.addParameter(p2QName, new proxy.Itinerary.class, javax.xml.rpc.ParameterMode.OUT);
javax.xml.namespace.QName p3QName = new javax.xml.namespace.QName("", "out2");
call.addParameter(p3QName, new proxy.DateLocationInfo.class, javax.xml.rpc.ParameterMode.OUT);
call.setReturnType(org.apache.axis.encoding.XMLType.AXIS_VOID);
call.setUseSOAPAction(true);
call.setSOAPActionURI("");
call.setOperationStyle("rpc");
call.setOperationName(new javax.xml.namespace.QName("http://localhost:8080/axis/services/EAirticketOrder/axis/services/EAirticketOrder", "orderAirTicket"));

Object resp = call.invoke(new Object[] {in0, in1});

if (resp instanceof java.rmi.RemoteException) {
    throw (java.rmi.RemoteException)resp;
} else {
    java.util.Map output;
    output = call.getOutputParams();
    try {
        out1.value = (proxy.Itinerary) output.get(new javax.xml.namespace.QName("", "out1"));
    } catch (java.lang.Exception e) {
        out1.value = org.apache.axis.utils.JavaUtils.convert(output.get(new javax.xml.namespace.QName("", "out1")), proxy.Itinerary.class);
    }
    try {
        out2.value = (proxy.DateLocationInfo) output.get(new javax.xml.namespace.QName("", "out2"));
    } catch (java.lang.Exception e) {
        out2.value = org.apache.axis.utils.JavaUtils.convert(output.get(new javax.xml.namespace.QName("", "out2")), proxy.DateLocationInfo.class);
    }
}

public java.lang.Object[]
getOperationDescByName(java.lang.String in0) throws java.rmi.RemoteException{
    if (super.cachedEndpoint == null) {
        throw new org.apache.axis.NoEndPointException();
    }
    org.apache.axis.client.Call call = createCall();
    javax.xml.namespace.QName p0QName = new javax.xml.namespace.QName("", "in0");
    call.addParameter(p0QName, new javax.xml.namespace.QName("http://www.w3.org/2001/XMLSchema", "string"), java.lang.String.class, javax.xml.rpc.ParameterMode.IN);
    call.setReturnType(new javax.xml.namespace.QName("http://schemas.xmlsoap.org/soap/encoding/", "Array"));
    call.setUseSOAPAction(true);
    call.setSOAPActionURI("");
    call.setOperationStyle("rpc");
    call.setOperationName(new javax.xml.namespace.QName("http://localhost:8080/axis/services/EAirticketOrder/axis/services/EAirticketOrder", "getOperationDescByName"));

    Object resp = call.invoke(new Object[] {in0});
    if (resp instanceof java.rmi.RemoteException) {
        throw (java.rmi.RemoteException)resp;
    }
else {
    try {
        return (java.lang.Object[]) resp;
    } catch (java.lang.Exception e) {
        return (java.lang.Object[])
    }
}

org.apache.axis.utils.JavaUtils.convert(resp,
java.lang.Object[].class);
}
}

public java.lang.Object[] getOperationDescs() throws
java.rmi.RemoteException{
    if (super.cachedEndpoint == null) {
        throw new org.apache.axis.NoEndPointException();
    }
    org.apache.axis.client.Call call = createCall();
call.setReturnType(new
javax.xml.namespace.QName("http://schemas.xmlsoap.org/soap/encoding/",
"Array"));
call.setUseSOAPAction(true);
call.setSOAPActionURI("");
call.setOperationStyle("rpc");
call.setOperationName(new
javax.xml.namespace.QName("http://localhost:8080/axis/services/EAirticketOrder/axis/services/EAirticketOrder",
"getOperationDescs"));

Object resp = call.invoke(new Object[] {});

if (resp instanceof java.rmi.RemoteException) {
    throw (java.rmi.RemoteException)resp;
} else {
    try {
        return (java.lang.Object[]) resp;
    } catch (java.lang.Exception e) {
        return (java.lang.Object[])
    }
}
}
}
LIST OF REFERENCES


BIOGRAPHICAL SKETCH

Raman N. Chikkamagalur is a native of Arsikere, India. He earned his high school diploma from St. Mary’s High School, Arsikere. He earned his Bachelor of Engineering in the field of computer science at Sri Jayachamarajendra College of Engineering, Mysore University, Mysore, India. After earning his bachelor’s degree Raman Chikkamagalur worked in companies like Sasken Technologies and Lucent Technologies. In August 2000, he went to pursue his master’s at the University of Florida, Gainesville, in the field of computer engineering. He likes the game of cricket, a popular game in India.