

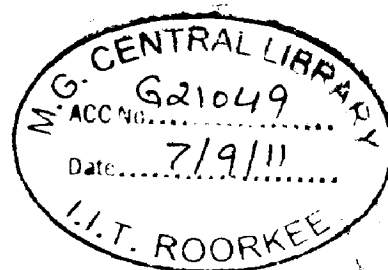
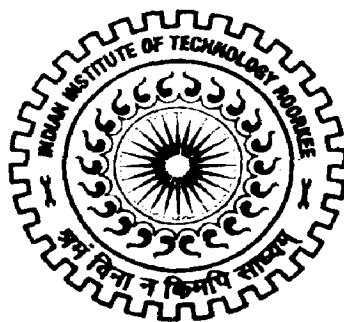
A FORMAL FRAMEWORK FOR WEB SERVICE COMPOSITION

A DISSERTATION

*Submitted in partial fulfillment of the
requirements for the award of the degree
of*
**MASTER OF TECHNOLOGY
in
INFORMATION TECHNOLOGY**

By

VIJAY VERMA



DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)
JUNE, 2011


CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in the dissertation entitled “**A FORMAL FRAMEWORK FOR WEB SERVICE COMPOSITION**” towards the partial fulfillment of the requirement for the award of the degree of **Master of Technology in Information Technology** submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee, Uttarakhand (India) is an authentic record of my own work carried out during the period from July 2010 to June 2011, under the guidance of **Dr. Rajdeep Niyogi, Assistant Professor**, Department of Electronics and Computer Engineering, IIT Roorkee.

The matter presented in this dissertation has not been submitted by me for the award of any other degree of this or any other Institute.

Date: 02/06/2011

Place: Roorkee


(VIJAY VERMA)

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date:

Place: Roorkee


(Dr. RAJDEEP NIYOGI)

Assistant Professor

Department of Electronics and Computer Engineering

IIT Roorkee.

Table of Contents

Candidate's Declaration & Certificate	i
Acknowledgements	ii
Abstract	iii
Table of Contents	iv
List of Figures	vi
1. Introduction and Statement of the Problem	1
1.1 Introduction.....	1
1.2 Motivation.....	2
1.3 Statement of the Problem.....	2
1.4 Organization of the Report.....	3
2. Background and Literature Review	4
2.1 Web Services.....	4
2.2 Understanding the basics of web service.....	7
2.2.1 Simple Object Access Protocol(SOAP).....	8
2.2.2 Web Service Description Language(WSDL).....	9
2.2.3 Discovering Web Services.....	10
2.2.4 More About WSDL.....	11
2.3 Web Service Composition.....	13
3. Formal Model For Web Service Composition	14
3.1 The Formal Model.....	14
3.1.1 Message Type.....	15
3.1.2 For Single Service.....	16
3.1.3 For Web Service Composition.....	17
3.2 Features included in Formal Model.....	19
3.2.1 Activeness Constraints.....	19
3.2.2 Relationships between Component Web Services.....	19

3.2.3 The Flow Web Service.....	20
3.2.4 Verification by using Pi-calculus.....	21
3.2.5 QoS Parameters.....	23
4. Implementation	25
4.1 Implementation of the service as a single service.....	25
4.2 Implementation of the service as a composite service.....	32
4.2.1 JOpera for Eclipse.....	32
4.2.2 Some Definitions related with JOpera.....	32
5. Results and Discussions	40
5.1 The Result of the service as a single web service.....	40
5.2 The Result of the service as a composite web service.....	41
5.3 Mapping of composite web service.....	42
5.4 Verification Using Pi-calculus.....	45
6. Conclusion and Future Work	48
6.1 Conclusion.....	48
6.2 Future Work.....	48
REFERENCES.....	49
APPENDIXES.....	52

LIST OF FIGURES

Figure 2.1	The basic layers of Web services	5
Figure 2.2	A common scenario of Web services in use.....	6
Figure 2.3	The Model-View-Controller paradigm	7
Figure 2.4	Structure of a Web-based SOAP message	8
Figure 2.5	Dynamic communication by inspecting WSDL.....	9
Figure 2.6	A UDDI Registry as a conceptual phone book	10
Figure 2.7	The WSDL Specification in a nutshell.....	12
Figure 3.1	A kind of single service model	14
Figure 3.2	The classification of message types	16
Figure 3.3	Verification Framework Using Pi-Calculus	22
Figure 4.1	Adding class to the package	27
Figure 4.2	Build Successful message	28
Figure 4.3	Successful Run Message at console.....	29
Figure 4.4	WSDL binding details.....	30
Figure 4.5	The Result of invoking enrollDecipher	31
Figure 4.6	Creating a program in JOpera.....	34
Figure 4.7	Java Snippet component for getcourse program.....	35
Figure 4.8	Running the program as a process.....	36
Figure 4.9	Data Flow Diagram for the Process	37
Figure 4.10	Run configuration for the process	38

Figure 4.11	Control Flow Diagram of the Composite Process.....	39
Figure 4.12	Data Flow Diagram of the Composite Process.....	39
Figure 5.1	The Result of service as single web service	40
Figure 5.2	The Result of service as composite web service.....	41
Figure 5.3	Message Interaction Diagram.....	42
Figure 5.4	Channels and Messages interaction for composite servic.....	45
Figure 5.5	Channels and Messages interaction between components.....	45

Chapter 1

Introduction and Statement of the Problem

1.1 Introduction

Web services provide a standard way to ensure the interoperability among different software applications running on a variety of platforms. Organizations use the Web service technology in Enterprise applications and business-to-business integration on the Internet. In each of these two categories, the Web services can vary in complexity from simple functions of query- answer type, to sophisticated long term transactions among several business partners. Regardless of the application, the web services are used for flexible integration of loosely coupled systems that can be decomposed and recomposed to reflect the dynamic nature of the business. The Web services promise to turn the Web from a static collection of documents into a vast library of programs. This is the reason why the notion of service is of a considerable interest from both the industry and the academic research [1].

Web services are merging as a promising technology for the development of next generation distributed heterogeneous software systems [2]. Roughly, a Web service is a self-describing software component universally accessible supported by three technologies:

- Simple Object Access Protocol SOAP [3],
- Web Services Description Language WSDL [4],
- Universal Description, Discovery, and Integration UDDI [5].

The appearance of web services makes web application convenient through providing services on web [7]. There are momentous advances in the theory and technology of Web service in the last years. Besides the reuse of service itself, a promising way of developing a new service which can implement the extra function is through the orchestration of existing web services in which every of web services merely hold sub-function respectively. Web service sometimes requires combining more than one to meet our requirement [6]. This is so-called web services composition. A web service-oriented

system refers to a system that integrates multiple web services components [8] . That is to say, a single business transaction usually invokes a number of web services [9].

1.2 Motivation

Web service composition (WSC) offers an effective way to organize different Web services distributed in the networks in order to finish more complicated business tasks. However, with the increasing complexity and variety of business logics and network environments, Web services in a WSC might become unavailable due to some unexpected exceptions, which makes the whole WSC fail and brings a lot of loss to enterprises and users [10].

Web services composition is not always in an ideal environment. There are many potential problems. So the problem of web services composition is needed to be solved.

Web service composition involves the combination of a number of existing Web services to produce a more complex and useful service [11]. But different Web services are always written in different language and on different platform in distributed environment. The existing standard published by W3C, such as Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) and Universal Description, Discovery, and Integration (UDDI), can not standardize the merging of web services.

1.3 Statement of the Problem

A Formal Framework For Web Service Composition.

In this dissertation, I have made an attempt to implement the formal model for web service composition and add some features so that with the help of featured model one can verify the some web service composition properties. With following considerations:

- (i) Additional constraint on web service composition.
- (ii) Relationships between component web services
- (iii) Flow web services
- (iv) Composite service verification.
- (v) Some QoS parameters.

1.4 Organization of the Report

This dissertation report comprises of six chapters including this chapter that introduces the topic and states the problem. The rest of the report is organized as follows.

Chapter 2 gives the background of web service and web service related technologies such as SOAP, WSDL and UDDI. Here the web service composition is also explained.

Chapter 3 describes the basics of formal model for web service composition. Along with the overview of the proposed featured model of web service composition.

Chapter 4 gives the implementation details of the service both as a single web service as well as a composite web service.

Chapter 5 discusses the result and analysis part.

Chapter 6 concludes the dissertation work and gives suggestions for future work.

Chapter 2

Background and Literature Review

2.1 Web Services

Web services are software applications that can be *discovered*, *described*, and *accessed* based on XML and standard Web protocols over intranets, extranets, and the Internet. The beginning of that sentence, “Web services are software applications,” conveys a main point: Web services are software applications available on the Web that perform specific functions. Next, we will look at the middle of the definition where we write that Web services can be “discovered, described, and accessed based on XML and standard Web protocols.” Built on XML, a standard that is supported and accepted by thousands of vendors worldwide, Web services first focus on interoperability. XML is the syntax of messages, and Hypertext Transport Protocol (HTTP), the underlying protocol, is how applications send XML messages to Web services in order to communicate.

Web services technologies, such as Universal Description, Discovery, and Integration (UDDI) and ebXML registries, allow applications to dynamically discover information about Web services—the “discovered” part of our definition. The message syntax for a Web service is described in WSDL, the Web Service Definition Language. When most technologists think of Web services, they think of SOAP, the “accessed” part of our Web services definition. SOAP, developed as the Simple Object Access Protocol, is the XML-based message protocol (or API) for communicating with Web services.

The last part of our definition mentions that Web services are available “over intranets, extranets, and the Internet.” Not only can Web services be public, they can exist on an internal network for internal applications. Web services could be used between partnering organizations in a small B2B solution.

Figure 2.1 gives a graphical view of that definition, shown as layers. Relying on the foundation of XML for the technologies of Web services, and using HTTP as the underlying protocol, the world of Web services involves standard protocols to achieve the capabilities of access, description, and discovery.

Figure 2.2 shows these technologies in use in a common scenario:

In Step 1, the client application discovers information about Web Service A in a UDDI registry.

In Step 2, the client application gets the WSDL for Web Service A from the UDDI registry to determine Web Service A's API. Finally,

In Steps 3 and 4, the client application communicates with the Web service via SOAP, using the API found in Step 2. We'll get more into the details of these technologies later in the chapter.

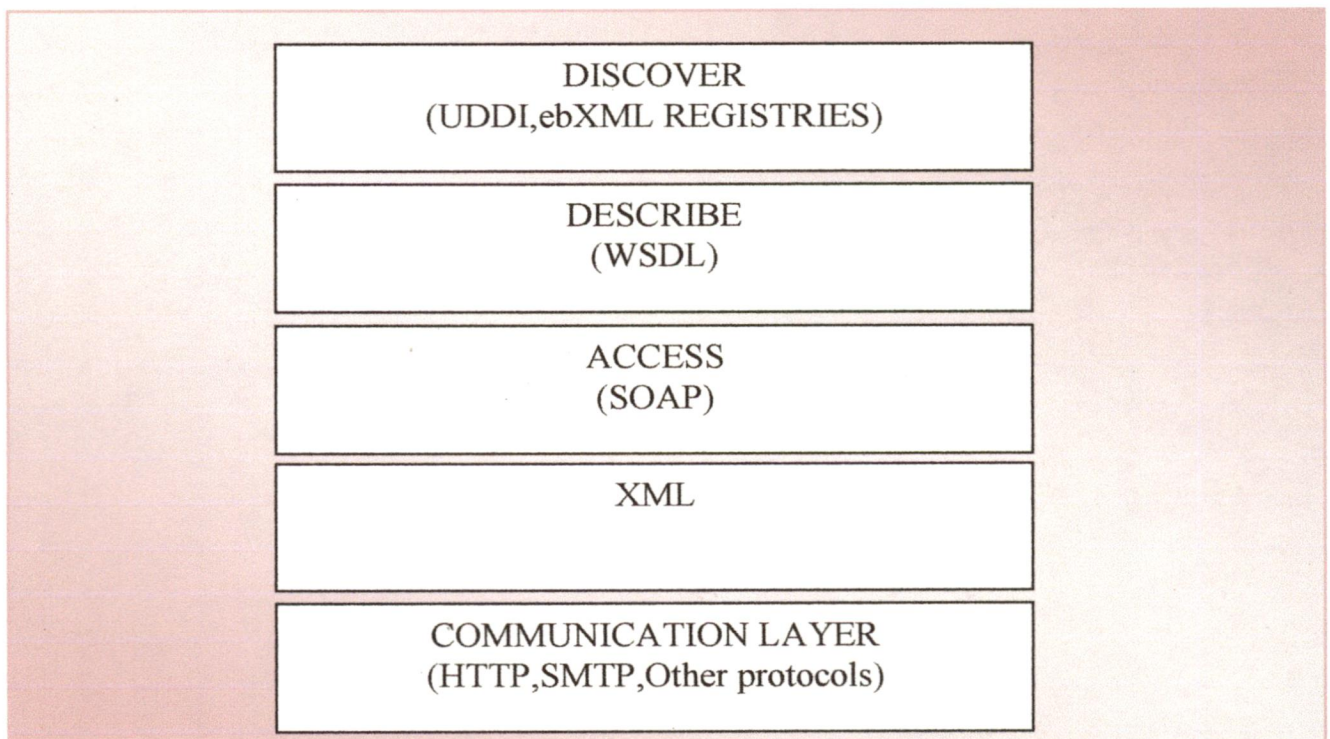


Figure 2.1 The basic layers of Web services.

This example scenario in Figure 2.2 shows the basics of client and Web service interaction. Because of these processes, such as discovery, the client application can automate interactions with Web services. Web services provide common standards for doing business and software integration—complementing a user-driven, manual navigation architecture to one where automated business process can be the focus.

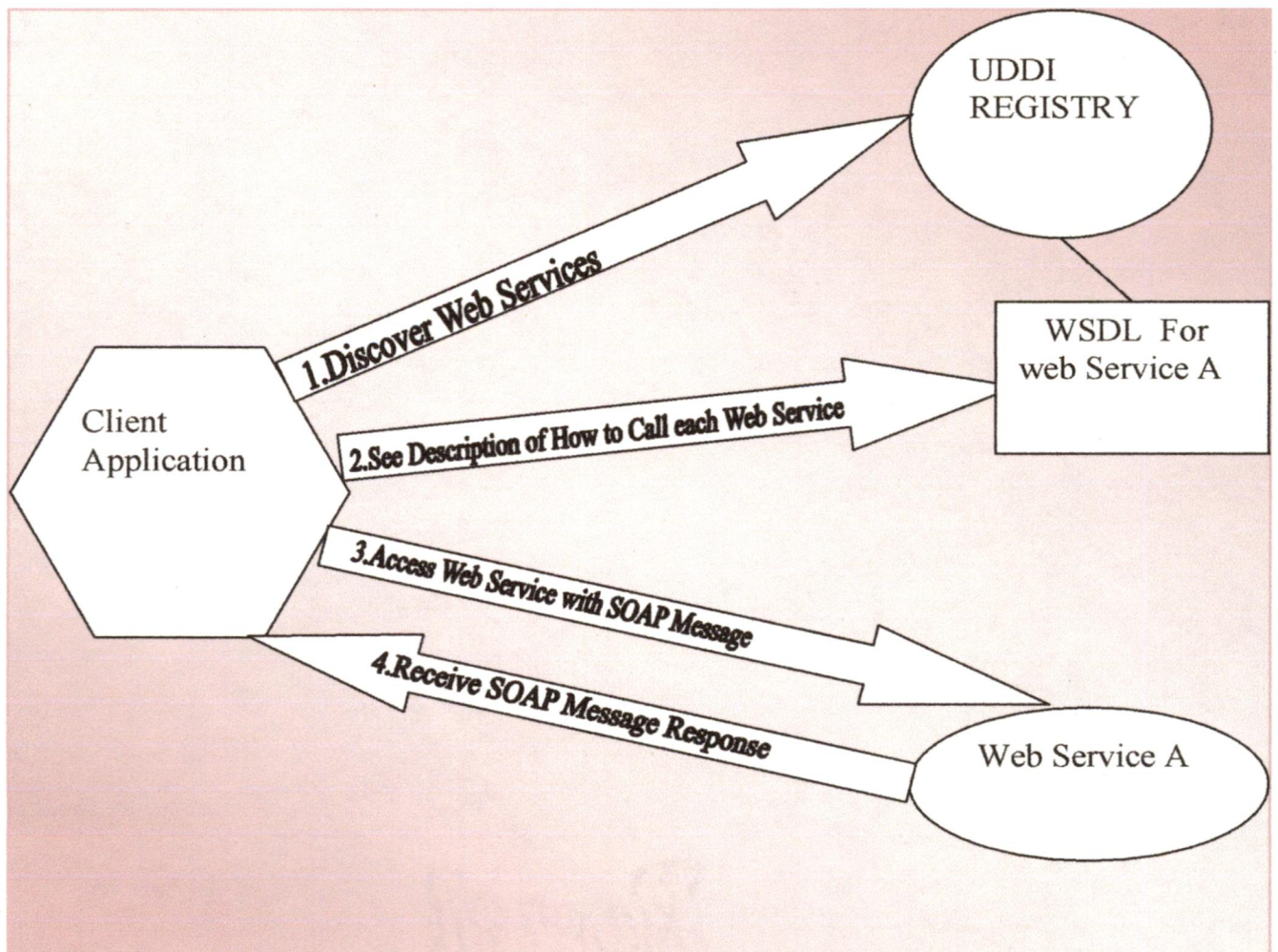


Figure2.2 A common scenario of Web services in use.

It is important to understand that Web services can be completely independent of the presentation, or the graphical user interface (GUI) of applications. Instead, Web services send data in XML format, and applications can add style and formatting when they receive the data. An example of a Web service could be a “driving directions finder” Web service that provides the capability to get text-based car directions from any address to any address, listing the driving distances and estimated driving times. The service itself usually provides no graphics; simply speaking XML messages to a client application. Any application, such as a program created in UNIX, a Microsoft Windows application, a Java applet, or server-side Web page, can take the information received from that application and style it to provide graphics and presentation.. Separating business logic from presentation is commonly known in software engineering as the Model-View-Controller (MVC) paradigm.

Web services support this paradigm. Shown in Figure 2.3, the user interface details (the view) and business logic (the model) are separated in two different components, while the component layer between them (the controller) facilitates communication.

Because the presentation is separate, the client application can present the information to the user in many different ways. This is an important concept because many browsers make it easier for you by offloading this processing with style sheets, using XSL Transformations (XSLT) [12].

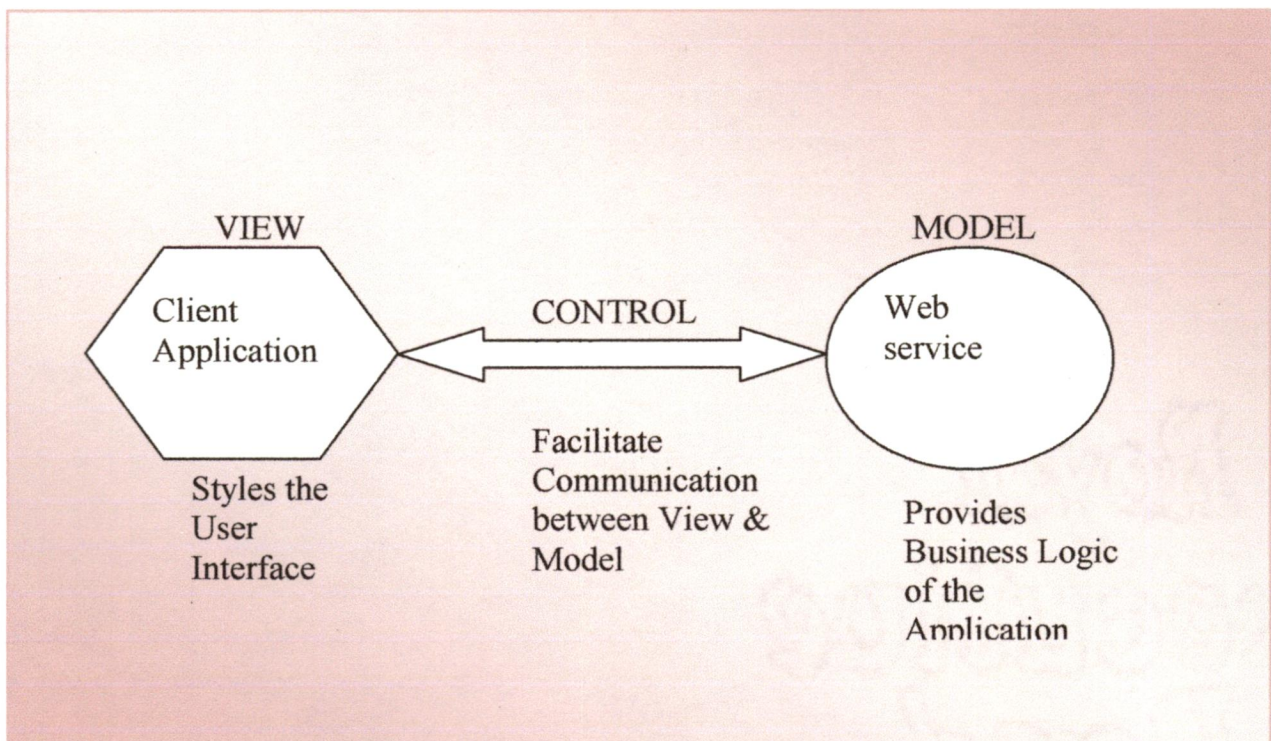


Figure2.3 The Model-View-Controller paradigm

2.2 Understanding the Basics of Web Services:

This section gives a high-level overview of some of the basic Web services technologies. In this section, we discuss the following concepts that are fundamental in understanding Web services [13]:

- Simple Object Access Protocol(SOAP)
- Web Service Description Language(WSDL)
- Discovering Web Service(UDDI)

2.2.1 Simple Object Access Protocol (SOAP):

SOAP is the envelope syntax for sending and receiving XML messages with Web services. That is, SOAP is the “envelope” that packages the XML messages that are sent over HTTP between clients and Web services. As defined by the W3C, SOAP is “a lightweight protocol for exchange of information in a decentralized, distributed environment.”

It provides a standard language for tying applications and services together. An application sends a SOAP request to a Web service, and the Web service returns the response in something called a SOAP response. SOAP can potentially be used in combination with a variety of other protocols, but in practice, it is used with HTTP.

The syntax of SOAP, in its basic form, is fairly simple, as shown in Figure 2.4. A SOAP message contains the following elements:

- A SOAP envelope that wraps the message
- A description of how data is encoded
- A SOAP body that contains the application-specific message that the backend application will understand.

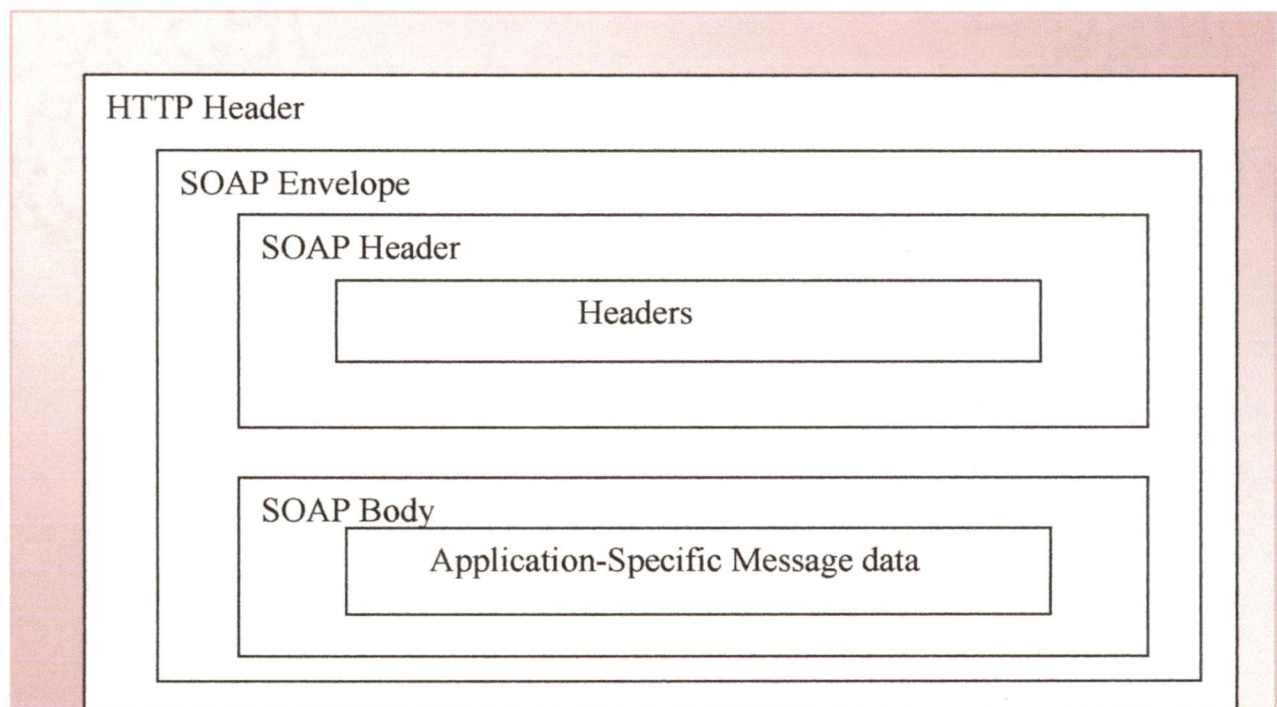


Figure 2.4 Structure of a Web-based SOAP message.

2.2.2 Web Service Definition Language (WSDL):

Whereas SOAP is the communication language of Web services, Web Service Definition Language (WSDL) is the way we describe the communication details and the application-specific messages that can be sent in SOAP. WSDL, like SOAP, is an XML grammar. The W3C defines WSDL as “an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.” To know how to send messages to a particular Web service, an application can look at the WSDL and dynamically construct SOAP messages. WSDL describes the operational information—where the service is located, what the service does, and how to talk to (or invoke) the service. When we create a Web service from our enterprise applications, most toolkits create WSDL for us. Figure 2.5 shows an example of how this process works.

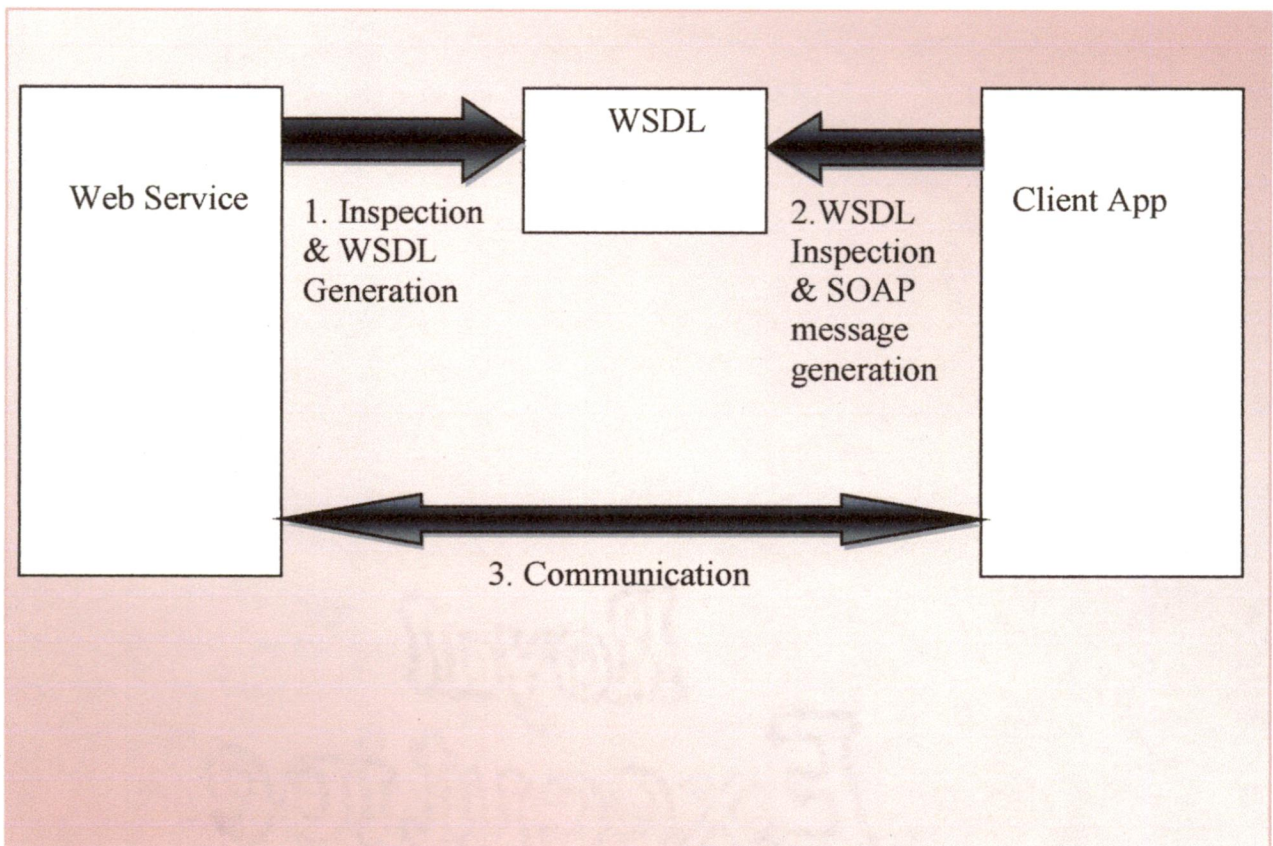


Figure2.5 Dynamic communication by inspecting WSDL.

2.2.3 Discovering Web Services:

Finding Web services based on what they provide introduces two key registry technologies:

UDDI (Universal Description, Discovery, and Integration) and ebXML registries. Both of these technologies are worth discussing, and while they may seem to be competing technologies, it is possible that they may complement each other in the evolution of Web services.

Universal Description, Discovery, and Integration (UDDI):

Universal Description, Discovery, and Integration is an evolving technology and is not yet a standard, but it is being implemented and embraced by major vendors. Simply put, UDDI is a phone book for Web services. Organizations can register public information about their Web services and types of services with UDDI, and applications can view information about these Web services with UDDI. The information provided in a UDDI business registration consists of three components: white pages of company contact information, yellow pages that categorize businesses by standard taxonomies and green pages that document the technical information about services that are exposed. Figure 2.6 demonstrates this concept.

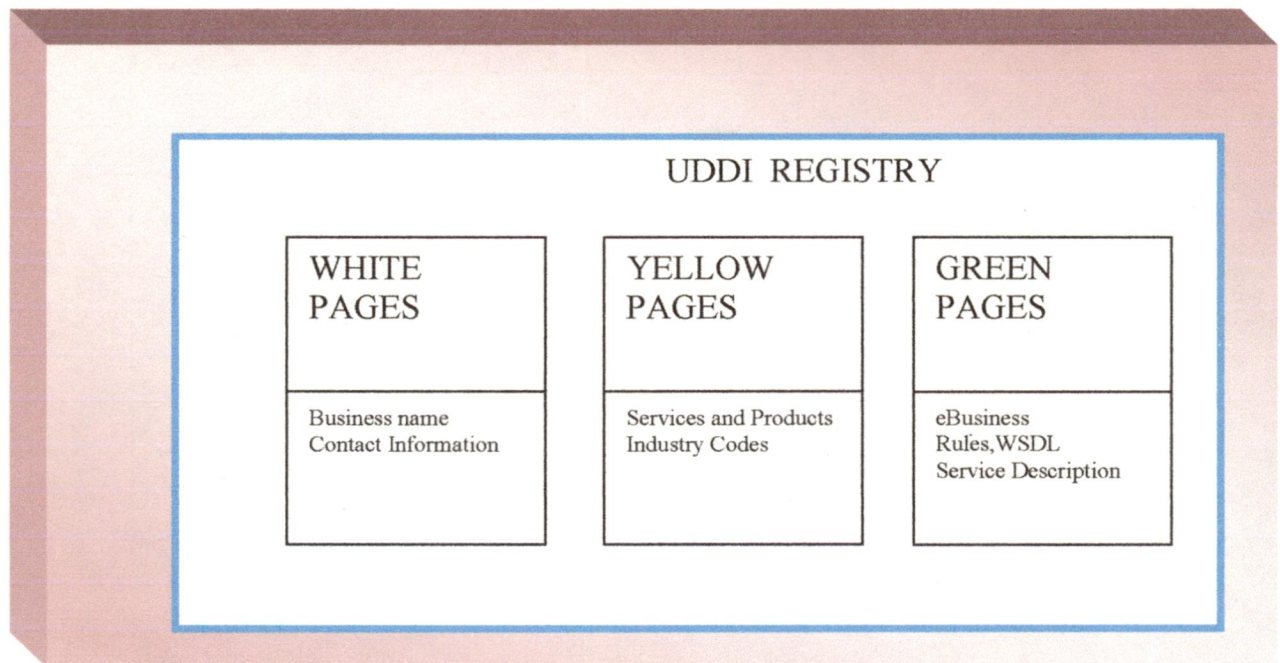


Figure 2.6 A UDDI Registry as a conceptual phone book.

2.2.4 More about WSDL :

WSDL is a specification 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 message requests and message responses
- Binding information about the transport protocol to be used
- Address information for locating the specified service

Using WSDL, a client can locate a web service and invoke any of its publicly available functions [14].

The WSDL Specification:

WSDL is an XML grammar for describing web services. The specification itself is divided into six major elements:

Definitions:

The definitions element must be the root element of all WSDL documents. It defines the name of the web service, declares multiple namespaces used throughout the remainder of the document, and contains all the service elements described here.

Types:

The types element describes all the data types used between the client and server. WSDL is not tied exclusively to a specific typing system, but it uses the W3C XML Schema specification as its default choice. If the service uses only XML Schema built-in simple types, such as strings and integers, the types element is not required.

Message:

The message element describes a one-way message, whether it is a single message request or a single message response. It defines the name of the message and contains zero or more message part elements, which can refer to message parameters or message return values.

PortType:

The portType element combines multiple message elements to form a complete one-way or round-trip operation. For example, a portType can combine one request and one

response message into a single request/response operation, most commonly used in SOAP services. Note that a portType can (and frequently does) define multiple operations.

Binding:

The binding element describes the concrete specifics of how the service will be implemented on the wire. WSDL includes built-in extensions for defining SOAP services, and SOAP-specific information therefore goes here.

Service:

The service element defines the address for invoking the specified service. Most commonly, this includes a URL for invoking the SOAP service.

To keep the meaning of each element clear, Figure 2.7 shows a concise representation of the WSDL specification.

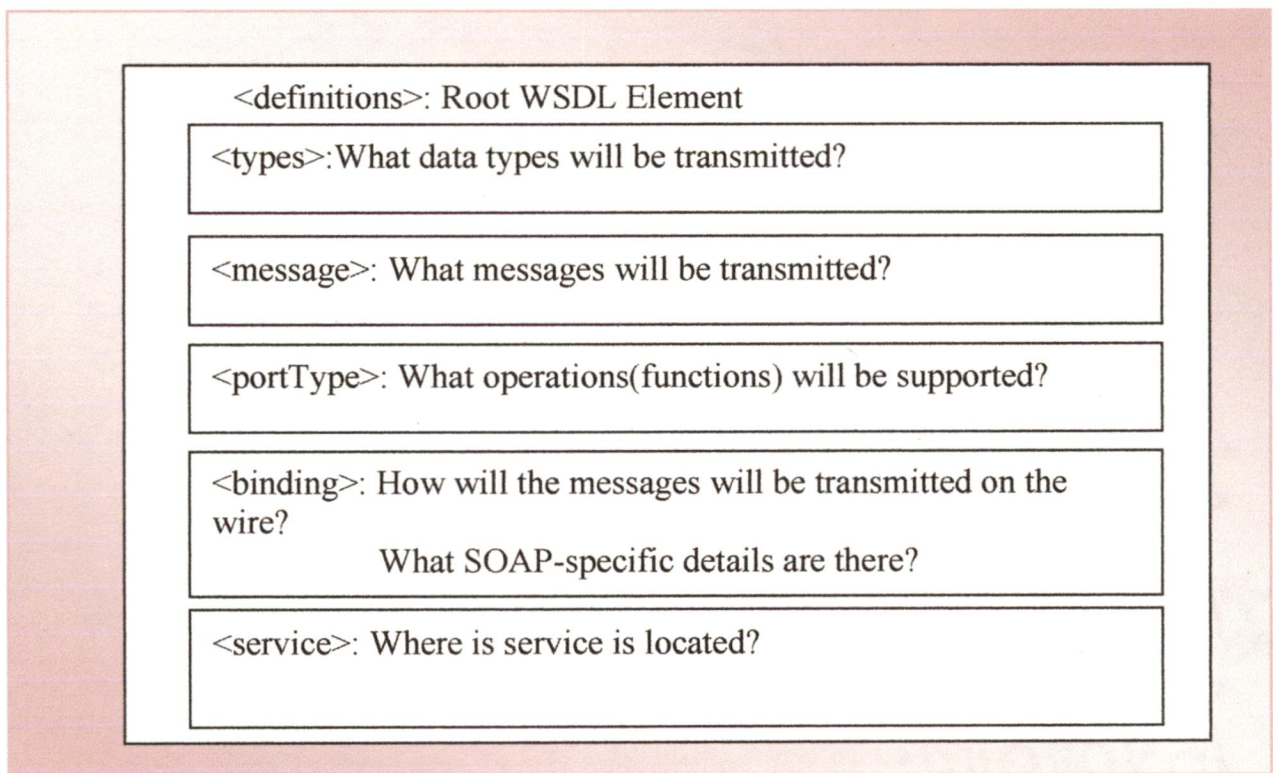


Figure2.7 The WSDL Specification in a nutshell

2.3 Web Service Composition:

Web service composition as an important value-added function provides an application foundation for reusing services and automating composition. Service composition may be defined from different perspectives and aspects.

From the perspective of structure and technology, service composition is a technique, by which relatively simple services may be composed as more complex ones. The perspective of dynamic process stresses that service composition is a process integrating dynamic discovery, composition and executing existing services with a certain order for creating a new service. From the perspective of work flow, service composition is defined that web services provided by different enterprises are linked each other for certain business goals through an apparent process model. From the perspective of enterprise functions, web services composition will integrate some basic services obtained from different enterprises to provide a value-added service [15].

Chapter 3

Formal Model for Web Service Composition

3.1 The Formal Model:

In this model [16], web services are classified into single service and composite service, which are defined informally as follows respectively:

Single Service: It is an independent service entity that can be invoked by other services, and it realizes its service functionality by calling other services to provide service support.

Composite service: It is a service which is the composition of many single services so as to fulfill more complicate service functionality. The single service that doesn't depend on other services is called *atomic service*; it provides its independent functionality without calling other services. Composite service depends on either single services or other composite services.

Single service model is the basis of composite service model; A new single service model is introduced and depicted in Fig 3.1.

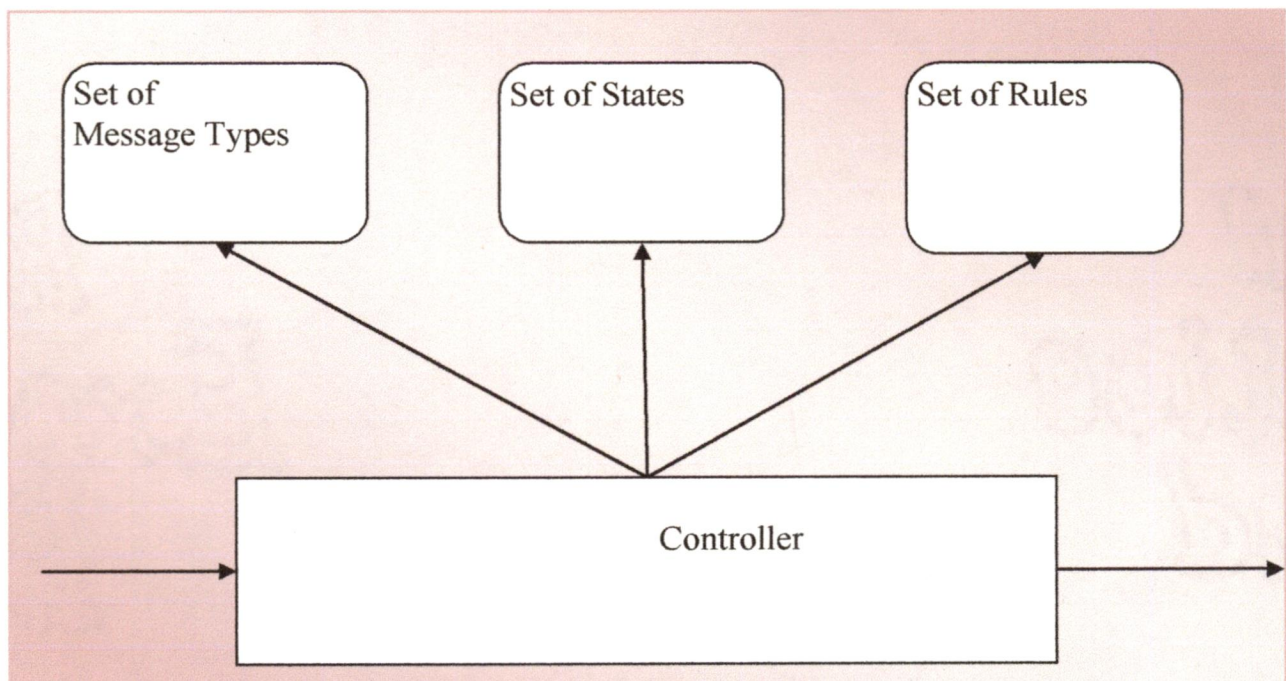


Fig.3.1 A kind of single service model.

The model is composed of four parts:

(i) *Set of message types*: it includes all message types that can be received and sent by the single service.

(ii) *Set of states*: there is a corresponding state element in this set for each message type. A state element may have multiple different attributes, each attribute value represents an attribute state of message type itself, where attribute value *received* denotes that whether or not a message has been received, if *received*=true, the message has been received by the service, otherwise, it hasn't been received; attribute value *sent* denotes whether or not a message has been sent, and attribute value *ready* denotes whether or not a service is ready to receive a message, etc.

(iii) *Set of rules*: it denotes the relationship among all kinds of message types, including the relationship between *request* and *response*, *input request* and *output request*, etc.

(iv) *Controller*: it takes charge of the control logic for receiving and sending messages, and further takes next action and modifies the data in relevant set by judging related data information. Controller is an execution mechanism in a single service.

According to this model, the basic processing procedure of service is as follows: firstly, the controller is ready to receive a message and judge whether or not the message is a type that can be received? If the answer is positive, receive the message and mark it with *received*. In the meanwhile, the controller decides what action should be taken next step according to the state of the received message. If the value of state is *ready*, next action is taken according to the set of rules.

3.1.1 Message Type:

Based on the direction of message being passed, the message types are classified into *input message* and *output message*. Input message denotes the message that can be received by a single service, and output message denotes the message that can be sent by a single service. Based on the characteristics of message, messages are classified into *request message* and *response message*. *Request message* has two types, one is *response-required request message* that needs a response, the message waits and receives a response message after its instance was created; the other is *no-response-required request message* without requiring a response, the message needn't wait a response after its

instance was created. For a response-required request message, there must be a response message type against it. In order to keep up consistency for processing, we introduce an empty message type Φ as the response type for a no-response-required request message, i.e., a service needs to receive a Φ message as a response after it sent a no-response-required request message.

Meanwhile, to differentiate all message types in a single service, different no-response-required request message types should be corresponded to different empty message types.

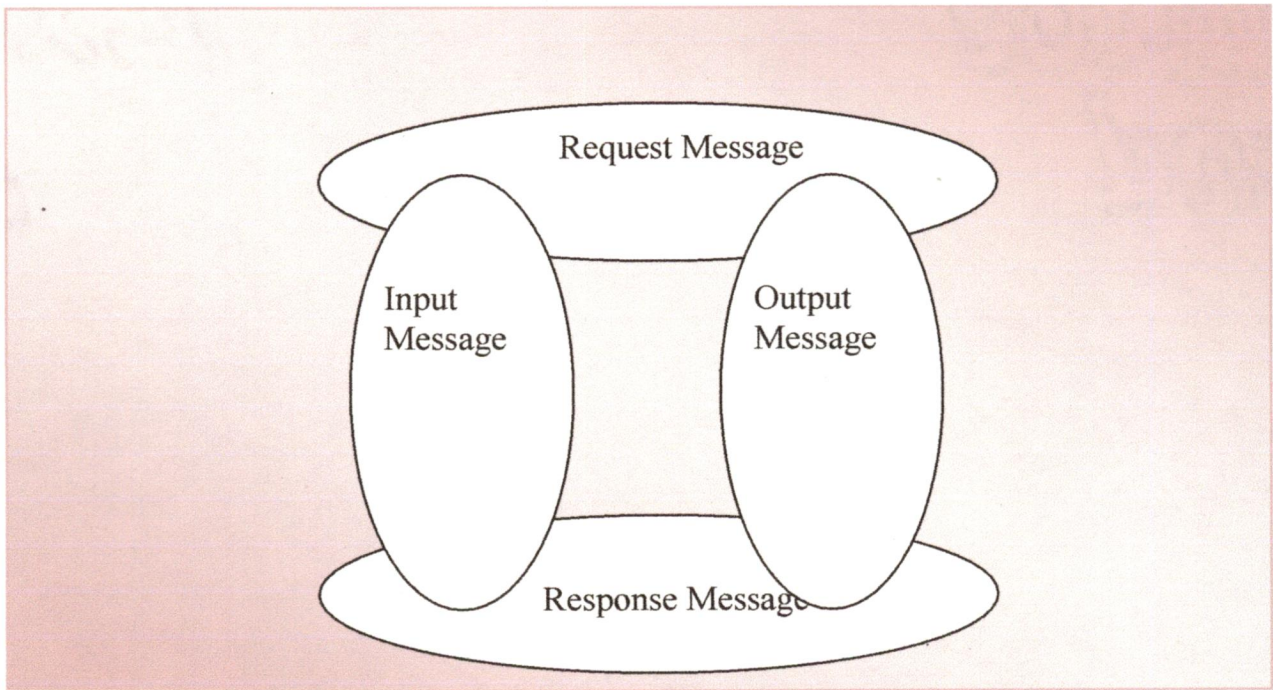


Fig.3.2 The classification of message types.

3.1.2 For Single Service:

Now, we can describe the formal model of a single service as follows:

Single service model: A single service model S is a quintuple: $S=(\Sigma,\delta,\Delta,P,F)$ where:

- $\Sigma = (M_1, M_2, \dots, M_n)$ is the set of message types received or sent by the service, each M_i denotes a message type. We have $n=|\Sigma|$ and for $\forall M_i, M_j \in \Sigma$ if $i \neq j$ then $M_i \neq M_j$. Further we have $\Sigma^{\text{in}}, \Sigma^{\text{out}}, \Sigma^{\text{req}}, \Sigma^{\text{res}}, \Sigma^{\text{res_in}}, \Sigma^{\text{res_out}}, \Sigma^{\text{req_in}}$ and $\Sigma^{\text{req_out}}$ obviously, they all are the subsets of Σ .
- $\delta: \Sigma^{\text{req}} \rightarrow \Sigma^{\text{res}}$, it defines the corresponding relationship between request message type and response message type.

- $\Delta : \Sigma^{\text{req_in}} \rightarrow \Sigma^{\text{res_out}} \cup \Phi$, it defines the mapping relationship between a request-input message type and a request-output message type. Empty set Φ denotes that a request-input message needn't to correspond a request-output message.
- $P = (p_1, p_2, \dots, p_n)$ is the set of states of all corresponding message types. $n = |\Sigma|$, p_i denotes the state information of message type M_i and it is a complex data type. Following two cases should be differentiated when we deal with p_i : (1) if $M_i \in \Sigma^{\text{in}}$, p_i has two types of attribute variables, thereinto variable *ready* denotes whether or not it is ready to receive message type M_i , while the variable *received* denotes whether or not the message type M_i has been received. (2) If $M_i \in \Sigma^{\text{out}}$ has one attribute variable *sent*, it denotes whether or not it has sent a message type M_i . If we use '=' operator to obtain a corresponding attribute value $p_i.\text{ready}=\text{true}$ denotes that the service is ready to interact with other service with message M_i , otherwise, $p_i.\text{ready}=\text{false}$ denotes that the service is not ready to interact with others with message M_i . The sufficiency and necessary condition for a message M_i to start to execute is that: $p_i.\text{ready}=\text{true}$ and $p_i.\text{received}=\text{true}$ hold simultaneously, i.e., a service is ready to receive a message and at the same time the message has been received.
- F is the computation controller that completes the processing based on the received message and the data in relevant set. For single service S if: $\Sigma^{\text{in}} = \Sigma^{\text{req}}$ and $\Sigma^{\text{out}} = \Sigma^{\text{res}}$ then the service is an atomic service that can be called by other services, but it provides functionalities without depending on other services.

3.1.3 For Web Services Composition:

A single service can't offer a complicated functionality; the collaboration of multiple services is needed. A service can call other services, many services can collaborate to complete complicated functionality by interactive call between them, therefore, there is a call relationship between some services.

Basic Call Relationship: Let S_1 and S_2 be two services, both Σ_1 and Σ_2 denote the set of messages of two services, respectively. If there is a message type M , satisfying: M

$\in \Sigma_1^{\text{req_out}}$ and $M \in \Sigma_2^{\text{req_in}}$ there is a call relationship between S_1 and S_2 marked as $S_1 \rightarrow S_2$.

Interactive logic Relationships:

Here, we only discuss the interactive logic among *request-output* messages that includes four basic types, i.e., *sequence*, *selection*, *loop* and *parallel*, which can be defined as follows respectively:

[*Sequence •*]: Let $M_1, M_2, \dots, M_n \in \Sigma^{\text{req_out}}$, if M_i must be executed after the *execution end* of M_{i-1} there exists sequence relationship between M_i and M_{i-1} furthermore, there exists sequence relationship among M_1, M_2, \dots, M_n marked as: $M_1 \bullet M_2 \bullet \dots \bullet M_n$.

[*Selection |*]: Let $M_1, M_2, \dots, M_n \in \Sigma^{\text{req_out}}$ If there is one and only one of them be selected at a time based on different conditions, there is a selection relationship among M_1, M_2, \dots, M_n marked as $M_1 | M_2 | \dots | M_n$.

[*Loop **]: Let $M \in \Sigma^{\text{req_out}}$ If a condition is satisfied, M is executed repeatedly till the condition is not satisfied, M has a loop relationship, marked as M^* .

[*Parallel ||*]: Let $M_1, M_2, \dots, M_n \in \Sigma^{\text{req_out}}$ Starting from a time point, if these messages can start to execute simultaneously and their execution time can be overlapped, there is a parallel relationship among M_1, M_2, \dots, M_n marked as $M_1 || M_2 || \dots || M_n$.

Now we describe, the service composition model(or SM) based on a kind of service composition pattern (or SC), therefore we define the SC as follow:

Service Composition Pattern: A SC is triple (Q,T,R) , where:

- $Q = \{ S_1, S_2, S_3, \dots, S_n \}$ is a set of n services that are used to compose a new composite service;
- $T: \Sigma_1^{\text{out}} \cup \Sigma_2^{\text{out}} \cup \dots \cup \Sigma_n^{\text{out}} \rightarrow \Sigma_1^{\text{in}} \cup \Sigma_2^{\text{in}} \cup \dots \cup \Sigma_n^{\text{in}}$ is the set of all call relationships between services, which denote the relationships between output messages and input message of different services.
- R : the set of interactive logic relationships between messages.

If we regard all services composed together as a service, concerning the outside interface regardless of internal structure, the whole composite service can be regarded as a single

service, it has the properties of single service and can take part in other services composing process. Composition pattern reflects the calling and interactive logic relationships between services composed.

3.2 Features included in the Formal Model:

The existing formal model for web service composition may be featured with respect to following dimensions.

3.2.1 Activeness Constraints:

In the process of web service composition, we must check whether the component web services are active or not. There may be various reasons due to which a component web service may not be active such as:

- The component web service is itself has some problems.
- The component web service is engaged with some other business transaction.

It is difficult to judge whether web services is active. So a tuple L is added to describe web services' status so a web service with status is denoted as : $S=(\Sigma,\delta,\Delta,P,F,L),L:\{0,1\}$

Constraint 1: Web service can be used only when it is active.

Constraint 2: A web services can be used only when has finished last composition. When the composition is started web services should be locked.

Constraint 3: Composition of a set of web services can be possible only when all the web services in the set are active at the time of composition.

3.2.2 Relationships between component web services:

Let service S_i and service S_j be sub-services of the composite service S while service S_i provides a different type of service from service S_j . The relationship R between sub-services S_i and S_j can be identified as follows[17]:

Independent Relationship:

Each sub-service is freely independent of the other. The order of execution of these two sub-services does not affect the composition service, which means that the result is the same in either case.

Prerequisite Relationship:

The prerequisite relationship means that one service has to finish before the other starts. Service S_i has to finish before service S_j starts.

Parallel-Prerequisite Relationship:

Service S_i executes at the same time as service S_j but service S_j has to wait for the result from service S_i before completing its process. This relationship differs from prerequisite in respect of the time which the service processes must start.

Parallel-Dependency Relationship:

Service S_i and service S_j process or execute in parallel (simultaneously) but the results of each service need to be compromised with the other. This kind of relationship needs negotiation and deadlock-free mechanisms.

Substitute Relationship:

Service S_i can be substituted by service S_j . The service S_i and S_j seem to provide the same service but they have some different attributes.

3.2.3 The Flow web service:

In a business transaction, involved in the process of web service composition, an intermediate web service is said to be “flow web service” if it satisfies the following conditions:

- (i) It takes at least one request input message from a web service ws_k (from set of web services involved in the transaction) and
- (ii) Corresponding to that request input message, it triggers at least one request output message to other web service(s) (other than ws_k) with out responding to the web service ws_k and
- (iii) (i) and (ii) are satisfied for at least one request input message that the web service can accepts in that business transaction.

These web services behaves like a pipe in the process of web service composition so can also be named as pipe web services.

Properties of the flow services:

- A web service may behave like a flow web service in a business transaction while in another business transaction it doesn't.
- If the point (iii) in the definition is satisfied for all request input messages that web service can accept in a particular business transaction, then the service is called pure flow web service otherwise it is called partial.

Flow terminating web service:

In a chain of flow web services a web service is said to be “flow terminating web service” if it satisfies the following:

- (i) it takes at least one “request input message” from a web service ws_k (from the set of web services involved in flow chain) and
- (ii) Corresponding to that request input message, it triggers at least one “response output message” to the web service which originates the chain of the flow web services.
- (iii) (i) and (ii) are satisfied for at least one “request input message”.

3.2.4 Verification by using Pi-Calculus:

The following framework shown in Figure 3.3 can be used to verify the correctness of composite web service with the help of Pi-calculus:

The Pi-calculus[19]:

The Pi-calculus is a concurrency theory proposed by Robin Milner to research communication between processes, whose basis is CCS (Calculus of Communication System). The basic elements in pi-calculus are process and name, where process denotes concurrent entities and the communication between processes is done by transferring names. Name stands for variables, identifiers and channels.

A composite service is a kind of concurrency system where atomic service communicates with each other by sending and receiving messages.

A process t can be defined as follows:

$$t ::= 0 \mid c \langle x \rangle .P \mid \bar{c} \langle x \rangle .P \mid \tau.P \mid P + Q \mid P \mid Q \mid (vc)P \mid \text{if } x = y \text{ then } P.$$

- 0 : a null process, which does not execute any operation and can also be expressed as NIL;
- $\bar{c} \langle x \rangle .P$: denotes to send x from path c and then execute process P . In pi-calculus, “ $P. Q$ ” denotes the sequential execution of process P and Q ;
- $c \langle x \rangle .P$: denotes to receive x from path c and then execute process P ;

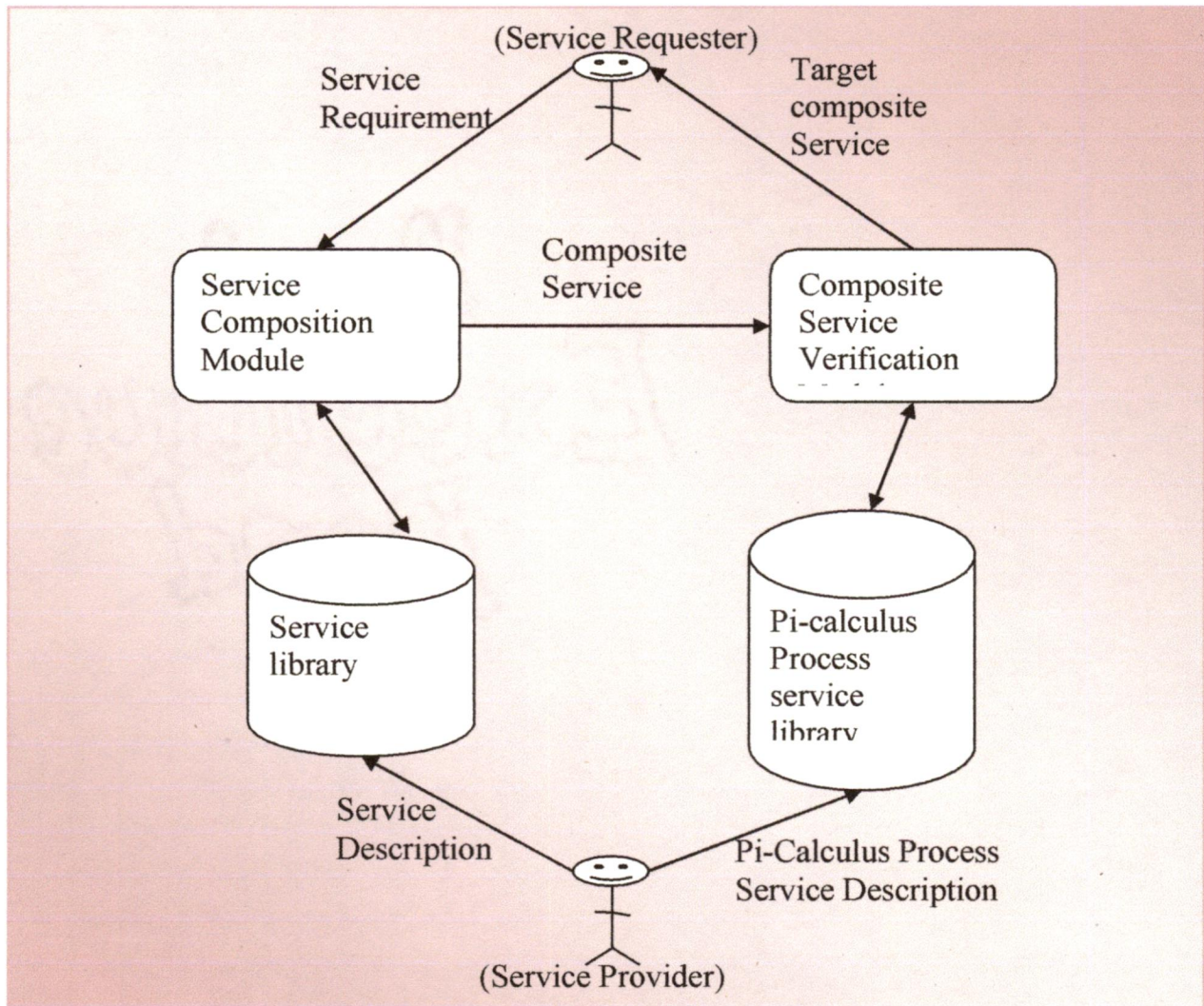


Figure 3.3 Verification Framework Using Pi-Calculus

- $\tau.P$: denotes to execute process P directly. τ is a dummy action, which do nothing;
- $P + Q$: denotes to select one process from P and Q to be executed;
- $P | Q$: denotes that P and Q are concurrently executed. P and Q can exchange messages through path.
- $(\nu c)P$: denotes that P can not communicate with environment through path c , but communication through path c can go on inside P ;
- $\text{if } x = y \text{ then } P$: denotes that if name x is equal to y , then execute process P .

Composite service verification using Pi-calculus:

The equivalent pi-calculus description of a web service description is as follows:

Let the service described in WSDL is shown as follows[20]:

```
<service name = "s">  
  <input message = "m"/>  
  <output message = "n"/>  
</service>
```

So it can be expressed as pi-calculus process $a\langle m \rangle \bar{a}\langle n \rangle$, which means that input message m through channel a and then send message n through channel a .

3.2.5 QoS parameters:

For Single Service:

To meet the quality requirement on composite web services of users, it needs to choose component services from the several ones that have similar functionality according to QoS metrics. The main QoS metrics of Web service is the QoS metric set defined by W3C [21], including performance, reliability, and robustness etc. Among all these metrics, response time, reliability and availability are most concerned by users. Besides, cost (i.e. price of the Web services) and reputation (i.e. the evaluation of the Web services by the users) are also the important factors to consider when composing Web services. Therefore, one can focus on these four metrics when selecting component web services [22].

Response time $q_{rsp}(ws)$: The time required to complete a Web service request between service consumer and provider, denoted as $q_{rsp}(ws) = T_d(ws) - T_i(ws)$, where $T_d(ws)$ is the timestamp when the service ws is delivered and $T_i(ws)$ is the timestamp when the service ws is invoked.

Cost $q_{cst}(ws)$: The fee paid by service consumer for using Web service ws to service provider.

Reputation $q_{rpu}(ws)$: The evaluation by service consumers after using the service ws . Reputation is always the statistical average of the service consumers' evaluation calculated as $q_{rpu}(ws) = \sum_{i=1}^n \text{Rank } i / n$ where $\text{Rank } i$ is the feedback rank given by service

consumers after using the service w_s which is a value between 0 and 1 (the larger the value, the higher the reputation) and n is the statistical times.

Availability $q_{alb}(ws)$: The probability that service w_s will be available, denoted as $q_{alb}(ws) = T_{alb}(ws) / T_{total}(ws)$, where $T_{total}(ws)$ is the total test time and $T_{alb}(ws)$ is the time in $()$ total Tws that service w_s is available.

For Composite service:

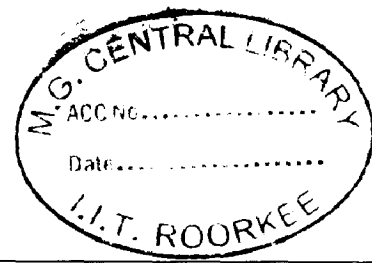
There are multiple ways to measure the performance of a system. The most commonly used performance metrics are response time (R) and throughput (X) [23].

Response time: To a composite web service, the response time is defined as the time interval from a request arriving at the service to the instant the corresponding reply begins to appear at the requestor's terminal, and it is determined by two factors: the quality of network transmission, and the processing capacity of the service. Here, we only consider the processing capacity of the service. The quality of network is considered to be very wide and difficult to analysis.

Throughput: The throughput is generally considered as a measure of the service's productivity, that is, the number of requests served successfully during the measurement period.

Relation with Queuing network model [18]:

The use of queuing network models for evaluating the performance of composite Web services is justified by many reasons. It is straightforward to map the request behavior of a Web Service into a queuing network. Web services are modeled by service centers and the requestors are modeled by customers. Another important reason is that queuing network models have a good balance between a relative high accuracy in the performance results and the efficiency in model analysis and evaluation. For a composite Web service, the queuing network model can be seen as some interconnected queuing systems for single Web services. The interconnections between sub-services form the topology of the queuing network. The topology of a queuing network shows the relationships between the services and the movement of the requests among them.



Chapter 4

Implementation

4.1 Implementation of the service as a single web service:

In this section we describe the implementation details of the service as a single web service. The following software tools are needed for implementation.

- Java SE 6.
- Eclipse IDE for java developers.

Java SE 6:

The following are various new features of Java SE 6[24]:

- Performance enhancements. Running a Java 5 app on Java 6 even without recompilation will run faster.
- Pluggable Annotation Processing API.
- Common Annotations.
- Java API for XML Based Web Services - 2.0
- Web Services Metadata.
- Streaming API for XML.
- XML Digital Signature.

Eclipse IDE:

The Eclipse IDE for Java EE Developers contains everything we need to build Java and Java Enterprise Edition (Java EE) applications. Considered by many to be the best Java development tool available, the Eclipse IDE for Java EE Developers provides superior Java editing with incremental compilation, Java EE 5 support, a graphical HTML/JSP/JSF editor, database management tools, and support for most popular application servers[25].

After successful installation of Java SE 6 , we configure Eclipse IDE to use Java SE 6 installed earlier.

Configuring Eclipse IDE:

- We select Window > Preferences > Java > Installed JREs, and click the Add button.
- We then enter a name, such as Java SE 6, to easily identify what version it is.
- Click the Browse button and locate the directory where JRE 6 was installed.
- Click OK.
- Select the Java SE 6 check box and then click OK.

After configuring Eclipse IDE the following steps are executed in order to create the service as single web service:

Step1: Create a project

- We select File > New > Project.
- Then we expand the Java folder and click Java Project.
- Click Next.
- Enter a project name, such as wsVijay3Example, when prompted.
- We select the Use default JRE radio button if it was previously selected by default; otherwise we select the Use a project specific JRE radio button, ensuring that it's Java SE 6.
- Click Finish to associate our project with the Java JDK we installed earlier.
- If we're prompted to switch Java perspective, then we click Yes.

Step2: Create the Server:

- We select File > New > Package.
- When the New Java Package window opens, we enter a name for the package, such as com.myfirst.wsServer.

Right-click the package name we just created, then select New > Class.

Configuring it as shown in Figure 4.1.

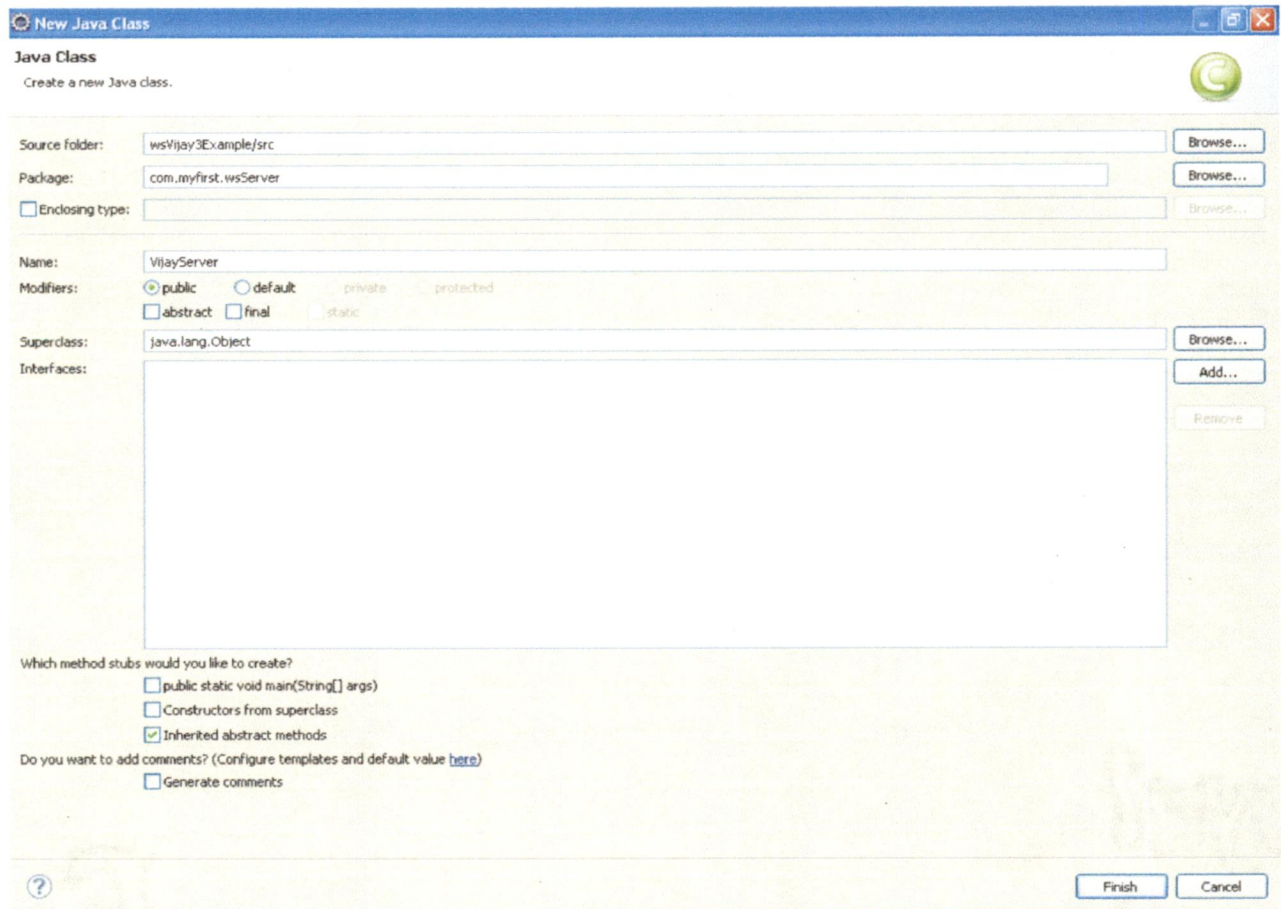


Figure 4.1 Adding class to the package

We create our class as public with no main method stub. Now that we have provided our package with a class, we can write the code for the server, as shown in Appendix A.

Step3: Generate the Server code with ant:

- We right-click the project and select New > File.
- And then enter the name build.xml when prompted, then click Finish.
- We open this file with the Ant Editor by right-clicking it and selecting Open With > Ant Editor. From now on, whenever we double-click this file, it opens with the Ant Editor.
- Enter the Ant script shown in Appendix B.1.
- To run the Ant build.xml file, we right-click Run As and select Ant Build, which executes the Ant file.
- Here we make sure that this results in a BUILD SUCCESSFUL message in the Eclipse Console window, as shown in Figure 4.2.

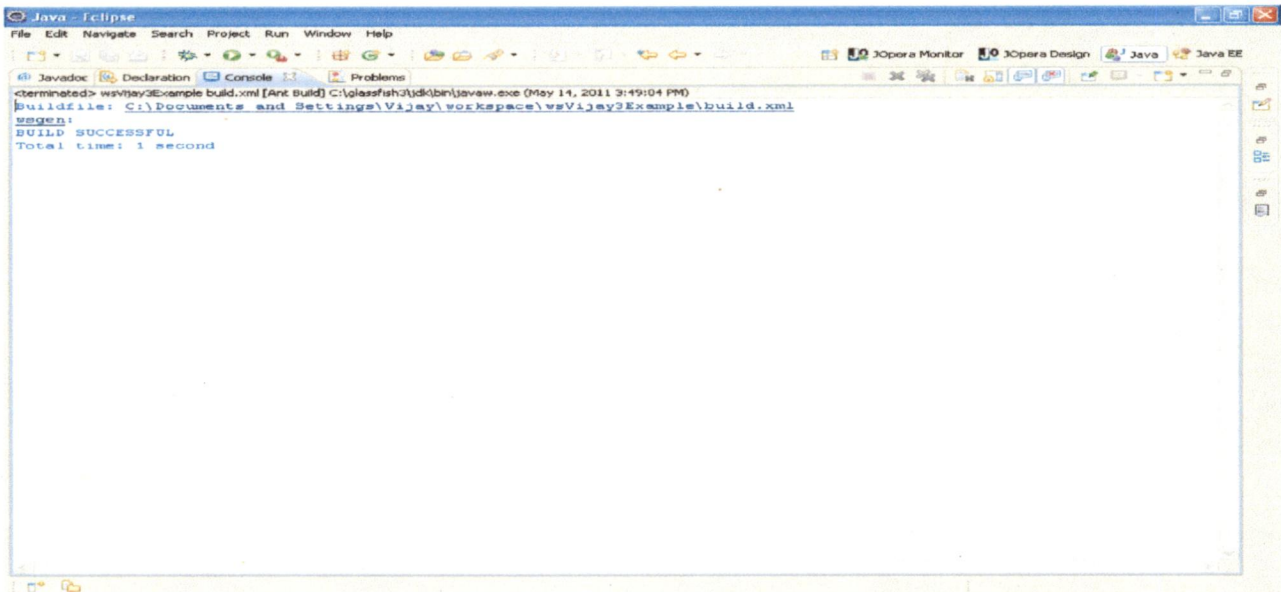


Figure 4.2 Build Successful message

- We return to the Eclipse project and refresh the project by right-clicking wsVijay3Example and selecting Refresh. We now see the generated code to run the web service created under the new package called com.myfirst.wsServer.jaxws.

Step4: Publish the web service:

After we have generated the code for the web service's server, we need to publish it so we can start using it:

- We create a new class under the com.myfirst.wsServer package we created, and call it something like RunService.
- Right-click the package and select New > Class, but this time select the option to create the main method stub.
- We write the code to publish our web service, as shown in Appendix B.2.
- Run this class by right-clicking it and selecting Run As > Java Application.

The Eclipse IDE Console window should display. We see an indication that the web server has started, as shown in Figure4.3

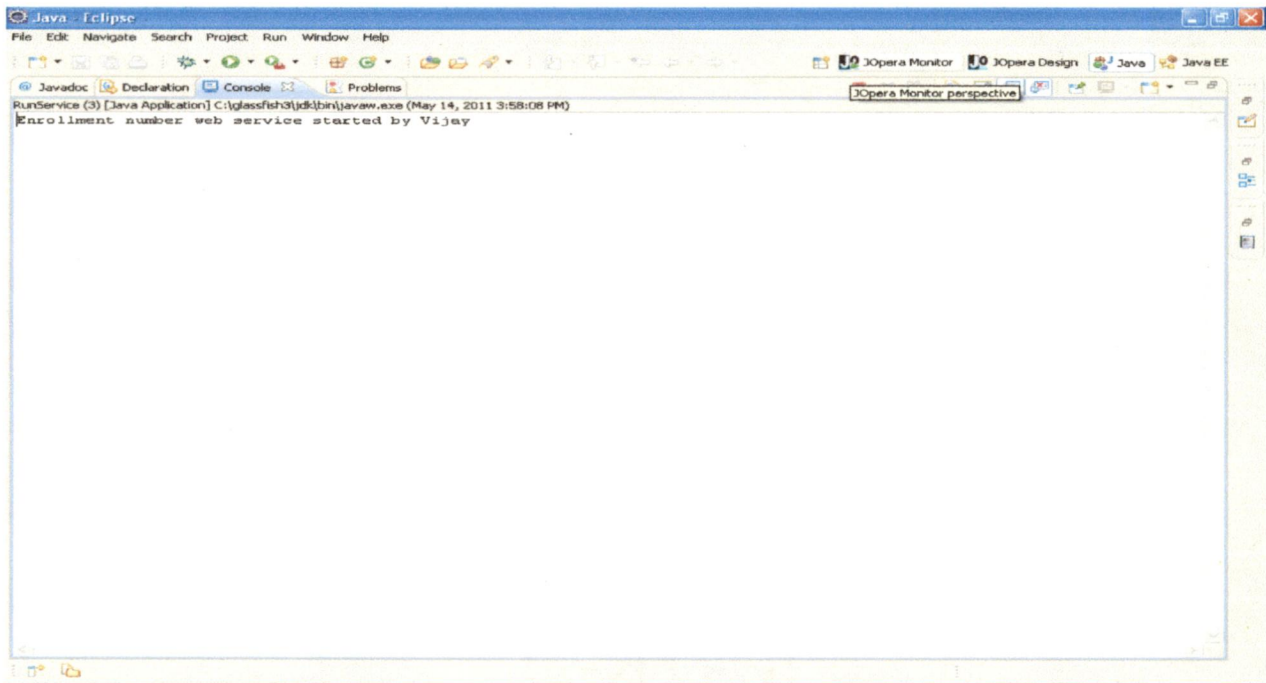


Figure4.3 Successful Run Message at console

Step5:View the wsdl:

- We open the internal Web browser in Eclipse by selecting Window > Show View > Other > General > Internal Web Browser.
- Now we type the URL, such as `http://localhost:8080/wsVijay3Example?wsdl`, which should display the web service's WSDL text, as shown in Appendix C
- When we have finished, we can stop the web service by clicking the red square in the Eclipse Console view.

Step6: Test the server:

Next we use the Eclipse Web Services Explorer tool to invoke the operations of a web service via native WSDL and SOAP to test the methods **enrollDecipher** and **getCourse** of the web service we just created.

- We need to change to the Java EE perspective. Click Window > Open Perspective > Other.
- When the window appears we select Java EE.
- Then we select Run > Launch the Web Services Explorer. Maximize the view by double- clicking its tab.

- Next we click the icon for WSDL page , this displays the WSDL page,
- In the Navigator pane, we click WSDL Main, and then enter the WSDL URL, in this case <http://localhost:8080/wsVijay3Example?wsdl>, then click the Go button.
- The WSDL is successfully opened, and we see a screen similar to Figure4.4

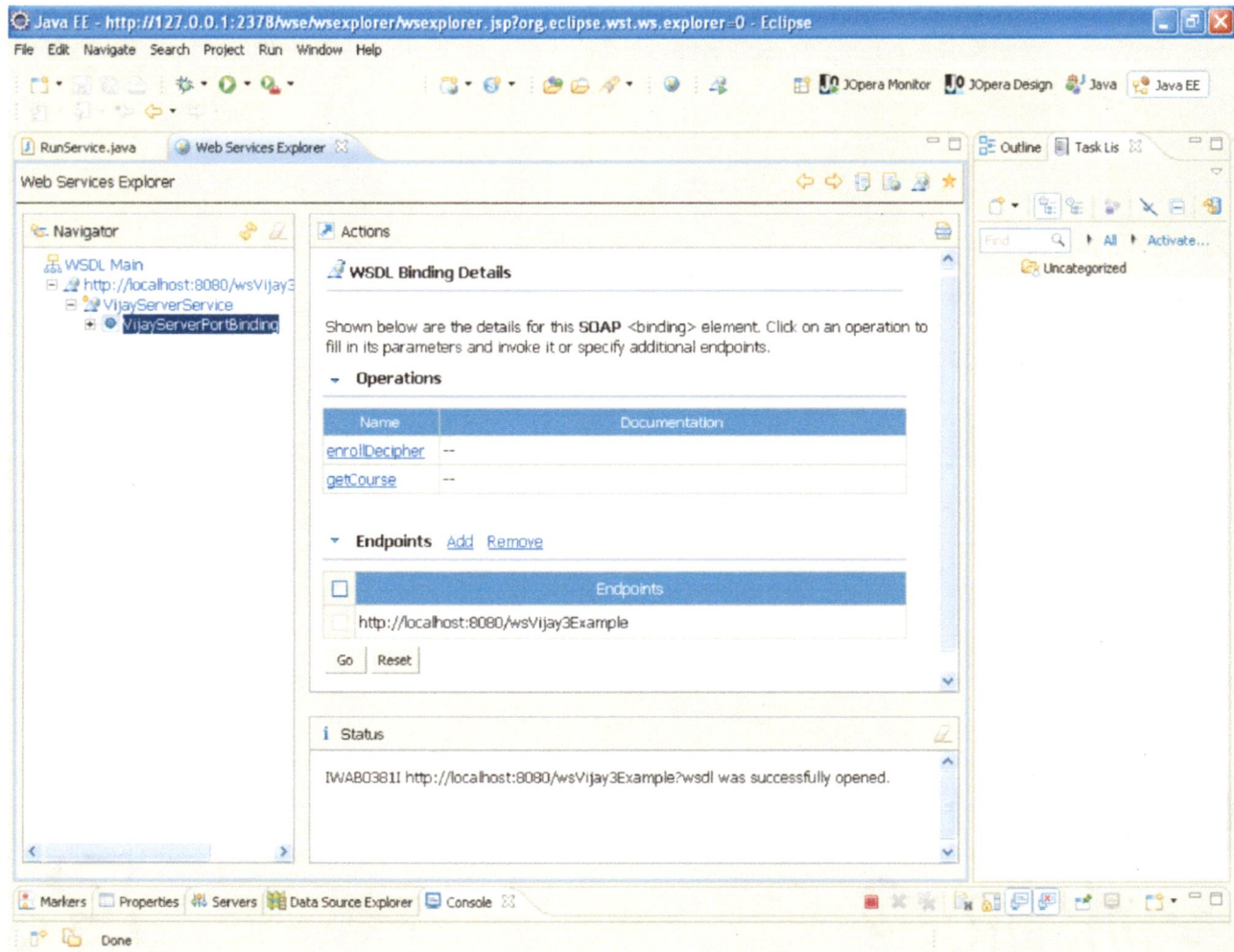


Figure4.4 WSDL binding details

- Next we invoke an operation by clicking enrollDecipher under Operations (shown in Figure4.4).
- Under the Body section, click the Add link (as shown in Figure4.5) to add a new row to the values table.
- Enter an enrollment number (here, 09536019), and click the Go button.

- In the Status section, enrollDecipher response displays the result.
We should see a result like return (string):
“hello ji.....Your Enrollment Number is:019 Your Branch is: IT Your Admission year is:2009”.

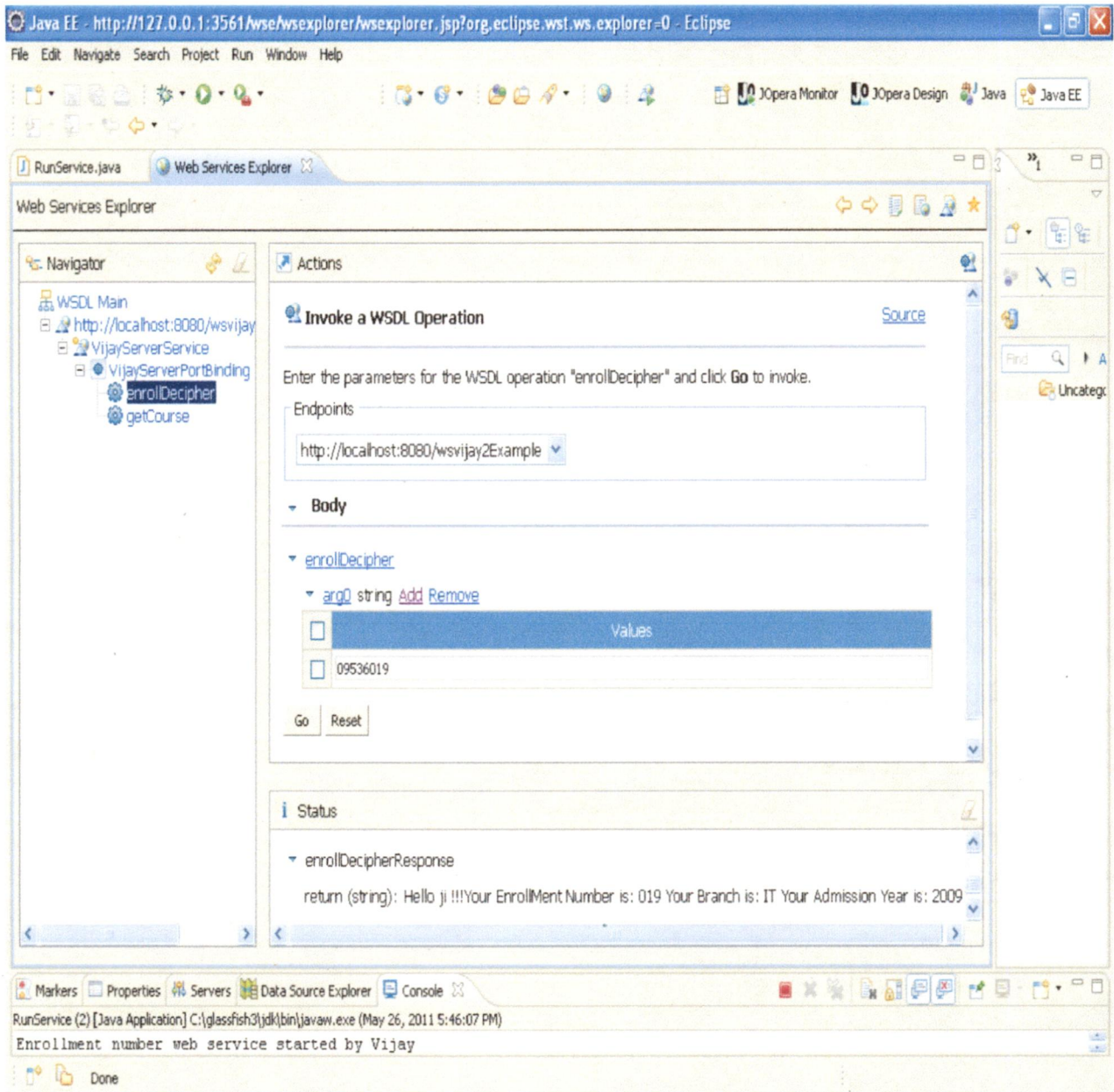


Figure4.5 The Result of invoking enrollDecipher.

4.2 Implementation of the service as a composite web service:

To implement the same service as a composite web service, we need to install JOpera, JOpera is built as a collection of plugins for Eclipse IDE.

4.2.1 JOpera for Eclipse[26]:

JOpera targets developers of Service-Oriented Business Applications and provides them with tools for rapid service composition. It includes a visual modeling environment, a light-weight execution engine, and also powerful debugging/refactoring tools which natively support the iterative nature of service composition. Service composition models in JOpera are defined at a higher level of abstraction than traditional BPM/BPEL languages and cover both architectural (structural) aspects as well as behavioral (flow) ones.

4.2.2 Some definitions related with JOpera[27]:

Process template: A process template describes how the tasks, its components, are connected together. It contains a control flow graph, which specifies the partial order to follow when starting the tasks as well as the data flow graph, which defines how tasks exchange data. A process templates is stored in an OML file.

Process instance: A process instance represents a running process template and contains the state of one execution, including all data that is produced and consumed by the tasks. Multiple instances of the same template can be active at the same time. We can use the Instance Navigator view in of the JOpera Monitor perspective to check what are the instances currently managed by the JOpera Kernel.

Task: A task is a basic process component. It can either be an activity or a subprocess.

An activity: An activity represents the invocation of an external program (or service) through a variety of protocols.

Program: A program is any software component or external system which can be accessed by JOpera using one of the following protocols.

- UNIX pipes (stdin/stdout) - for standard UNIX applications
- SOAP messages - for Web services
- Java local method invocations - for Java classes and Java snippets
- SSH - for remote UNIX command-line applications

- JDBC - to send SQL queries to a database directly from a process
- RESTful interactions on top of HTTP

The following steps are needed to implement the composite web service with JOpera.

Step1: Creating a new project:

In order to create a new Project, we right-click in the JOpera Navigator and then select New > JOpera Project. We choose an appropriate "Project name" ("my_web_service" in this case) and click on the "Finish" button.

Step2: Creating a new OML file:

Now that we have an empty JOpera project, we can add OML files into it by right-clicking the project in the JOpera Navigator and selecting New > OML File. Enter an appropriate file name ("composite.oml" in this case) and click on the "Finish" button.

Step3: Setting up the Composite service Process:

Before we can create a composition service process we need to define what the components are. In JOpera, we need to create some programs that will be later connected into a process. Here our process uses the following programs:

getcourse, getyears, getrollno, output, start and validity.

These programs are used as components to form the composite service process.

Step4: Creating a program:

(i) Click on the Add button in the Programs overview.

(ii) Click on the Edit button to edit the New Program.

(iii) Rename the program to getcourse.

(iv) The program is going to receive an input string and produce an output message. To exchange data, JOpera programs use input and output parameters.

- Add an Input Box Parameter and call it "enrolls".
- Add an Output Box Parameter named "course".(as shown in Figure4.6)

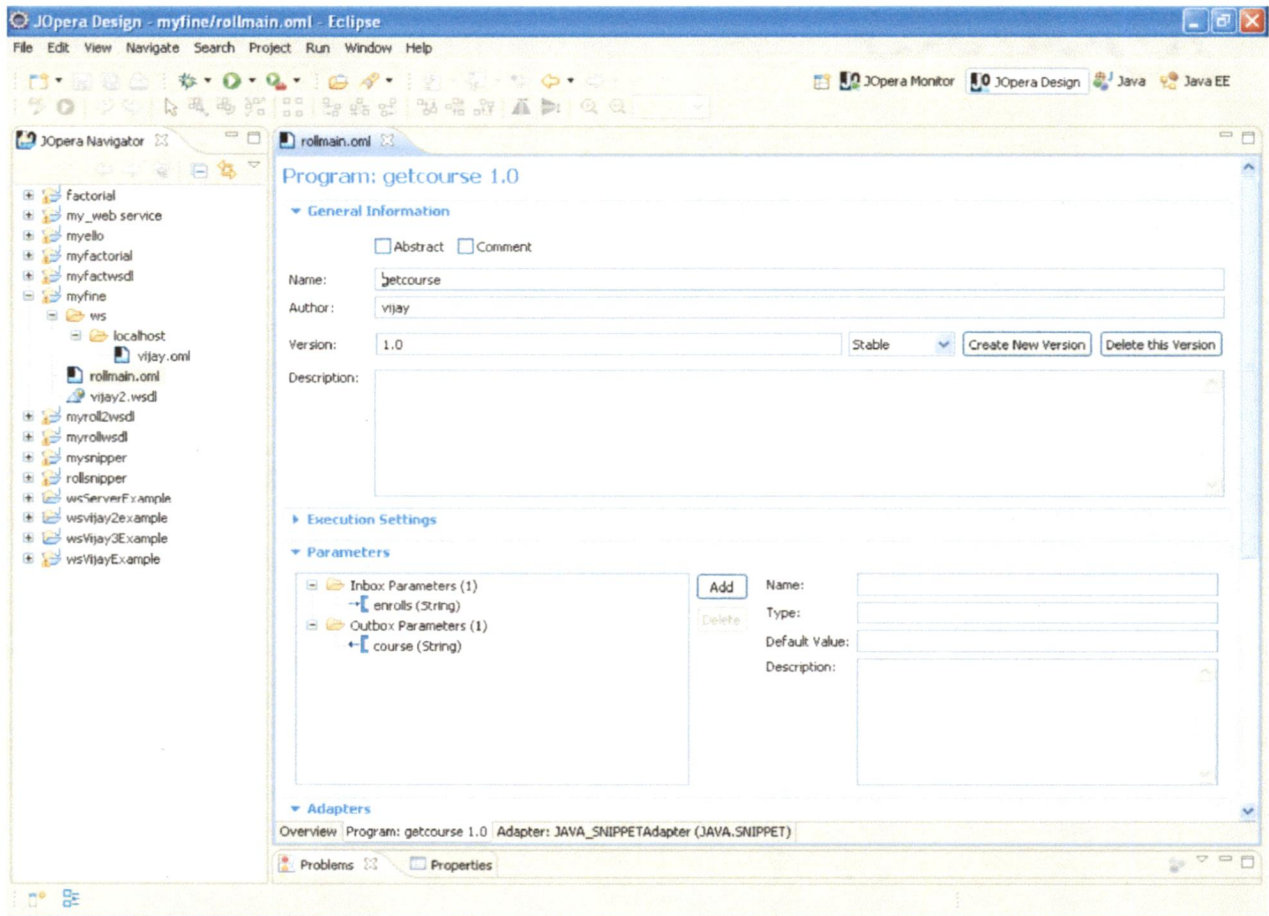


Figure4.6 Creating a program in JOpera

(v) We add an adapter describing how JOpera is going to run this program, we click on the Add... button within the Adapter (Access Method) section and

- Choose the JAVA.SNIPPET component type from the list in the dialog box and click Ok.
- Click on Edit.
- Then we enter the following java code shown in figure 4.7 that will perform the desired task:


```
course=enrolls.substring(3,5);
int courseid=Integer.parseInt(course);
switch(courseid){
    case 35
        course="CSE";
        break;
    case 36
        course="IT";
        break;
    default:
        course="NA";
        break;
}
```

Figure4.7 Java Snippet component for getcourse program

(vi) Running the Program with a test Process:

Now that we have setup the getcourse program, we can run it by calling it from a test process.

- Select the getcourse Program and click on the Test button.
This will create a new process which contains a single activity which references the program we just added. The process has the same input and output parameters and, if we check the data flow view, they are already connected to your program, which is now ready to test.
- Save the OML file.
- Click on the Start button to start the process. The button is located in the Overview tab next to the list of processes, , Since this is the first time, JOpera will prompt us to enter some values for its input parameters. Enter 09536019 for the input parameter in and click Run.

(vii) Checking the Results:

If all went well, the process runs very fast and is finished by the time Eclipse has switched to the JOpera Monitor perspective. We can look in the Properties view for the values of the output parameters as shown in the figure4.8.

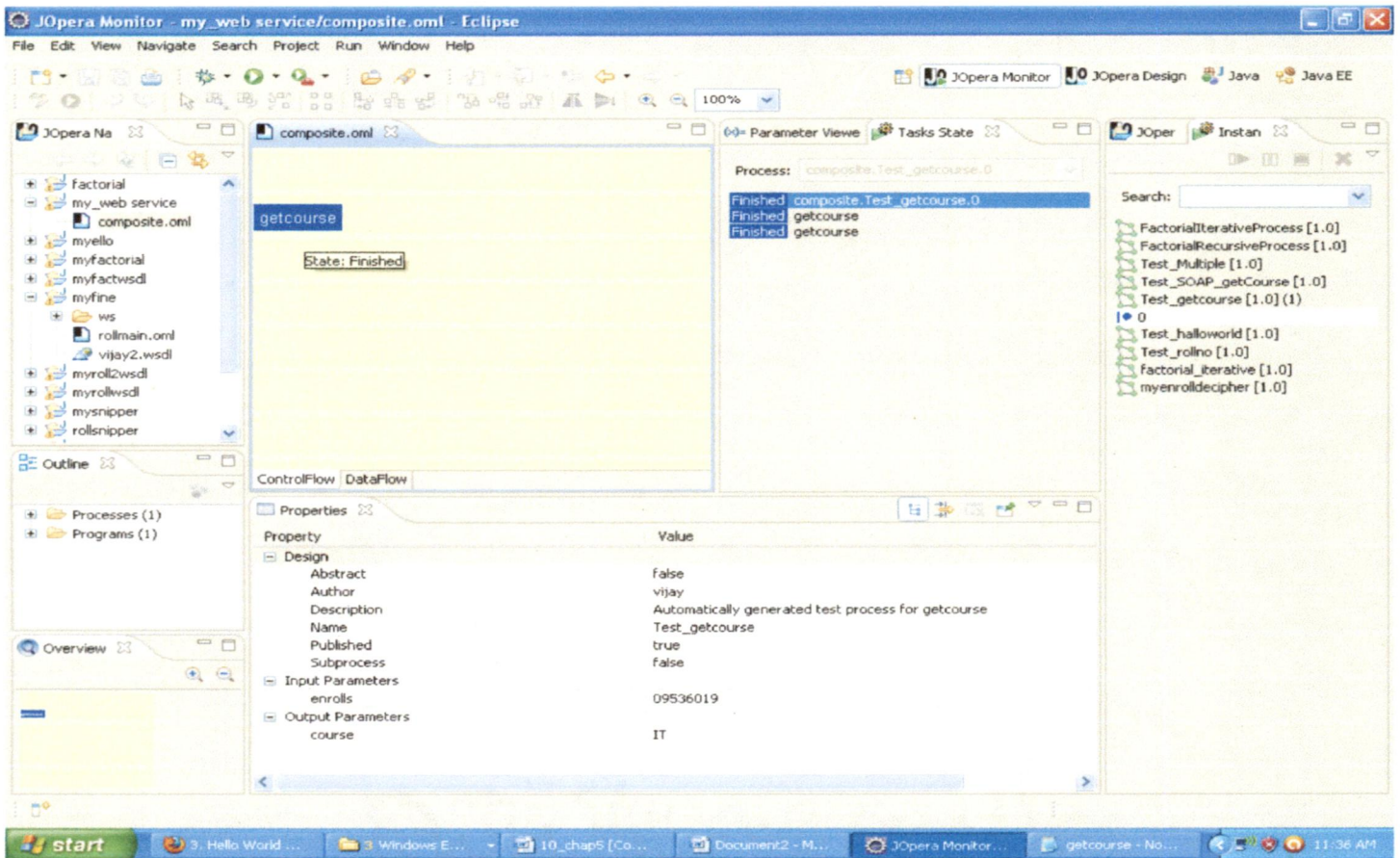


Figure4.8 Running the program as a process

Step5: Creating other component programs:

Similar to step4, we can create the other programs like getyears, getrollno, output, start and validity. The input and output parameters for each program with java snippet code are listed in Appendix C.

Step6: Creating the new Process :

- (i) Click on the Add button in the Processes overview
- (ii) Click on the Edit button to edit the New Process
- (iii) Rename the process to myenrolldecipher

(iv) The process is going to receive an input string and produce an output message. To exchange data, JOpera processes use input and output parameters.

- Add an Input Box Parameter and call it “enrollmentno”.
- Add an Output Box Parameter named “yourstatus”.

(v)Populating the Process:

After having created the process, we populate it with tasks. In order to do this we can simply drag programs from the "Outline" view and drop them on the data flow of the process we wish to populate.

(vi)Draw Data Flow Connections:

In order to draw the data flow connections, we first have to switch to the "Data Flow" view of this process. Then the input and output parameters of the process need to be displayed. As soon as this has been done, the input/output parameters of the process are connected with input/output parameters of various tasks so that correct flow of data is maintained. In case of this process the data flow is defined as shown in Figure4.9:

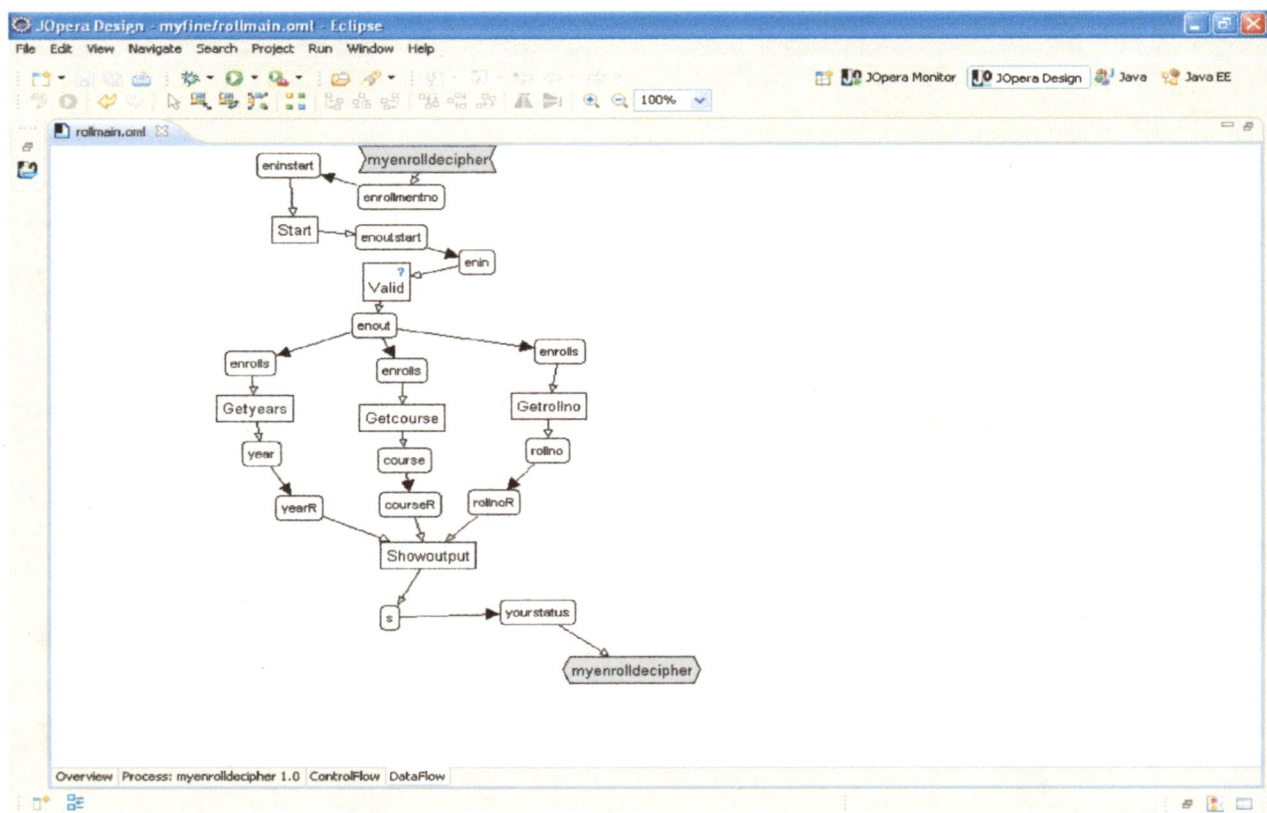


Figure4.9 Data Flow Diagram for the Process

Step7 Compilation of the Process:

Compilation of the process is simply done by saving the OML file. Given that there are no problems (check the "Problems" view), the process should be compiled upon saving the corresponding OML file. All we need to make sure is that in the menu "Project", "Build Automatically" is selected.

Step8.Executing and Monitoring the Process:

In order to run the process, we switch to the overview page, and select the process name we want to start and then click on the Start button. Make sure that the oml file has been saved.. The process will be started as soon as we click on Run. The next time we start the process, it will be immediately executed. If we want to change the input parameter values, we should use the Run... menu and look for the launch configuration corresponding to our process.

The control flow and data flow diagrams of the process are shown in figure4.11 and figure4.12 respectively.

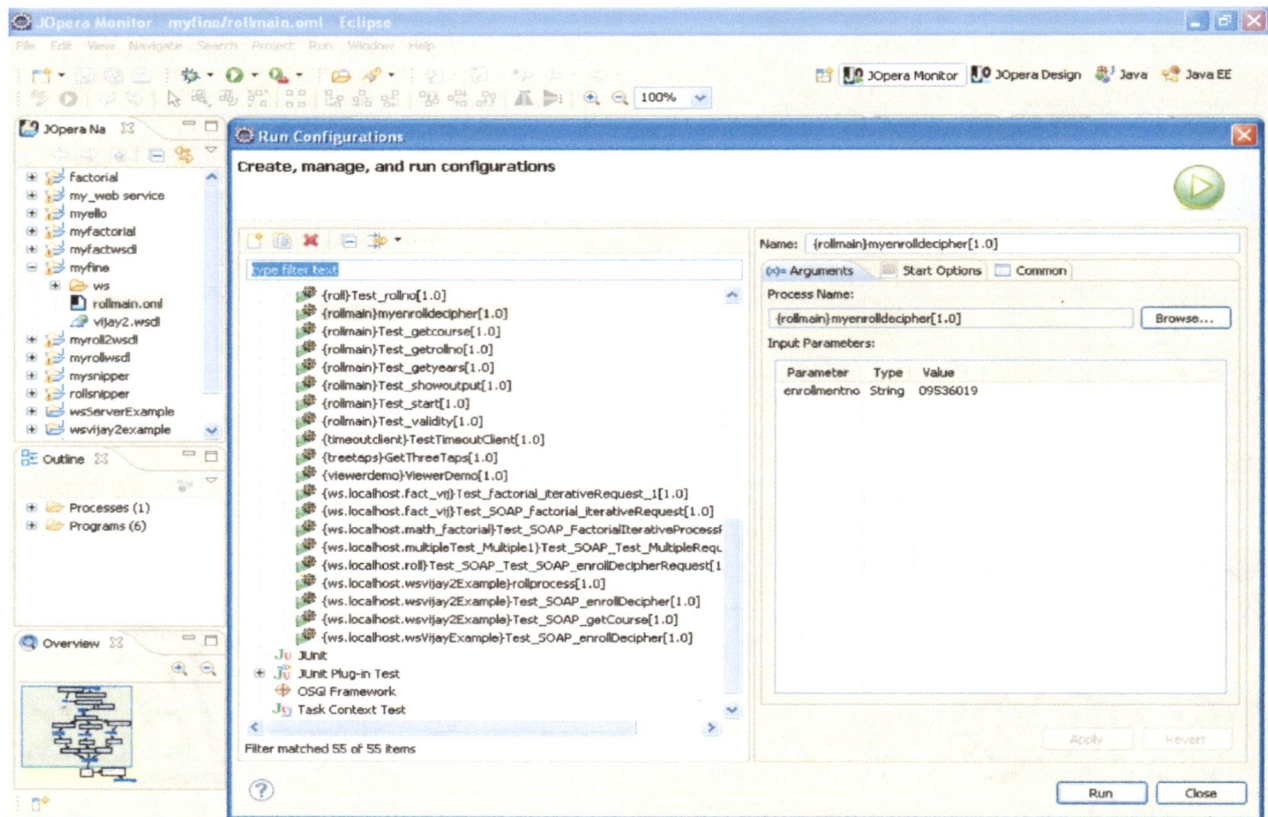


Figure4.10 Run configuration for the process

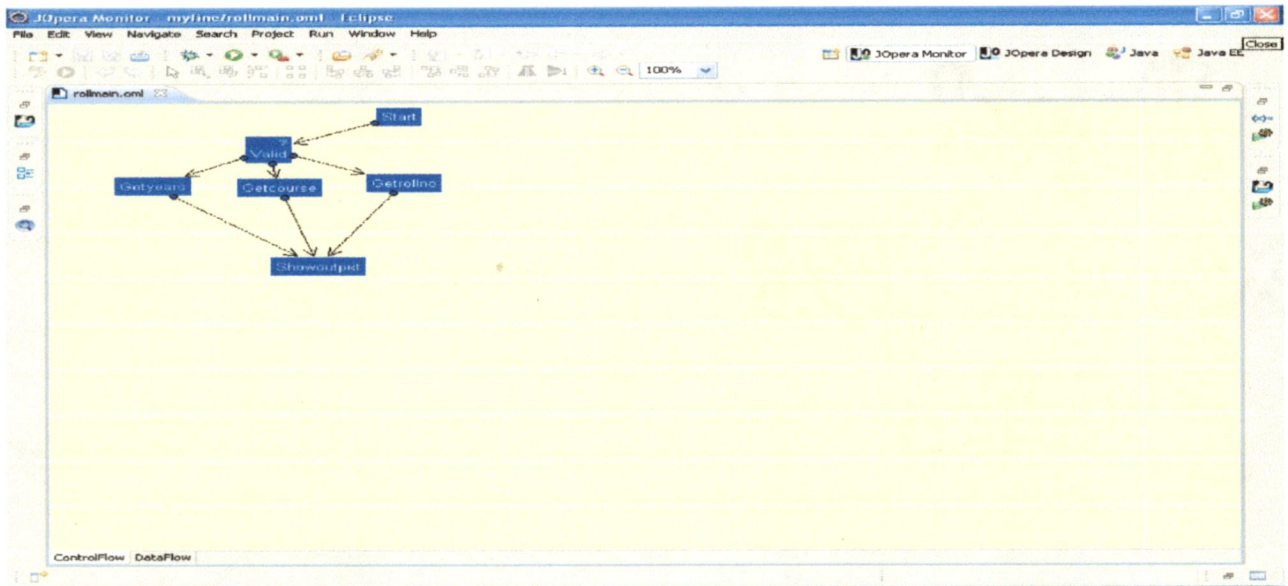


Figure4.11 Control Flow Diagram of the Composite Process

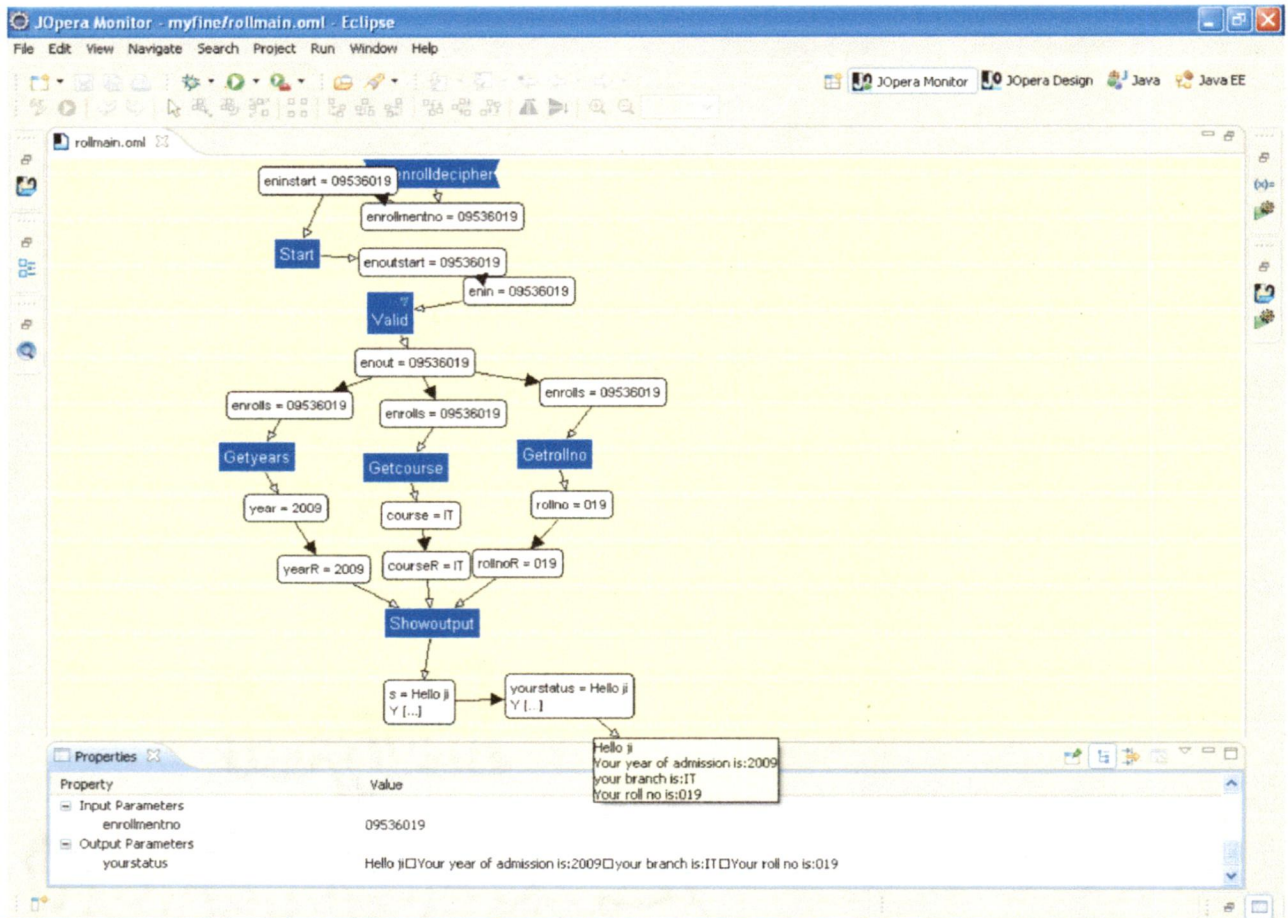


Figure4.12 Data Flow Diagram of the Composite Process

Chapter 5

Results and Discussions

We have implemented the same service as a single web service and as a composite web service. Both the implementations provide the same result.

5.1 The Result of the service as a single web service:

We implement the service as single web service with the help of Eclipse IDE, and when we explore the single web service with the help of web service explorer tool, the result can be seen in the status window as shown in Figure 5.1

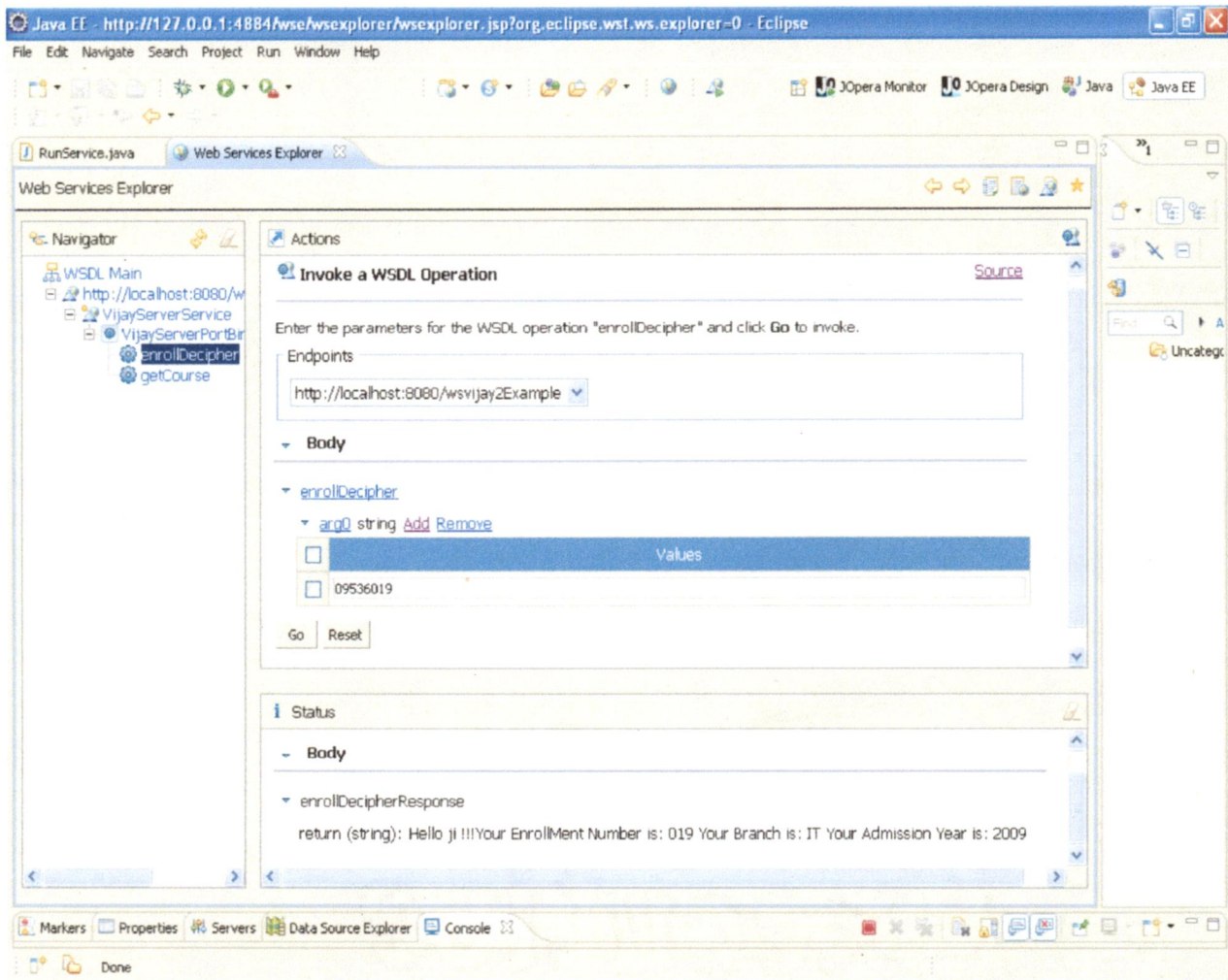


Figure5.1 The Result of service as single web service

5.2 The Result of the service as a composite web service:

We implement the service as composite web service with the help of Eclipse IDE and JOpera, and when we explore the composite web service with the help of JOpera Monitor perspective the data flow graph and control flow graph both show the successful finished tasks. The result can also be seen in the properties window as shown in figure5.2.

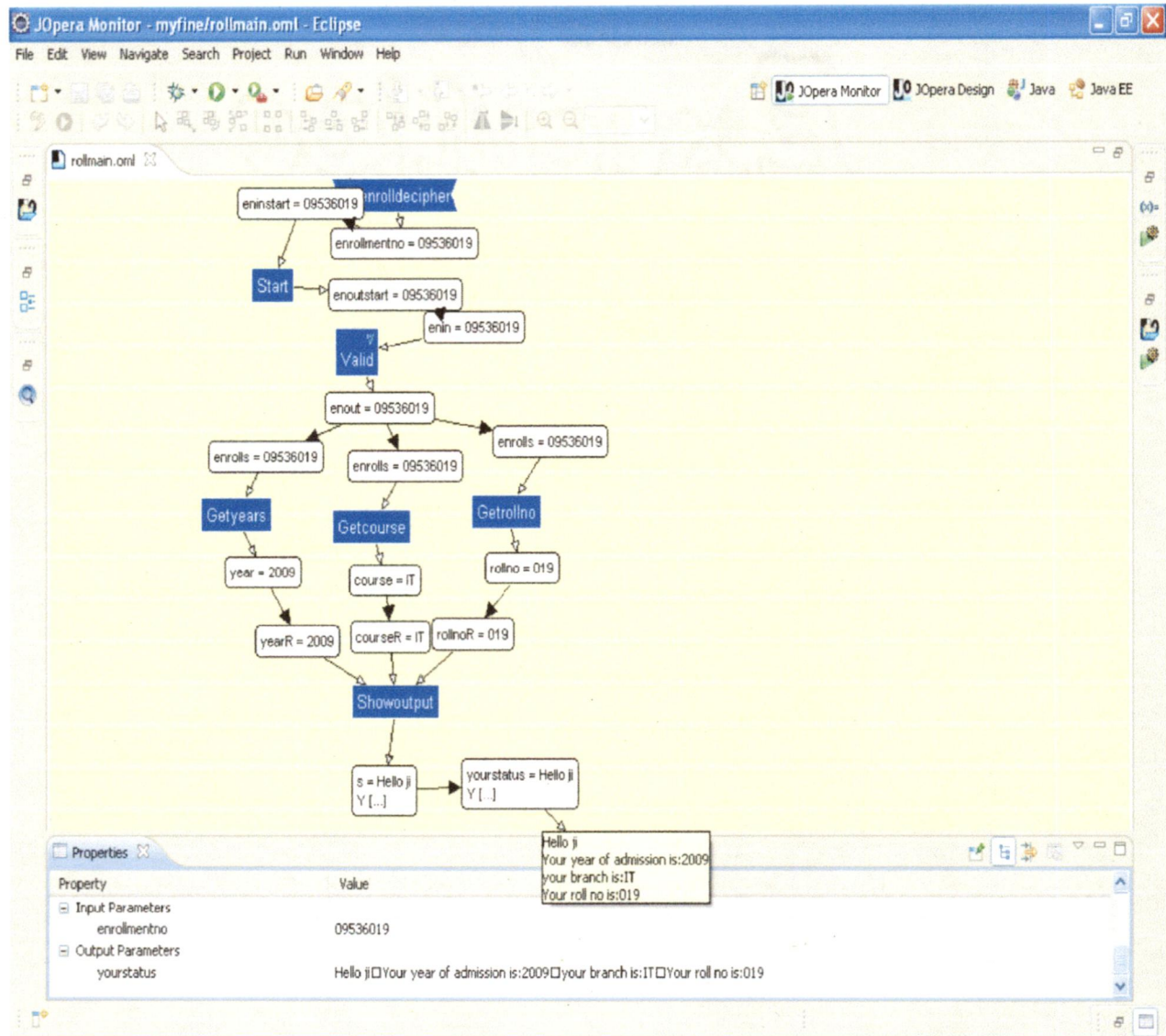


Figure5.2 The Result of service as composite web service

5.3 Mapping of composite web service:

The following diagram figure 5.3 shows the message interaction between component web services and composite web service.

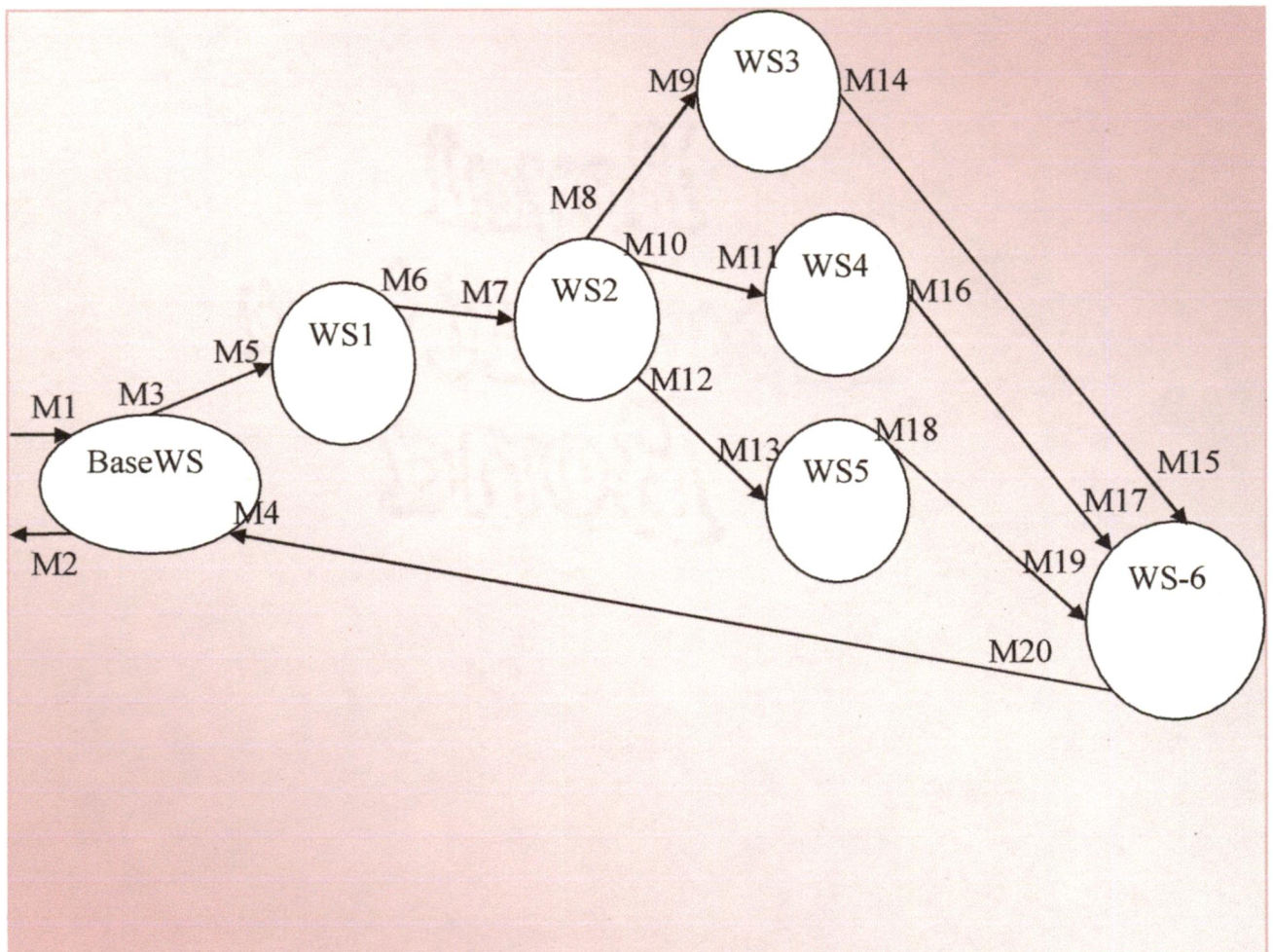


Figure5.3 Message Interaction Diagram

For Each single service we can write single service model as:

For web service BaseWS:

BaseWS=($\Sigma_{base}, \delta_{base}, \Delta_{base}, P_{base}, F_{base}$) where

$\Sigma_{base} = \{ M_1, M_2, M_3, M_4 \}$;

$\Sigma^{req} = \{ M_1, M_3 \}$ $\Sigma^{res} = \{ M_2, M_4 \}$

$\Sigma^{in} = \{ M_1, M_4 \}$ $\Sigma^{out} = \{ M_2, M_3 \}$

$\Sigma^{req_in} = \{ M_1 \}$ $\Sigma^{req_out} = \{ M_3 \}$

$$\Sigma^{\text{res_in}} = \{M_4\} \quad \Sigma^{\text{res_out}} = \{M_2\}$$

δ_{base} is defined as follows:

$$\delta_{\text{base}}(M_1) = M_2, \delta_{\text{base}}(M_3) = M_4$$

Δ_{base} is defined as: $\Delta_{\text{base}}(M_1) = M_3$

For web service WS1:

$WS1 = (\Sigma_{ws1}, \delta_{ws1}, \Delta_{ws1}, P_{ws1}, F_{ws1})$ where

$$\Sigma_{ws1} = \{M_5, M_6\}$$

$$\Sigma^{\text{req}} = \{M_5, M_6\} \quad \Sigma^{\text{res}} = \Phi$$

$$\Sigma^{\text{in}} = \{M_5\} \quad \Sigma^{\text{out}} = \{M_6\}$$

$$\Sigma^{\text{req_in}} = \{M_5\} \quad \Sigma^{\text{req_out}} = \{M_6\}$$

δ_{ws1} is not defined and Δ_{ws1} is defined as: $\Delta_{ws1}(M_5) = M_6$.

For web service WS2:

$WS2 = (\Sigma_{ws2}, \delta_{ws2}, \Delta_{ws2}, P_{ws2}, F_{ws2})$ where

$$\Sigma_{ws2} = \{M_7, M_8, M_{10}, M_{12}\}$$

$$\Sigma^{\text{req}} = \{M_7, M_8, M_{10}, M_{12}\} \quad \Sigma^{\text{res}} = \Phi$$

$$\Sigma^{\text{in}} = \{M_7\} \quad \Sigma^{\text{out}} = \{M_8, M_{10}, M_{12}\}$$

$$\Sigma^{\text{req_in}} = \{M_7\} \quad \Sigma^{\text{req_out}} = \{M_8, M_{10}, M_{12}\}$$

δ_{ws2} is not defined and Δ_{ws2} is defined as: $\Delta_{ws2}(M_7) = \{M_8, M_{10}, M_{12}\}$.

For web service WS3:

$WS3 = (\Sigma_{ws3}, \delta_{ws3}, \Delta_{ws3}, P_{ws3}, F_{ws3})$ where

$$\Sigma_{ws3} = \{M_9, M_{14}\}$$

$$\Sigma^{\text{req}} = \{M_9, M_{14}\} \quad \Sigma^{\text{res}} = \Phi$$

$$\Sigma^{\text{in}} = \{M_9\} \quad \Sigma^{\text{out}} = \{M_{14}\}$$

$$\Sigma^{\text{req_in}} = \{M_9\} \quad \Sigma^{\text{req_out}} = \{M_{14}\}$$

δ_{ws3} is not defined and Δ_{ws3} is defined as: $\Delta_{ws3}(M_9) = M_{14}$.

For web service WS4:

$WS4 = (\Sigma_{ws4}, \delta_{ws4}, \Delta_{ws4}, P_{ws4}, F_{ws4})$ where

$$\Sigma_{ws4} = \{M_{11}, M_{16}\}$$

$$\Sigma^{\text{req}} = \{M_{11}, M_{16}\} \quad \Sigma^{\text{res}} = \Phi$$

$$\Sigma^{\text{in}} = \{M_{11}\} \quad \Sigma^{\text{out}} = \{M_{16}\}$$

$$\Sigma^{req_in} = \{ M_{11} \} \quad \Sigma^{req_out} = \{ M_{16} \}$$

δ_{ws4} is not defined and Δ_{ws4} is defined as: $\Delta_{ws4}(M_{11}) = M_{16}$.

For web service WS5:

$WS5 = (\Sigma_{ws5}, \delta_{ws5}, \Delta_{ws5}, P_{ws5}, F_{ws5})$ where

$$\Sigma_{ws5} = \{ M_{13}, M_{18} \}$$

$$\Sigma^{req} = \{ M_{13}, M_{18} \} \quad \Sigma^{res} = \Phi$$

$$\Sigma^{in} = \{ M_{13} \} \quad \Sigma^{out} = \{ M_{18} \}$$

$$\Sigma^{req_in} = \{ M_{13} \} \quad \Sigma^{req_out} = \{ M_{18} \}$$

δ_{ws5} is not defined and Δ_{ws5} is defined as: $\Delta_{ws5}(M_{13}) = M_{18}$.

For web service WS6:

$WS6 = (\Sigma_{ws6}, \delta_{ws6}, \Delta_{ws6}, P_{ws6}, F_{ws6})$ where

$$\Sigma_{ws6} = \{ M_{15}, M_{17}, M_{19}, M_{20} \}$$

$$\Sigma^{req} = \{ M_{15}, M_{17}, M_{19} \} \quad \Sigma^{res} = \{ M_{20} \}$$

$$\Sigma^{in} = \{ M_{15}, M_{17}, M_{19} \} \quad \Sigma^{out} = \{ M_{20} \}$$

$$\Sigma^{req_in} = \{ M_{15}, M_{17}, M_{19} \} \quad \Sigma^{req_out} = \Phi$$

$$\Sigma^{res_in} = \Phi \quad \Sigma^{res_out} = \{ M_{20} \}.$$

δ_{ws6} is not defined and Δ_{ws6} is also not defined.

The Service Composition pattern is $SC = (Q, T, R)$ where:

$$Q = \{ BaseWS, WS1, WS2, WS3, WS4, WS5, WS6 \}$$

$$T: T(M_3) = M_5, T(M_6) = M_7, T(M_8) = M_9, T(M_{10}) = M_{11}, T(M_{12}) = M_{13}, T(M_{14}) = M_{15},$$

$$T(M_{16}) = M_{17}, T(M_{18}) = M_{19}.$$

$$R: M_8 ||| M_{19} ||| M_{12}.$$

So the Service Composition Model is $SM = \{ \Sigma, \delta, \Delta, P, F, SC \}$ where:

$$\Sigma = \{ M_1, M_2 \}, \Sigma^{in} = \{ M_1 \}, \Sigma^{out} = \{ M_2 \}, \Sigma^{req} = \{ M_1 \}, \Sigma^{res} = \{ M_2 \}, \delta(M_1) = M_2$$

Δ is an empty set, the algorithm of F is consistent with single service model.

5.4 Verification using Pi-calculus:

The Pi-calculus based verification of our composite web service can be performed by expressing both the composite web service and the service requirement as pi-calculus process.

The service requirement can be expressed as Pi-calculus process:

$$P_{\text{serviceRequirement}} = \bar{a}\langle \text{enroll} \rangle . e\langle \text{status} \rangle$$

Where a and e are channel names.

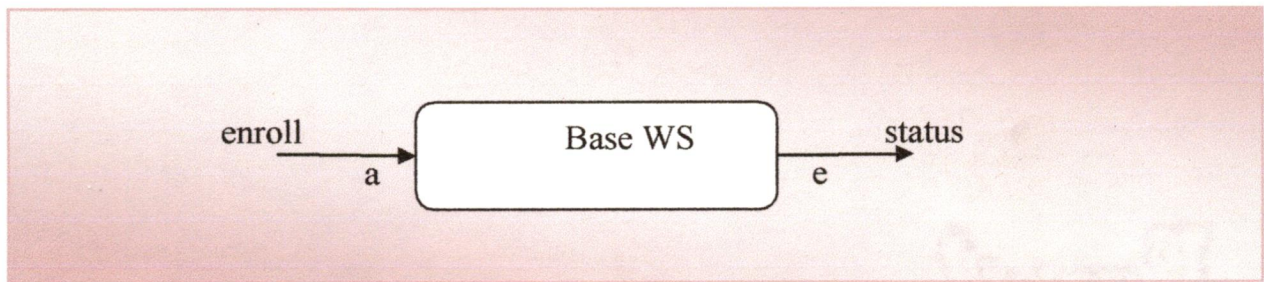


Figure5.4 Channels and Messages interaction for composite service

As our composite service is composed of five component web services, the interaction between component services with channels and messages can be graphically described as shown in the figure5.6:

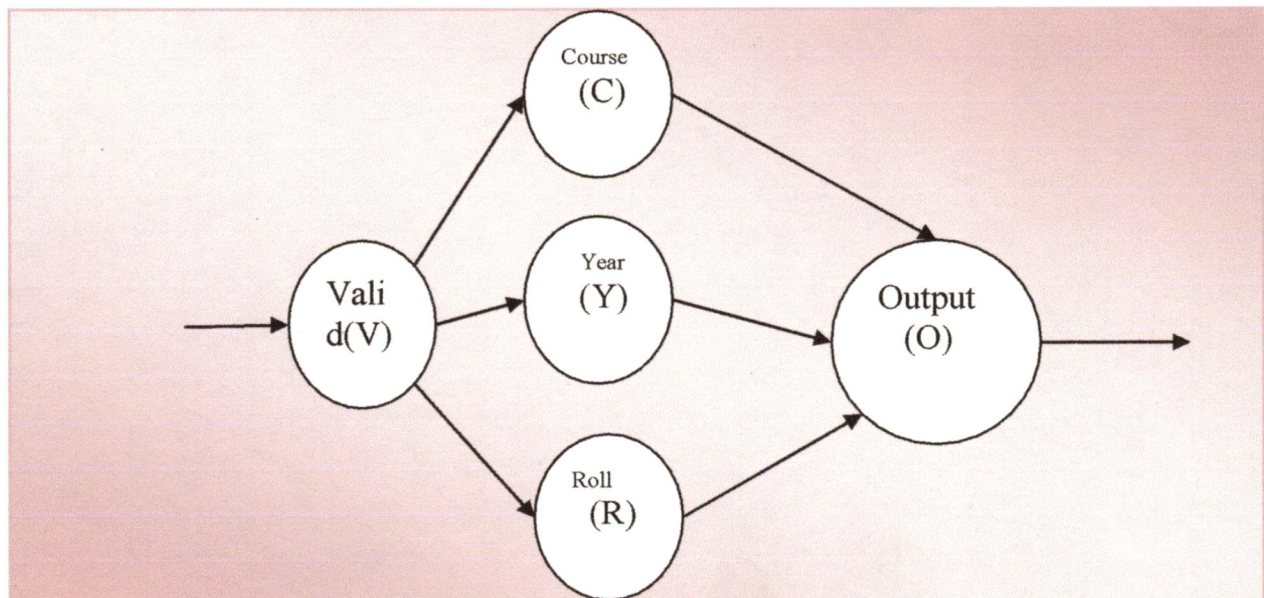


Figure5.5 Channels and Messages interaction between components

The pi-calculus expression for each component can be written as:

$$P_{\text{valid}} = a\langle \text{enroll} \rangle.(\bar{b}\langle \text{enroll} \rangle \mid \bar{c}\langle \text{enroll} \rangle \mid \bar{d}\langle \text{enroll} \rangle)$$

$$P_{\text{course}} = b\langle \text{enroll} \rangle.\bar{b}'\langle \text{course} \rangle$$

$$P_{\text{year}} = c\langle \text{enroll} \rangle.\bar{c}'\langle \text{year} \rangle$$

$$P_{\text{roll}} = d\langle \text{enroll} \rangle.\bar{d}'\langle \text{roll} \rangle$$

$$P_{\text{output}} = (b'\langle \text{course} \rangle \mid c'\langle \text{year} \rangle \mid d'\langle \text{course} \rangle).\bar{e}\langle \text{status} \rangle$$

Now we can write the pi-calculus expression for composite web service as:

$$\begin{aligned} P_{\text{CompositeService}} &= P_{\text{valid}}.(P_{\text{course}} \mid P_{\text{year}} \mid P_{\text{roll}}).P_{\text{output}} \\ &= a\langle \text{enroll} \rangle.(\bar{b}\langle \text{enroll} \rangle \mid \bar{c}\langle \text{enroll} \rangle \mid \bar{d}\langle \text{enroll} \rangle)(b\langle \text{enroll} \rangle.\bar{b}'\langle \text{course} \rangle \mid \\ &\quad c\langle \text{enroll} \rangle.\bar{c}'\langle \text{year} \rangle \mid d\langle \text{enroll} \rangle.\bar{d}'\langle \text{roll} \rangle). \\ &\quad (b'\langle \text{course} \rangle \mid c'\langle \text{year} \rangle \mid d'\langle \text{roll} \rangle).\bar{e}\langle \text{status} \rangle \end{aligned}$$

Now we have formalized the service requirement and composite service with pi-calculus processes, we can reason about the correctness of composite service formally.

$$\begin{aligned} P_{\text{CompositeService}} &= a\langle \text{enroll} \rangle.(\bar{b}\langle \text{enroll} \rangle \mid \bar{c}\langle \text{enroll} \rangle \mid \bar{d}\langle \text{enroll} \rangle)(b\langle \text{enroll} \rangle.\bar{b}'\langle \text{course} \rangle \mid \\ &\quad c\langle \text{enroll} \rangle.\bar{c}'\langle \text{year} \rangle \mid d\langle \text{enroll} \rangle.\bar{d}'\langle \text{roll} \rangle). \\ &\quad (b'\langle \text{course} \rangle \mid c'\langle \text{year} \rangle \mid d'\langle \text{roll} \rangle).\bar{e}\langle \text{status} \rangle \\ &\xrightarrow{\text{course}} a\langle \text{enroll} \rangle.(\bar{b}\langle \text{enroll} \rangle \mid \bar{c}\langle \text{enroll} \rangle \mid \bar{d}\langle \text{enroll} \rangle)(b\langle \text{enroll} \rangle. \mid \\ &\quad c\langle \text{enroll} \rangle.\bar{c}'\langle \text{year} \rangle \mid d\langle \text{enroll} \rangle.\bar{d}'\langle \text{roll} \rangle). \\ &\quad (c'\langle \text{year} \rangle \mid d'\langle \text{roll} \rangle).\bar{e}\langle \text{status} \rangle \\ &\xrightarrow{\text{year}} a\langle \text{enroll} \rangle.(\bar{b}\langle \text{enroll} \rangle \mid \bar{c}\langle \text{enroll} \rangle \mid \bar{d}\langle \text{enroll} \rangle)(b\langle \text{enroll} \rangle. \mid \\ &\quad c\langle \text{enroll} \rangle. \mid d\langle \text{enroll} \rangle.\bar{d}'\langle \text{roll} \rangle). \\ &\quad (d'\langle \text{roll} \rangle).\bar{e}\langle \text{status} \rangle \end{aligned}$$

$$\xrightarrow{roll} a\langle enroll \rangle . (\bar{b} \langle enroll \rangle \mid \bar{c} \langle enroll \rangle \mid \bar{d} \langle enroll \rangle) (b \langle enroll \rangle . \mid c \langle enroll \rangle . \mid d \langle enroll \rangle .) . \bar{e} \langle status \rangle$$

$$\xrightarrow{3 \text{ times}} a \langle enroll \rangle . \bar{e} \langle status \rangle$$

Now we perform verification:

$$P_{CompositeService} \mid P_{serviceRequirement} = a \langle enroll \rangle . \bar{e} \langle status \rangle \mid \bar{a} \langle enroll \rangle . e \langle status \rangle$$

$$\xrightarrow{enroll, status} 0 \mid 0.$$

From the above verification process, the transition sequence of the composite service and service requirement terminates at a null process, it indicates that both composite service and service requirement process can come to an end after sending and receiving messages. Thus the composite service can achieve the goal of service requirement.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

On the modeling of service composition, there are many research topics in both academia and industries. Building models for service composition is not only helpful to understand the service composition precisely, but also helpful to analyze and verify some properties specific to service composition and assure the correctness and quality of service.

In this dissertation, we have discussed how a formal model of web service is described based on some composition pattern ideas. The model concentrates on message interaction between services, so it can be used to simulate message interaction between services.

Here, we have implemented the same service as a single service as well as composite service. The single service is implemented with Eclipse IDE while the composite service is implemented with JOpera plugins.

The formal model is also featured with the following dimensions:

- Additional constraint on web service composition.
- Relationships between component web services.
- Flow web services.
- Composite service verification.
- QoS parameters.

6.2 Future Work

The described formal model is just a initial model of service composition, in future work, complex service composition properties should be studied in more details, the merits of all kinds of related models should be absorbed to improve the model's expression ability so as to simulate the execution of service composition.

Also the implementation of composite service verification framework is an important research aspect that can be done in future. The relation of formal model with queuing network model can be explored to evaluate performance measures of interest.

REFERENCES

- [1] Chifu V.R., Salomie I. and St. Chifu E., “Fluent calculus-based Web service composition –From OWL-S to fluent calculus”. In: *ICCP 4th International Conference on Intelligent Computer Communication and Processing*, Cluj-Napoca, 28-30 Aug. 2008, pp.161 – 168.
- [2] Huaiguang Wu and Guoqing Wu, “Formal Depiction of Composition of Web Services Based on CCS and Modal μ -calculus”. In: *IEEC International Symposium on Information Engineering and Electronic Commerce*, Ternopil, 16-17 May 2009, pp. 408 – 412.
- [3] Simple Object Access Protocol[Online]. Available: <http://www.w3.org/TR/soap>.
- [4] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana. Web Services Descript Language(WSDL)1.1[Online]. Available: <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>.
- [5] Universal Description, Discovery and Integration of Web Services[Online]. Available: <http://www.uddi.org>.
- [6] Zuohua Ding, Mingyue and JiangJing Liu, “Model Checking Service Component Composition by SPIN”. In: *8th IEEE/ACIS International Conference on Computer and Information Science*, Shanghai, 1-3 June 2009, pp. 1029 – 1034.
- [7] Web Services [Online]. Available: <http://www.w3.org/2002/ws/Activity>.
- [8] Jia Zhang, Chang, C.K., Jen-Yao Chung and Kim, S.W., “WS-Net: A Petri-net Based Specification Model for Web Services”. In: *Proc. IEEE International Conference on Web Services*, 6-9 July 2004, pp. 420 – 427.
- [9] Thomas, J.P., Thomas, M., and Ghinea, G., “Modeling of Web Services Flow”. In: *IEEE International Conference on E-Commerce*, 24-27 June 2003, pp. 391 – 398.
- [10] Jin-dian Su, Shan-shan Yu and He-qing Guo, “Dynamic Substitutability Analysis of Web Service Composition via Extended Pi-Calculus”. In: *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, Shanghai, 17-20 Dec. 2008, pp. 447 – 452.
- [11] Yu Tang, Luo Chen, Kai-Tao He and Ning Jing, “SRN: An Extended Petri-Net-Based Workflow Model for Web Services”. In: *Proc. IEEE International Conference on Web Services*, 6-9 July 2004, pp. 591 – 599.

- [12] Michael C. Daconta, Leo J. Obrst, Kevin T. Smith “Understanding Web Services” in *The Semantic Web*, Ed., 1st ed. Indianapolis, Indiana: Wiley Publishing, Inc., 2003, pp. 57-61.
- [13] Michael C. Daconta, Leo J. Obrst, Kevin T. Smith “Understanding Web Services” in *The Semantic Web*, Ed., 1st ed. Indianapolis, Indiana: Wiley Publishing, Inc., 2003, pp. 65-70.
- [14] Ethan Cerami “WSDL Essentials” in *Web Services Essentials*, Ed., 1st ed. New York: O'Reilly, 2002, pp. 102-105.
- [15] Huigui Rong, Ning Zhou, Hongqin Chen and Hongli Cheng, “Research on Strategy of Web Service Composition based on Software Life Cycle”. In: *WiCOM'4th International Conference on Wireless Communications, Networking and Mobile Computing*, Dalian, 12-14 Oct. 2008, pp. 1 – 4.
- [16] Bixin Li, Yu Zhou, Ying Zhou and Xufang Gong, “A Formal Model for Web Service Composition and Its Application Analysis”. In: 2nd IEEE Conference on Asia-Pacific Service Computing, Tsukuba Science City, 2007, 11-14 Dec. 2007, pp. 204 – 210.
- [17] B. Benatallah, M. Dumas and M-C. Fauvet, “Towards Patterns of Web Services Composition”, In *Patterns and Skeletons for Parallel and Distributed Computing*, Springer Verlag, UK, “citeseer.ist.psu.edu/benatallah02towards.html”.
- [18] Kaiyu Wang and Naishuo Tian, “Performance Modeling of Composite Web Services”. In : *PACCS Pacific-Asia Conference on Circuits, Communications and Systems*, Chengdu, 16-17 May 2009, pp. 563 – 566.
- [19] The Pi-calculus [Online]. Available: <http://en.wikipedia.org/wiki/pi-calculus> .
- [20] Yanbin Peng, Lv Ye, Zhijun Zheng, Jian Xiang, Ji Gao, Jieqing Ai; Zhenyu Lu, Yu Jin and Xueqin Jiang, “Automatic service composition verification based on Pi-calculus”. In : *EBISS '09 International Conference on E-Business and Information System Security*, 23-24 May 2009, pp. 1 – 4.
- [21] QoS for Web Services: requirements and possible approaches [Online]. Available: <http://www.w3c.or.kr/kr-office/TR/2003/ws-qos/>.

[22] Aijun Jiang, Xiaoyong Mei, Shixian Li and Fudan Zheng, "A QoS Tool Framework for Developing Composite Web Service". In : *ISISE '08 International Symposium on Information Science and Engineering*, 20-22 Dec. 2008,pp. 663 – 668.

[23] E. Lazowska, J. Zahorjan, S. Graham and K. Sevcik, Quantitative System Performance: Computer System Analysis Using Queueing Network Models, Prentice Hall, Englewood Cliffs, N. J., 1984,pp.203-300.

[24] Java SE 6[Online]. Available:
<http://www.oracle.com/technetwork/java/javase/documentation/index.html> .

[25] Eclipse IDE[Online]. Available: <http://www.eclipse.org/> .

[26] JOpera plugins [Online]. Available: <http://www.jopera.org/>

[27] FAQs of JOpera[Online]. Available: http://www.jopera.org/docs/help/jop_7.html/

Appendix A

Java class: VijayServer

```
package com.myfirst.wsServer;
import javax.ws.WebService;
@WebService
public class VijayServer {
    public String enrollDecipher(String enrollS) {
        enrollS =enrollS.trim();
        String s = null;
        if (enrollS.length() < 8){
            s = "Invalid enrollment Number";
            return s;        }
        String yearS = enrollS.substring(0, 2);
        String courseS = enrollS.substring(3, 5);
        int courseID = Integer.parseInt(courseS);
        String EN = enrollS.substring(5, 8);
        yearS = "20" + yearS;
        String course = getCourse(courseID);
        s = "Hello ji.....\n" + "Your Enrollment Number is: " + EN + "\n" + "Your Branch is: " + course
        + "\n" + "Your Admission Year is: " + yearS;
        return s;    }
    public String getCourse(int courseID) {
        String s = null;
        switch (courseID) {
            case 35:
                s = "CSE";
                break;
            case 36:
                s = "IT";
                break;
            default:
                s = "NA";
                break;        }
        return s;    }
}
```

Appendix B

Ant Script : build.xml

B.1 Ant Script build.xml:

```
<project default="wsgen">
  <target name="wsgen">
    <exec executable="wsgen">
      <arg line="-cp ./bin -keep -s ./src -d ./bin
com.myfirst.wsServer.VijayServer"/>
    </exec>
  </target>
</project>
```

B.2 Java Class RunService:

```
package com.myfirst.wsServer;
import javax.xml.ws.Endpoint;
public class RunService {
    public static void main(String[] args) {
        System.out.println("Enrollment number web service started by Vijay");
        Endpoint.publish("http://localhost:8080/wsVijayExample",
            new VijayServer());
    }
}
```

Appendix C

Programs Code

C.1 Program Start:

Inbox parameters: `eninstart(String)`

Outbox parameters: `enoutstart(String)`

JAVA.SNIPPET:

```
If (eninstart.length(<8)
    enoutstart= "invalid enrollment";
else
    enoutstart=eninstart;
```

C.2 Program Validity:

Inbox parameters: `enin(String)`

Outbox parameters: `enout(String)`

JAVA.SNIPPET:

```
enout=enin;
```

C.3 Program getrollno:

Inbox parameters: `enrolls(String)`

Outbox parameters: `rollno(String)`

JAVA.SNIPPET:

```
rollno=enrolls.substring(5,8);
```

C.4 Program getyears:

Inbox parameters: `enrolls(String)`

Outbox parameters: `year(String)`

JAVA.SNIPPET:

```
year=enrolls.substring(0,2);  
year= "20"+year;
```

C.5 Program showoutput:

Inbox parameters: courseR(String)
rollnoR(String)
yearR(String)

Outbox parameters: s(String)

JAVA.SNIPPET:

```
S= "Hello ji"+ "\n"+  
"Your year of admission is:"+year +"\n"+  
"Your branch is:"+courser+"\n"+  
"Your roll no is:"+rollnoR;
```

Appendix D

WSDL File of single web service

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI
2.1.6 in JDK 6. -->
<!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI
2.1.6 in JDK 6. -->
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://wsServer.myfirst.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="VijayServerService"
targetNamespace="http://wsServer.myfirst.com/">
<types>
<xsd:schema>
<xsd:import namespace="http://wsServer.myfirst.com/"
schemaLocation="http://localhost:8080/wsvijay2Example?xsd=1"/>
</xsd:schema>
</types>
<message name="enrollDecipher">
<part element="tns:enrollDecipher" name="parameters"/>
</message>
<message name="enrollDecipherResponse">
<part element="tns:enrollDecipherResponse" name="parameters"/>
</message>
<message name="getCourse">
<part element="tns:getCourse" name="parameters"/>
</message>
<message name="getCourseResponse">
<part element="tns:getCourseResponse" name="parameters"/>
</message>
<portType name="VijayServer">
<operation name="enrollDecipher">
<input message="tns:enrollDecipher"/>
<output message="tns:enrollDecipherResponse"/>
</operation>
<operation name="getCourse">
<input message="tns:getCourse"/>
<output message="tns:getCourseResponse"/>
</operation>
</portType>
<binding name="VijayServerPortBinding" type="tns:VijayServer">
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="enrollDecipher">
```

```
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
<operation name="getCourse">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
</binding>
<service name="VijayServerService">
<port binding="tns:VijayServerPortBinding" name="VijayServerPort">
<soap:address location="http://localhost:8080/wsvijay2Example"/>
</port>
</service>
</definitions>
```