A Web Service Request Language

Gaurav Mitra

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Software Engineering at
The School of Computer Science
Australian National University

October 2009

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this thesis is my own original work.

Gaurav Mitra
30 October 2009

Acknowledgements

Firstly, I'd like to thank my supervisor Dr. Xuan Zhou for the sheer amount of time and effort that he has spent in helping me with this project. His insight and guidance have been very important as far as my contribution in this project is concerned. The original content and ideas presented in this thesis have largely been developed in collaboration with Xuan. I would also like to thank Dr. Athman Bouguettaya for his help in this project.

I'd like to thank all my friends and colleagues for providing me with their support and advice. In particular I'd like to thank Martin Bolanca for his support throughout the year, and Dr. Ashvin Parameswaran for giving me constructive feedback about the structure and content of this thesis.

I would like to thank my parents, Rupa and Debasis who have always given me tremendous emotional support. I couldn't have done it without them. Last but not the least, I would like to thank Arya for her love and encouragement in every step. I definitely couldn't have done it without her.

Abstract

The rapidly progressing paradigm of service oriented computing is realised by web services which have yet to reach their full potential in the e-marketplace. This project is an endeavour to bring web services and service oriented computing closer to clients and businesses alike in terms of accessibility and composability. We have created a web service request language based on which clients will be able to specify their service request/requirements in a clear and unambiguous manner. This language is accompanied by a design of the request system that shall process the service request, generate a service execution plan, invoke services according to the plan and provide the results to the user.

To address the issues related to automatic composition of web services we take an approach derived from graph theory and finite state machines to create the concepts of a service variable, a user state and a request oriented service model. This model is then used to capture dynamic information about services and their capabilities in terms of service variables. The request language is designed to express user requests in terms of these service variables.

Algorithms have been developed to map user requests to states in the user state graph, extract the service execution plan from the graph and to execute the plan. The thesis concentrates on developing approaches that are feasible to implement in real-world scenarios minimising time complexities for algorithms developed. Many future possibilities in regards to this relatively new way of modelling web services from a service request point of view, have also been discussed.

Contents

List of Figures

List of Definitions

List of Algorithms

Chapter 1

An Introduction

An emerging trend in the Internet today is that of e-marketplaces. Companies are marketing their products via the Internet in a very large scale manner. The range of products available via the Internet is also diverse. Internet marketing has already become mainstream. As a result, the customer-vendor interface is usually very well defined in the form of internet websites which provide customers with all possible and intuitive options for specifying their business needs.But what happens when instead of a human customer, another business application (or a computer) tries to acquire products via the internet ? The business application could be trying to conduct this transaction on behalf of its own customers. The answer to this question lies in service oriented computing.

Products on the Internet are also being provided as *services* to the customer. For e.g. if a customer purchases an airline ticket from an online travel agency, the travel agency has just provided the customer with a service. This conceptual framework of a service has very general connotations. Any abstract product may be represented as a service. Even a request for information about a product may be thought of as a service. Multiple services together may be composed or orchestrated to form a single service.

This way of thinking gives us another layer of abstraction and modularity over any type of product or resource that can be provided via a network. It provides us with a way of conceptualising abstract products as loosely coupled components that may be reused to form larger components. It introduces a component based architecture to the Internet marketplace. What it also does, is provide a platform on the basis of which computers (or applications) may interact with other computers to obtain data or results. Hence, services could in a way be thought of as classical AI intelligence agents.

In this service oriented computing (SOC) paradigm, customers should expect a much better Internet marketing experience. In fact, all they should have to do is specify what they want through a standardised interface, or via a standardised language and expect to receive their desired products or results. In order for this to be realised, a system must be created that can accept such requests from users, interact with necessary services, obtain a result and deliver it to the user. As SOC is still in stages of

infancy with respect to the internet marketplace, such a system has not yet been conceptualised and created. The goal of this project is to conceptualise and design such a system.

The fundamental requirement for such a request system would be a dedicated service request language that may be used to generate service requests based on the users requirements. An important thing to understand is that a user's request may span several services. By this, we mean that a user may want a result that can only be delivered by a composition of multiple services. This composition is better described as a business process which conducts a business transaction with a customer. Current efforts at service composition are highlighted by the Business Process Execution Language or BPEL [Andrews 2003]. It is used for pre-defining these business transactions and processes into BPEL specifications that do not allow the customer the flexibility to provide a request that requires a business transaction that is undefined. To satisfy such a request, another BPEL specification has to be recreated manually. The Web Service Choreography Description Language or WS-CDL [Pelz 2003] has also been created to express an ordering of services belonging to different business organisations, but it also relies on a predefined or known composition of services.

Let us now consider an example to understand service composition. A customer wants to buy a book, *The Lost Symbol* by *Dan Brown* and wants to pay less than $20 for it via credit card. He describes his request in the request language and sends it to our system. Based on the customers requirements, our system determines that it needs to get information about the book, and its exact price (which must be less than $20) from a book-seller service first. The system then needs to bill the customers credit card by invoking the credit card billing service with the exact price of the book (assuming that the previous step provided the price information). Provided that the customer's credit card details are registered in the request system and that the card is valid, the billing service returns a receipt to our system.

At this point, a delivery service must be invoked, that physically delivers the book from the book-seller to the customer's address. The payment receipt is given to the customer by the system and the delivery service eventually gets the book to the hands of the customer. In this case we clearly see that the customer is only interacting with the request system and is not aware of the services that are invoked by the system to deliver the desired result. Such a system that gives a normal user access to any number of web services, can be safely described as a step forward towards the success of SOC. Such a system provides users and machines alike, a single & new entry point into the web services domain. Figure 1.1 shows clearly the sequence in which these events happen.

One might argue that companies on the internet today provide websites that can achieve this transaction for a customer, so why do we need another system for this ? The answer to that question lies in the fact that companies that provide such transac-

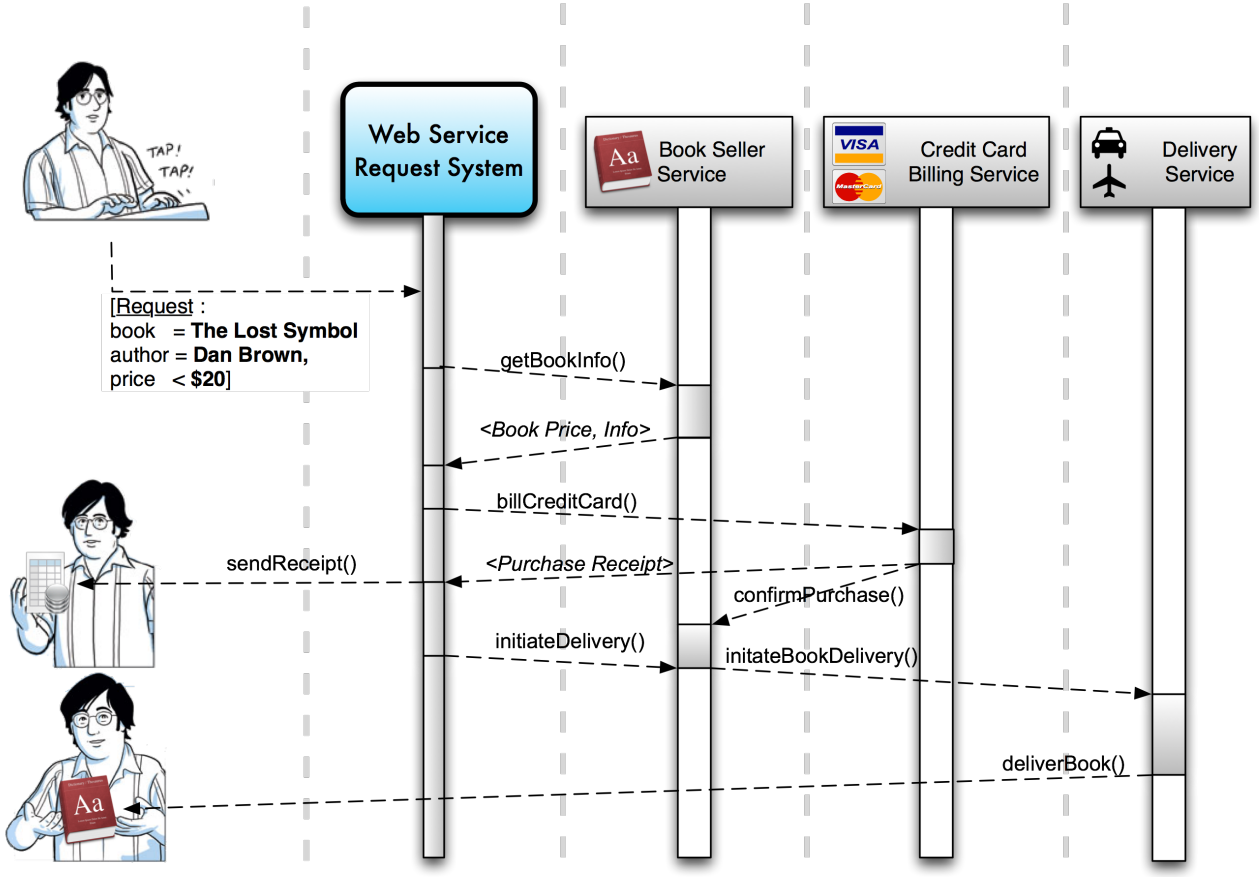

Figure 1.1: Purchasing A Book Via The Request System

tions today are focused on their own products and therefore can only deliver a certain range of products. This means that customers need to go to several different websites to purchase different types of products. The request system that we conceptualise in this thesis, solves that problem. It provides a single entry point for a customer to any product on the internet, provided that the service that delivers that product is registered in the system.

The business case for such a request system must be considered as it shows exactly how much potential such a system has. If this system were to be implemented, tested and made road ready, it would be usable by any organisation or company that required deployment of their products in the form of web services. Multiple companies could collaborate and use a single request system to provide all their services in the internet marketplace. The current case of internet websites dedicated to single businesses, would become inherently obsolete. In the case of business to business transactions, this system would also deliver well, because any service request, whether written by a human or a machine would be treated equally by the system. By having a

standard request language, businesses also gain the potential to automate the request specification stage. When that is achieved, we potentially will have implementations of classical AI agents who provide services to customers and are fully automated.

What the system has achieved in the book purchase example is an automatic composition of 3 separate services, namely the book-seller, the credit card billing and the delivery service. It has generated an ordering of service execution that defines the sequence in which services must be invoked and what information needs to be given to each service, along with what information is returned by each service. This ordering is called a service execution plan and it essentially represents a business process. The system then executes this plan to actually invoke the services and complete the business transaction. We can now identify the exact parts of such a request system. The system would need an internal *model* for representation of information regarding web services and requests made by users. A dedicated request *language* based on this model would be essential, and a *framework* to process the requests and deliver results to users would be mandatory.

To take a step forward in realising this vision of SOC, we introduce the Web Service Request System (WSRS) and its associated language, the Web Service Request Language (WSRL) in this thesis. We create and formalise a request oriented model for web services that describes web services from the view point of service requests. We also define the processes for creating instances of the model, generating a service execution plan, and executing the plan. We also provide algorithms that are part of these processes as well as the ones that show how the service execution process and service execution plan generation process operate.

Previous attempts have been made to create a request language and an associated request system. The most important ones being the XML Service Request Language [Papazoglou et al. 2002a] and the Framework for Web Service Query Algebra and Optimisation [Bouguettaya and Yu 2008]. However, they have implemented certain parts of a request system, and have not concentrated on conceptualising and entire system as such. We shall elaborate about them in *Chapter 3*.

The primary motivation for this project as we have mentioned before, was to further the cause of web services and SOC. We therefore started off this project with the idea of only creating a request language for web services and realised along the way that the system associated with handling the requests specified in the language, was far more important and had not been designed or implemented yet. We therefore set out to design the entire concept of a web service request system and the processes, algorithms and language associated with it.

1.1 Project Goals & Scope

The primary goal of this project is to clearly conceptualise all parts of the request system. This goal also comprises the following sub-goals :

1. **Modelling :** This is the most important goal as far as the system is concerned. Clear and well defined models need to be created for describing the services and the users w.r.t the request system.
2. **Creation Of The Language :** A formalised language needs to be created based on the models. The users of the system shall express their service requests using this language.
3. **Processes :** The processes involved in creation of model instances, generation of execution plans, and execution of these plans must be clearly defined.
4. **Algorithms :** The algorithms used in these processes must be clearly defined and their time complexities specified.
5. **Prototype Implementation :** A proof of concept implementation must be created to demonstrate the advantages of the approach used by this project.

The conceptual objectives of this project are as follows :

- Incorporate practical graph based techniques into approach, similar to the ones described in [Bouguettaya and Yu 2008].
- Focus on modelling the service space, the user space and the problem space as the models are integral to the practicality of the approach.
- Focus on pre-computing options that enable the system to reuse planning options during runtime and therefore reduce time to obtain results.
- Enable the user to express his request based on service functionality, capability and quality.

The scope of this project primarily includes designing the request system so that future work may concentrate on implementation and optimisation. A high level requirements specification for this project is provided in Appendix A.

Conventions Followed

A few conventions have been adhered to in this thesis. These are :

- We use the term *Service* and *Web Service* interchangeably. Although we agree that there is a distinct difference between the two, for reasons of brevity and concise expression we abuse notation throughout this thesis.
- We also use the Online Travel Agency (OTA) example which we elaborate in Chapter 4, to demonstrate new concepts throughout the rest of this thesis. We use this example to honour the convention followed by numerous published papers in the web services domain, that use this particular example to demonstrate their ideas.

1.2 Contribution Summary

Summarising exactly what we have achieved in the course of this project, or rather our exact contribution present in this thesis, we have :

- **The Web Service Request Oriented Model :** Used to describe web services and the users of web services from the view point of service requests. There are two parts of this model :
  - *The User State Model :* Describes the user's requirements with respect to web services.
  - *The Service Description Model :* Describes web services with respect to user requests.
- **The Web Service Request Language :** Allows users to easily express service requests spanning multiple services.
- **The Web Service Request Framework :** Contains processes and algorithms to accept the service request, generate the service execution plan and execute the plan.

Efforts were made to finish the prototype implementation for the system, but due to restrictions in time it was not fully achieved. The exact achievements in this regard are as follows :

- **An XML Request Parser :** This module (written in Java) takes an XML file containing web service requests written in the web service request language (WSRL), parses the file and creates a web service request as stated in *Definition 7*, using the request encoding process as stated in *Algorithm 2*.
- **Object Oriented representations of each concept :** All concepts introduced in the request oriented model have been implemented as java classes with their required attributes.

1.3 Thesis Structure

The rest of this thesis is organised as follows :

- **Chapter 2 :** In this chapter we provide a brief introduction to the fundamental concepts of service oriented computing and the technologies involved in the web services domain.
- **Chapter 3 :** We discuss the previous attempts at creating a request system i.e. XSRL and the Query Framework in this chapter.
- **Chapter 4 :** We provide a detailed service domain example of the Online Travel Agency that we use to demonstrate concepts in the rest of the thesis.
- **Chapter 5 :** We introduce, describe and define all the parts of the web service request system that we have designed.
- **Chapter 6 :** We discuss future work possibilities regarding the request system and provide a conclusion for our work in this thesis.

As a note to readers of this thesis, we would like to mention that *Chapter 2* is purely based on describing all the technologies required to implement and use web services in general, along with ideas of service oriented and cloud computing. Therefore, if readers are well versed in these concepts, they may skip this chapter. However we would recommend reading it as it leads up to eliciting needs or requirements for the request system.

1.4 Research Plan

This project was carried out adhering to the following research plan :

1. Investigate all technologies, standards concepts relating to web services.
2. Try to find related publications that attempt to create or describe a similar system or parts of it.
3. Understand what has already been done with regards to the system.
4. Properly create a finite scope for the project and clearly define what is not involved.
5. Define precise goals of the project.
6. Collaborate with supervisor to create new material that fulfil the goals of the project.
7. Test whether the new material is feasible using examples.
8. Write thesis about work done on project.

Chapter 2

Background

2.1 Service Oriented Computing (SOC)

An emerging trend today is that of service oriented computing or SOC. It is an attempt to provide a framework that gives Internet business providers the ability to encapsulate their products into a self-describing service. The description of this service can then be published in a common registry. Customers that want a desired product or functionality may then obtain the relevant service description from the registry and invoke it. [Papazoglou 2008] describes a service as a self-contained module - deployed over standard middle-ware platforms - that can be described, published, located, orchestrated, and programmed using XML-based technologies over a network. This creates a plethora of possibilities for existing computing resources to be reused into this new paradigm of *software as a service*.

The functionality of a service can range from a simple request for information to a complex business transaction. Therefore this generic abstraction of services can be mapped to a vast number of business domains making it widely adoptable. In a way, SOC enforces a component based architecture to any business domain where components may be loosely coupled and highly modular. This greatly increases the replaceability factor of business functionality units. Considering the vast range of businesses that provide similar functionality over the Internet, SOC provides a means of selecting the ones that are most appropriate for the consumer.

Another promising aspect of SOC is the fact that it isn't a completely new technology. It is an amalgamation of various existing principles in computing including distributed systems, software engineering principles, computer networks, common Internet standards such as XML and HTTP, web applications etc. [Papazoglou and Georgakapoulos 2003] says that the value of an application is actually no longer measured by its functional capabilities but rather by its ability to integrate with its surrounding environment. SOC capitalises on this fact and ensures that services prioritise on collaborating with other services.

2.1.1 SOC : The Future

The Internet comprises of a vast number of systems and applications that offer a diverse range of functionalities. These systems have been created mostly using different architectural styles and technologies in isolation without contemplating their possible use in conjunction with other systems. At present if we need to obtain an artefact or result, the path to obtain that result might span a set of different functionalities offered by different business providers, our only option is to create a new system that delivers the result rather than reusing present resources due to disparity in the way the resources have been created initially.

One of the most important concepts in software engineering is that of code or software reuse. While it can be generally applied in small systems today, it cannot be applied to large scale business domains. SOC provides the solution to this problem. It provides a medium for disparate systems or businesses to interact and deliver a compound result. It enables software reuse at a very large scale and that is exactly why it is widely considered as the future of internet business.

2.1.2 Moving to the cloud

Putting things in a broader perspective, the cloud computing paradigm fits well with the idea of *Software As A Service* or *Platform As A Service*. [Buyya et al. 2008] defines a cloud as a type of parallel and distributed system consisting of a collection of interconnected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumer.

Cloud computing covers both software clouds and hardware clouds alike and several companies have recently started providing cloud computing facilities. For e.g. the Amazon Elastic Compute Cloud [Amazon 2008] and the Microsoft Live Mesh [Microsoft 2008] are clouds that provide computing and storage facilities and are therefore hardware clouds, whereas the Google App Engine [Google 2008] provides web application hosting facilities and is therefore a software cloud.

Therefore, we see that the focus on computing utilities is shifting (or has almost shifted) from individual computers to computing clouds. There are several reasons for this massive change. First of all, the internet is growing at an exponential rate and is widely available now. Access to the internet is no longer a privilege for the average computer user. Most importantly, computers and essential services offered by computers such as communication, media, information , utility etc. have now become as much a part of our lives as gas and electricity.

As a result, consumers also require access to these services regardless of their location in the world, as long as they have access to the internet. A combination of these factors has shifted the scale towards service oriented computing in the cloud.

2.1.3 SOC Implementation : Services in the cloud

The best way to demonstrate these concepts is by an example. We shall consider an online travel agency (OTA, a classic example used to demonstrate the utility of SOC). It may offer several services to its customers including flight booking, car rental, hotel booking or the entire travel package. Now, for a business to create such an online travel agency from scratch, it would have to create the entire infrastructure from scratch and make deals with a set of other businesses that may provide necessary functions such as flight booking, car rental, hotel booking etc. With the help of SOC, this process of making deals with other businesses could be immensely simplified and made more dynamic.

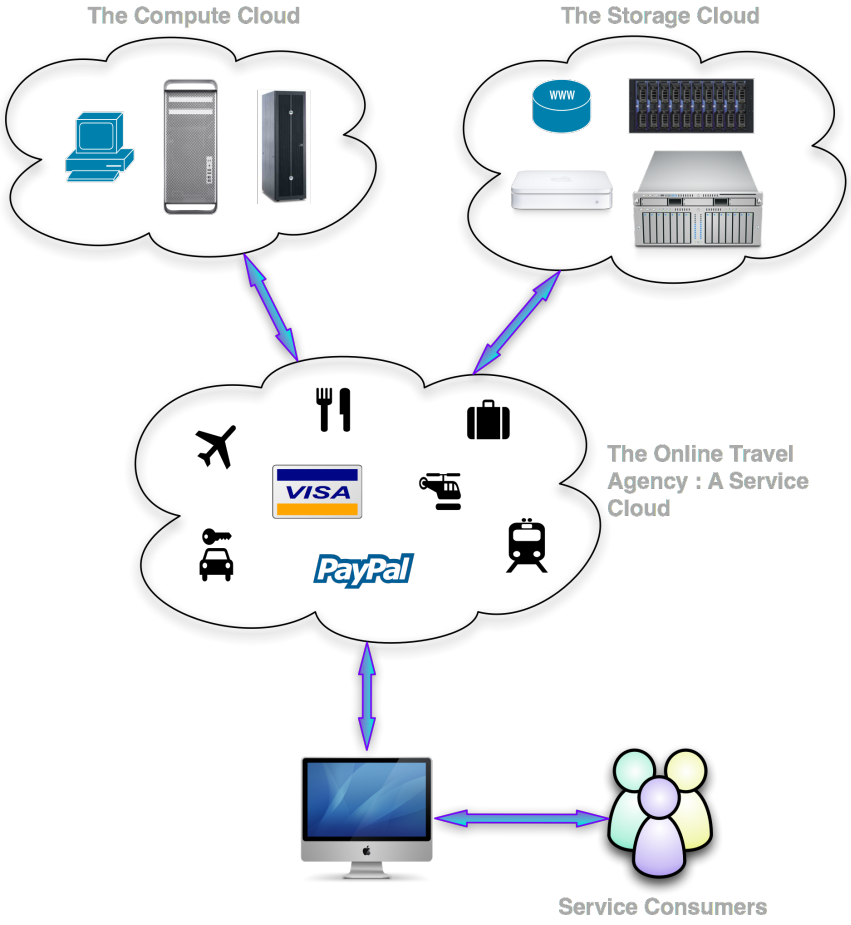

Figure 2.1: A Service Cloud Example

Each of these businesses could publish its functionality as a service description in a service registry. This registry can be visualised as a tracker for available services. When the OTA would receive a request from a customer to book a certain flight for less than a certain amount of money, it would merely query the service registry for

the most appropriate flight booking service available at the time, offering the desired rates, and invoke that service to obtain the flight booking which it would then forward to the customer.

In this example, all the individual services (flight booking, credit card etc) that have their descriptions published in the service registry, form a cloud of services. The OTA is merely demarcating a subset of the cloud and offering customers an aggregated service i.e. the entire holiday package. What this facilitates is exactly why SOC is very important, business to business interaction and software reuse.

Figure 2.1 appropriately describes this example. Here we define the customer as a service consumer, the OTA as a service aggregator that acts as a service cloud, and the organisations providing the services that are part of the cloud as service providers. These three roles are fundamental to SOC and compose the service oriented architecture or SOA.

The figure also shows that the service cloud utilises the compute and storage clouds offered by other organisations. This gives the service providers the freedom of not worrying about hosting their software implementation of their services on their own servers. This abstraction allows service providers the privilege to concentrate on their business logic rather than hosting issues, and therefore provide better quality of service to consumers.

2.2 Web Services

SOC is an architectural concept and is independent of technology as such. To realise SOC, the web service [Alonso et al. 2003] technology is used. Web services allow the presentation of service functionalities in an accurate way. [Papazoglou 2008] defines a web service as a self-describing, self-contained software module available via a network, such as the Internet, which completes tasks, solves problems, or conducts transactions on behalf of a user or application.

Web services can provide any sort of functionality, static or dynamic. They provide a black box abstraction for services i.e. all implementation details are hidden from the consumer who only provides inputs and receives an output or result. Web services provide a platform independent way of accessing functionalities using interoperable and widely established protocols such as XML and HTTP. They can also reuse other web services to provide results, as per SOC principles.

It is important that we now state the exact differences between a *web service* and a *service*. A service is an implementation of some functionality that can be used to produce artefacts or results. Customers pay money to obtain these results. A Web Service

is mainly a description of one service or multiple services that outlines exactly what functionality is provided by the service/s. It also defines the Quality Of Service (QoS) that provides important information (for the customer) such as price, security, reliability, availability etc. of the service. Finally a web service describes exactly how a service can be invoked, including the network address, protocols and parameters required for invocation. Summing up, a web service is a semantic wrapper for a service that describes what the service does and how we can use it. Figure 2.2 gives an example of this difference.

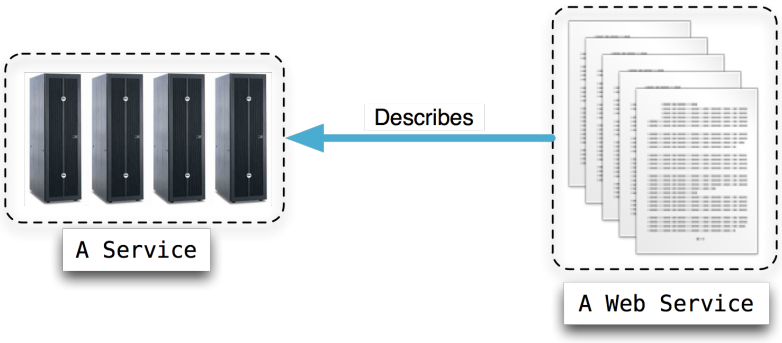

Figure 2.2: The Difference Between A Service And A Web service

Web services come in different flavours. [Papazoglou 2008] classifies web service in two categories :

1. *Simple/Informational :* Support simple request/response operations
2. *Complex :* Implement some coordination between inbound and outbound operations

We can decompose each of these categories into subcategories and describe them by means of an example.

Simple/Informational services

1. *Pure content services :* A weather report information service
2. *Simple trading services :* Business logistic services e.g. an inventory report service
3. *Information syndication services :* A value added commerce service e.g. a rating service

Complex services

1. *Complex services that compose simple web services :* inventory checking service
2. *Complex services that compose interactive web services :* An online travel agency service

We shall concentrate on such complex services that are compositions of other interactive or simple services, throughout this thesis.

Functional & Non-Functional Characteristics

The operational properties of a web service are known as its functional properties. For e.g. for a flight booking service, the ability to book a flight is a functional property of the service. Similarly, the non-functional properties of a web service are those that are based on its quality attributes such as cost, response time, availability etc. These properties determine which service is chosen by a consumer from a list of services with similar functional characteristics.

2.3 Building Blocks : The Web Service Technology Stack

The fundamental technologies required to implement and use web services can be best described in the form of a stack shown in Figure 2.3.

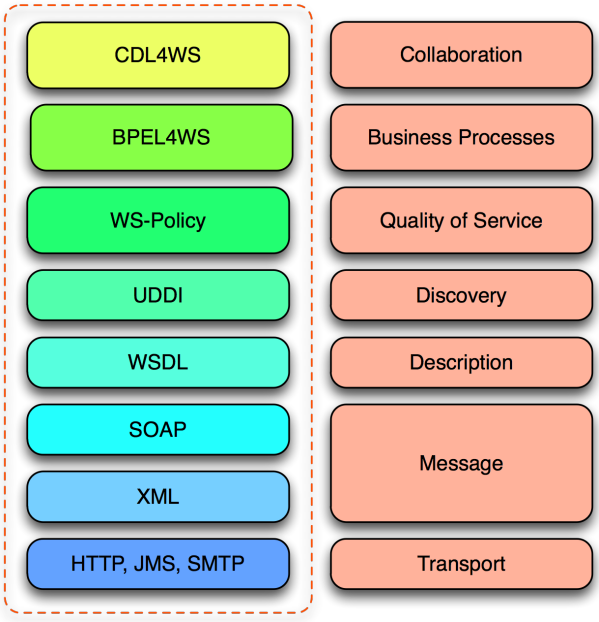

Figure 2.3: The Web Service Technology Stack

We shall now describe each level in the stack starting from bottom up.

2.3.1 The Transport Layer

Common protocols used to transfer data over the internet such as the hyper-text transfer protocol (HTTP) are part of the transport layer. Although web services do not specifically tie themselves to any protocol, they build on the commonly used ones such as HTTP, to ensure wide reachability and support.

2.3.2 The Message Layer

We can only interact with web services (i.e. invoke them) via messages that are written in XML and are defined by the simple object access protocol (SOAP).

2.3.2.1 XML

All web service technologies are based on XML or the Extensible Markup Language. XML is a w3c (World Wide Web Consortium) standardised protocol for the description and delivery of marked up electronic text over the Internet. XML stands out compared to other mark up languages due its portability. The same character encoding scheme is used for all XML documents. Namespaces are used in XML to distinguish between local and global tags and elements. Also, an XML Schema is used to define the content and structure of a class of XML documents.

2.3.2.2 SOAP

A messaging protocol for exchanging information between web services is a fundamental requirement. SOAP [W3C 2003] addresses that requirement. It is an XML based communication protocol meant for exchanging messages between computers. It is independent of operating system or programming environment . HTTP is most commonly used as its medium of transport, but other protocols such as SMTP,FTP or RMI may also be used.

SOAP supports both one way and two way messaging between web services. A SOAP client is used to encode an XML message for dispatching and a SOAP server receives this message, decodes it to programming language specific objects required and delivers it to the receiving web service. Both the sender and the receiver must have the same XML schema's to process the SOAP information accurately.

A simple SOAP message may look like this [W3C]:

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
  <soapenv:Header>
    <wsa:ReplyTo>
      <wsa:Address>http://schemas.xmlsoap.org/ws/2004/08/
            addressing/role/anonymous</wsa:Address>
```

```
    </wsa:ReplyTo>
    <wsa:From>
      <wsa:Address>http://localhost:8080/axis2/services/
                  MyService</wsa:Address>
    </wsa:From>
    <wsa:MessageID>ECE5B3F187F29D28BC11433905662036
    </wsa:MessageID>
  </soapenv:Header>
  <soapenv:Body>
    <req:echo xmlns:req="http://localhost:8080/
                  axis2/services/MyService/">
      <req:category>classifieds</req:category>
    </req:echo>
  </soapenv:Body>
</soapenv:Envelope>
```

The three main parts of a SOAP message as demonstrated by this example are :

1. <**Envelope**> : This is a container element for the soap message and shows what is in a message and how it can be processed.

2. <**Header**> : The header contain information about destination and origin of the message.

3. <**Body**> : This is the message payload or application specific data that this message carries.

2.3.3 The Description Layer

For web services to be machine consumable i.e. to effect business to business transactions as per the SOC vision, a standardised description language must be used to create descriptions of web services. These descriptions must also be self contained i.e. a consumer or a machine must not require any extra information to interpret the functionality, invocation details of a web service. To address this need, the Web Service Description Language or WSDL [Christensen et al. 2001] was created and standardised by the w3c.

2.3.3.1 WSDL

WSDL is also based on XML and is used to define a public interface for a web service that acts as a binding contract between a service provider and a client. Conforming with SOC, WSDL is also platform and language independent. In essence, a WSDL description of a web service answers three basic questions :

1. What functions does the web service provide ?

2. Where can the web service be found on the Internet ?

3. How can the web service be invoked ?

There are two parts to a WSDL specification of a web service :

1. *The service interface definition :* This abstract definition of a web service is used to define the operations supported by a web service and the messages and data types used in these operations
2. *The service implementation definition :* This concrete definition of a web service is used to define the network location and invocation details of a web service.

We now explain parts of the interface and implementation description via an example of a simple web service that provides stock quotes [W3C 2001] :

The Interface Description

The *StockQuoteService* WSDL interface description is given below :

```
<?xml version="1.0"?>
<definitions name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
       <schema targetNamespace="http://example.com/stockquote.xsd"
              xmlns="http://www.w3.org/2000/10/XMLSchema">
           <element name="TradePriceRequest">
              <complexType>
                  <all>
                      <element name="tickerSymbol" type="string"/>
                  </all>
              </complexType>
           </element>
           <element name="TradePrice">
              <complexType>
                  <all>
                      <element name="price" type="float"/>
                  </all>
              </complexType>
           </element>
       </schema>
    </types>

    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>
    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd1:TradePrice"/>
    </message>

    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
```

```
            <input message="tns:GetLastTradePriceInput"/>
            <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>
</definitions>
```

The description begins with a declaration of XML namespaces that are used and that the default namespace is of WSDL found at *http://schemas.xmlsoap.org/wsdl/*. We now explain each part of an interface description citing bits of the example above.

- <**types**> : This section describes all the abstract data types that are used to describe artefacts of a web service. In this particular example, we see that there are two abstract data types that have been defined, *TradePriceRequest* which has an attribute 'tickerSymbol' of type 'string' and *TradePrice* which also has a single attribute 'price' of type 'float'.

- <**message**> : Each message element describes an input or output message that may be sent or received respectively, from a web service. In this example, two messages have been defined, *GetLastTradePriceInput* which contains its data or payload of type TradePriceRequest and *GetLastTradePriceOutput* which must have data of type TradePrice.

- <**portType**> : This element is a container for the operations supported by a web service. It can be compared to a Java interface which has method declarations. This is the most important element of a web service description and the rest of the elements are usually required to define parts of this element. In this example we see the *StockQuotePortType* which has an operation named 'GetLastTradePrice'.

- <**operation**> : An operation element contains only one *input* message and one *output* message. A web service is invoked when an input message is sent to one of its operations and it returns a response to the invocation by delivering an output message. It may also deliver a *fault* message that describes an error that might have occurred during execution of the operation. In our example we see that the *GetLastTradePrice* operation is described with its input and output messages. A fault message is not specified here.

The Implementation Description

The *StockQuoteService* WSDL Implementation description is given below [W3C 2001]:

```
<?xml version="1.0"?>
<definitions name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
```

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="GetLastTradePrice">
  <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
     <input>
       <soap:body use="encoded" namespace="http://example.com/stockquote"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
     </input>
     <output>
       <soap:body use="encoded" namespace="http://example.com/stockquote"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
     </output>
 </operation>
</binding>

<service name="StockQuoteService">
      <documentation>The Stock Quote Service</documentation>
     <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
        <soap:address location="http://example.com/stockquote"/>
     </port>
</service>
</definitions>
```

Yet again we find the namespace declarations beginning the description. We now explain each part of the implementation description citing bits of the example above.

- <**binding**> : This is the most important element of the implementation description. It is used to describe how the customer and the web service should exchange messages. Each binding element maps to one portType element of the interface description. It can be visualised as a Java class that implements a java interface interface and provides the implementations for the operations declared in the interface. In this example we see that the *StockQuoteSoapBinding* binds to the *StockQuotePortType*. It mentions that a 'document' style SOAP message must be transported via HTTP to the 'GetLastTradePrice' operation available at *http://example.com/GetLastTradePrice/*. The input and output types here represent the SOAP encoding styles and namespace to be used for the input and output data types defined in the interface description.

- <**port**> : This element provides a combination of the network address at which a web service is available and its corresponding binding information to its abstract description. Here we see that there is only one port element *StockQuotePort* which specifies that the *StockQuoteService* is available at *http://example.com/stockquote* and the operation provided at this port of the service is defined by the portType *StockQuotePortType* as the *StockQuoteSoapBinding*, specified here, binds to that port type.

- <**service**> : This element is composed of one or more port elements and has a unique name. In this example it is *StockQuoteService*. The ports of a service

element therefore provide the actual operations to consumers.

2.3.3.2 The Separation between Interface and Implementation

This is a very important facet of web services. The abstraction provided by allows the two parts of a web service description to be developed separately. The interface part can possibly be developed by an organisation that wants to act as a manager of multiple services. A simple example would be the Online Travel Agency (OTA). It would have to develop interfaces for the airline booking service, the railway booking service, the credit card service and so on. Different service providers could in turn receive these interface definitions and provide their own implementation definitions to the OTA.

What this achieves is more choice for the consumer. The types of services provided by the OTA would be a few, but the services that implement these few types could be many. The service providers could choose to implement their service types in whatever way required, whether by implementing them entirely, or outsourcing parts to other providers.

2.3.4 The Discovery Layer

This layer defines the entry point for a service consumer to the web service domain. Consumers can only invoke or utilise web services when they have WSDL descriptions. All consumers also have specific needs that they need to fulfil via web services and therefore they need to find services based on these specific needs. The service discovery layer provides consumers with the means to query a database of service descriptions and find the right match. This layer provides service registries such as the UDDI.

2.3.4.1 UDDI

The Universal Description, Discovery and Integration or UDDI [Bellwood et al. 2002] provides an interface that service clients can use to dynamically search for desired web services (WSDL Descriptions) using keywords that may describe the service in some way. Service providers are required to register their WSDL descriptions for their services with UDDI beforehand. UDDI is also based on SOAP,XML and all requests and responses to and from UDDI are in the form of SOAP messages.

2.3.5 The Quality of Service Layer

The important part in the WS technology stack from a service clients perspective is the Quality of Service or QoS. It reflects the ability of a web service to perform and deliver according to pre-determined expectations. Parameters of QoS cover both functional and non-functional service quality attributes. Based on the QoS properties a Service Level Agreement or SLA [Jin et al. 2002] is drawn up between the client and

the provider, which acts as a QoS formal agreement or guarantee. It reflects the expectations of both the client and the provider in the form of a contract.

2.3.5.1 QoS Attributes

[Mani and Nagarajan 2002] provides a comprehensive description of QoS attributes :

- *Availability :* It can be described as the probability of the service being available at a certain time. The absence of service downtimes increases this probability. Another measurement associated with availability is time to repair or TTR which is the average time required to repair the service after it has gone down.
- *Reliability :* It represents the ability of a service to deliver the correct functionality in a consistent manner. It also represents the service's dedication to delivering the same quality despite interruptions such as a network failure.
- *Security :* Aspects such as authentication, authorisation, message integrity and confidentiality are part of security.
- *Accessibility :* It represents the degree with which the web service request is served. High availability shows that the web service can serve many clients at a time.
- *Integrity :* It describes the conformance of the web service with its WSDL description and the SLA agreement.
- *Conformance to Standards :* Describes the compliance of the web service with standards and versions of standards mentioned in the SLA.
- *Performance :* Performance is measured in terms of
  - *Throughput :* Number of web service requests served in a given time period. Higher values represent good performance.
  - *Latency :* Time elapsed between a request to and a response from a web service. Lower values represent good performance.
- *Scalability :* Describes the ability to consistently serve requests independent of the volume of incoming requests.
- *Transactionality :* Describes the degree of transactional behaviour demonstrated by the web service.

2.3.5.2 Service Level Agreement (SLA)

An SLA on the other hand formalises the client-provider details associated with a web service such as price, delivery process, acceptance, quality criteria, penalties etc. [Jin et al. 2002] decomposes an SLA into the following parts :

- *Purpose :* The reason behind the creation of the SLA.
- *Parties :* Describes the client and provider of the web service.
- *Validity Period :* Details when the SLA will expire.
- *Scope :* Details the web services covered by the SLA.
- *Restrictions :* Describes any necessary constraints on the services.
- *Service-level Objectives :* Describes the service level QoS objectives that need to be upheld by the service.
- *Penalties :* Describes consequences of the service provider failing to provide the service according to terms in SLA.
- *Optional Services :* Details any optional services that may be required in the process.
- *Exclusion Terms :* Details what is not covered in the SLA.
- *Administration :* Describes the organisational authority that may alter the SLA.

SLA's can be :

- *Static :* Remains unchanged for multiple service transactions.
- *Dynamic.:* May change from transaction to transaction.

SLA's and QoS attributes are expressed using the Web Service Policy Framework (WS-Policy) [Bajaj 2006].

2.3.6 The Business Processes Layer

As we have mentioned before, web services are either simple or complex. A complex web service delivers a result that might require participation of several simple web services. However, when we talk of a business transaction between a company and a consumer, we cannot always model the entire business as a complex web service. Such a transaction may be composed of several complex services at different points in the transaction. Such an explicit sequencing in web services that achieves a purely business goal is defined as a business process. The Business Process Execution Language or BPEL was created to model such business processes.

2.3.6.1 BPEL4WS : Web Service Orchestration

BPEL4WS or BPEL in short is an XML based flow language for the formal specification of business processes and interaction protocols. BPEL is also used as a means of web service orchestration which describes how the web services interact at a message level, including the business logic and execution order of the interactions from a single

service's perspective [Pelz 2003]. Two other protocols exist that assist in the orchestration of services. They are WS-Coordination [Cabrera 2005b] and WS-Transaction [Cabrera 2005a]. They complement BPEL to provide standard protocols for use in orchestration.

2.3.7 The Collaboration Layer

Moving on to a higher layer of abstraction we find that care needs to be taken to define exactly how business processes that belong to different organisations interact with each other. The collaboration between different business organisations to provide a an ordering in business processes (and in turn services) can be broadly termed as choreography between the services offered by the organisations. The global communication between these processes needs to be well defined, in order for a deterministic result each time. For this purpose, the Web Service Choreography Description Language was created.

2.3.7.1 CDL4WS : Web Service Choreography

CDL4WS or WS-CDL or the Web Service Choreography Description Language is used to define global communication between services from different business organisations, their rules of interaction, and the binding contracts that form between multiple business processes. WS-CDL can be used to track all the communication and transaction between all the parties involved in the business transaction, such as the client, the partners, suppliers, distributors, providers, etc. It offers a medium through which all the rules and agreements between all the collaborators can be accurately laid out and agreed upon [Pelz 2003]. Once a WS-CDL specification is approved, it can be used to generate BPEL4WS [Andrews 2003] work-flows for all parties involved which in turn can be used to execute the business transactions.

2.4 Web Service Composition

Now that we have enough information about how an atomic (simple) web service is described and defined, we move on to understanding how multiple atomic services can be composed to form a complex web service or a business process. We have seen that BPEL is focused on describing the execution of web services where the actual composition or the sequencing of the services are known before hand i.e. before the BPEL execution plan is created. WS-CDL defines the global message exchange between services or processes that belong to different organisations. Again, CDL depends on the fact that this service interaction behaviour between organisations is well defined before a CDL specification can be created. The challenge lies in automating this composition process and generating CDL and BPEL specifications based on the automatically generated composition.

Automated web service composition is the most important hurdle that needs to be overcome for the eventual success of SOC. As we have described previously, web services have a component based architecture that eases service reuse. A complex or compound business goal could be composed of several different business activities spanning several organisations. Web services that encapsulate these business activities are offered by these organisations. Therefore, to achieve this business goal, multiple services need to be composed into a single business transaction.

What we need to understand here is that the technologies specified in the stack, do not provide any scope for automatic or semi-automatic service composition techniques. To achieve automatic composition, a *semantic description* of each service is required that provides information about what pre and post conditions are involved with respect to a web service being invoked. Efforts have been made by the semantic web services community to create these semantic descriptions in the form of ontologies.

2.4.1 Service Semantics : The Ontology Approach

An Ontology is a representation of knowledge in general and is used to define relationships between separate entities. Ontologies when applied to web services, are described best as meta-models for web services that provide meta-data information i.e. the service semantics. The vision of the semantic web by Tim Berners Lee (inventor of the internet), is a vision of the future of the web in which information has machine understandable meaning.

The semantic web is based on XML and the Resource Description Framework or RDF [W3C 2004] which is used to describe relations between URI's (Universal Resource Identifiers, an e.g. is a URL of any web page). A level above RDF, we have the Web Ontology Language or OWL [McGuinness and Harmelen 2004]. OWL is used to formally describe the meaning of terminology used in web pages and documents. A brand of OWL is OWL-S, which is OWL for Services that provides machine understandable meaning for terminology related to web services. Another such ontology for services is the Web Service Modelling Ontology or WSMO [Roman et al. 2005].

2.4.1.1 OWL-S

OWL-S [Martin et al. 2004] is primarily designed to provide semantic markup for web services. It provides a basis for automatic service discovery, invocation and composition. It provides an upper ontology for services that has the service profile (the interface), the service grounding (the implementation) and the service model which describes the parts of the service itself. The service profile also contains information describing the functionality of a service such as the inputs required, the output generated, the preconditions for the service and the result of the service being invoked.

2.4.1.2 WSMO

Similar to OWL-S, the web service modelling ontology provides conceptual and formal constructs for semantically describing all relevant aspects of web services. [Roman et al. 2005] describes the 4 main elements of WSMO as :

1. *Ontologies :* Provides description of the terminology used by the other WSMO elements. Used to describe the relevant and syntactic aspects of the domain of discourse in a machine understandable manner.

2. *Web Services :* Represent information about the actual web service descriptions and the service capability and functionality.

3. *Goals :* Describe aspects related to the desires of users of the service. This element provides a model for the user's view in the service usage process.

4. *Mediators :* These elements handle all the problems of interoperability between the other elements of WSMO.

Therefore we see here that OWL-S and WSMO provide us with some basis for automatic composition of web services to be realised.

2.5 The Web Service Usage Scenario

Now that we know how web services and business processes that integrate multiple web services are described, we can move on to understanding how web services are currently used by an independent consumer. The service oriented architecture provides a distinct process that is followed by the three parties involved in a web service transaction or usage instance. This process is cyclic in nature and can be described simply as the *publish-find-bind* cycle shown in Figure 2.4.

The three main phases of this transaction/process are :

1. **Publish :** The service provider (Amazon, in this example) creates an implementation of a service that it wants to provide to consumers. It then creates the WSDL descriptions of this service (interface and implementation) and publishes these descriptions in a service registry (UDDI, in this example). This is the service description and creation phase.

2. **Find :** The service customer/consumer/client (Mr. John Smith, in this example) goes to the service registry and finds the WSDL description of the service he is looking for. This is the service discovery phase.

3. **Bind :** The service consumer then uses the WSDL implementation description that he has obtained from the registry to understand and compose a SOAP input message for the service operation (and portType) that he wants to invoke and get a response from. He then sends this SOAP message to the network address

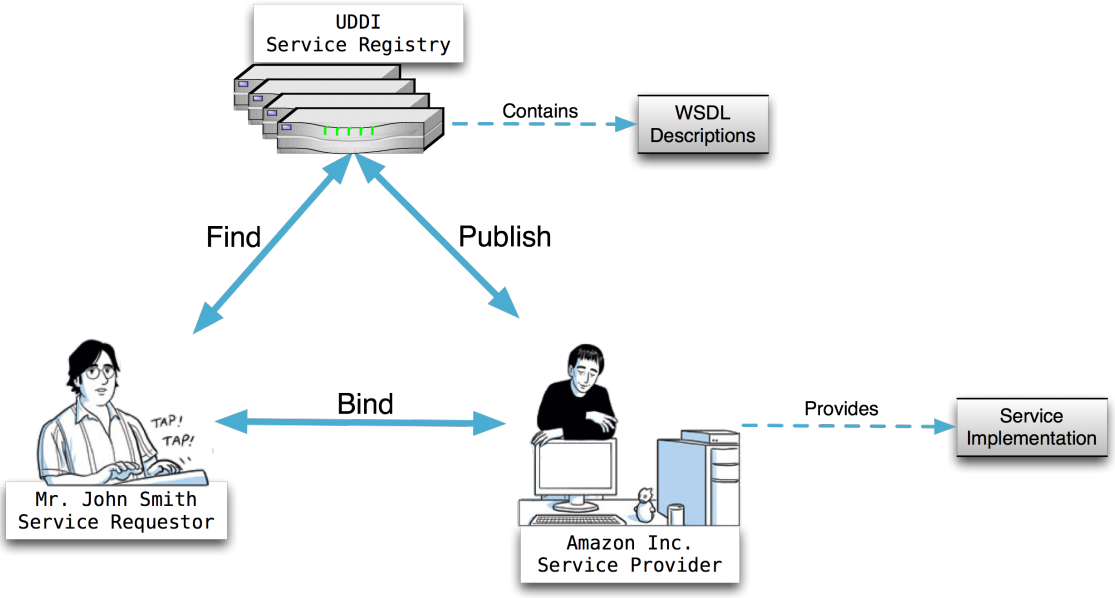

Figure 2.4: The Web Service Usage Process

provided the in the WSDL implementation description and awaits a response. This phase describes the binding between the consumer and the provider and is the actual service usage phase.

We can now see that the *entry point* for a service consumer into the service domain is the service registry. We also see that it is a time consuming process for the service consumer to manually find the service description (according to his requirements), and then bind with the service provider to obtain a result.

What we must also understand is that the consumer might want a result that might require performing several such transactions with several different services. For e.g. if the consumer is after a holiday package, he has to find and bind to all the services required to build the package such as airline booking services, hotel booking services, car rental services and so on. Of course, he might create a BPEL specification for performing these transaction in the sequence he wants, but that would require him to know the semantics of BPEL and exactly how to write a BPEL specification. We already discounting the fact that the consumer has to know about WSDL, SOAP, UDDI and the rest of the WS technology stack, for him to perform a find and bind with a service.

It is reasonable to assume that a normal consumer (Mr. John Smith for example) might have very little knowledge about the technologies used to describe, use and compose web services.

2.6 The Need For A Request System

The future of SOC depends on the usability of web services. As we have seen, it is quite difficult for a consumer with little knowledge about the web service domain to use web services, which in turn renders the usability of services to being very low. The challenge now is to create a request system that :

- Abstracts the user from all the necessary terminology and knowledge required to use web services.
- Provides enough details about available services and what parameters (input information) are required to invoke them.
- Provides the user with a platform and request language to express a service request (i.e. An expression of what the user wants to achieve from invoking web services, similar to WSMO Goals).
- Automatically generates a service composition i.e. an execution plan by interacting with available web services
- Invokes the web services that are part of the execution plan in proper sequence and obtain a final result
- Returns the result or service response to the user.

The figure below describes this vision of a web service request system and a normal users interaction with it.

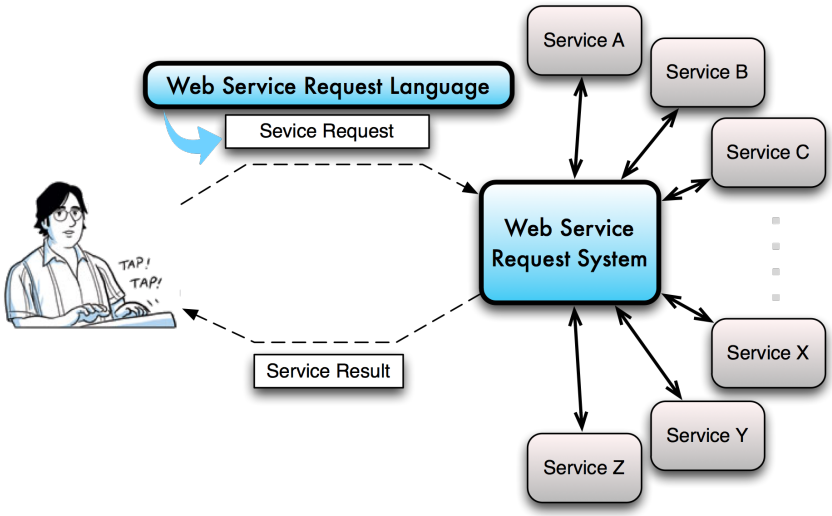

Figure 2.5: The Vision of the Web Service Request System

2.6.1 Creation of the Request System : Research Challenges

Not to mention, creation of such a request system poses a significant number of research challenges. The primary ones are described as follows :

- Creation of service and user models that describe exactly what a user might want to request from a service and how the service might respond to a request
- Creation of a service request language based on the service and user models, that can be used to describe a service request.
- Automatic or semi-automatic discovery of services that may be used to fulfil a request
- Automatic or semi-automatic composition of services and generation of a service execution plan based on the composition.
- Execution of the services in sequence described in the execution plan to obtain results

Previous attempts have been made to address some of these challenges, which are describe in the next chapter.

Chapter 3

Related Work

3.1 Request Systems and their languages

After many papers and considerable research, we found two very relevant but different approaches to implementing parts of a request system.

3.1.1 XSRL

In 2002, Papazoglou, Aiello, Pistore and Yang published two papers on XSRL [Papazoglou et al. 2002a] [Papazoglou et al. 2002b] and then another one with Carman, Serafini and Traverso [Papazoglou et al. 2002] in the same year. They proposed a request language based on XML, and the corresponding system based on AI-planning and constraint satisfaction. They had named it XSRL or the XML Service Request Language.

XSRL was targeted to be expressive enough for the user to be able to express his/her objectives clearly and unambiguously. It was an amalgamation and extension of constructs from the XQuery [Boag et al. 2002] language and EaGLe [Lago et al. 2002], a formal AI planning language.

A basic XSRL request was of the following form :

```
request ::= '<XSRL>' request_expression goal_formula '</XSRL>'
```

A basic *request_expression* would have the form :

```
request_expression ::= '<Request>' FlwrExpr '</Request>'
```

Here the FLWR sub expression represents the FOR, LET, WHERE and RETURN statements. Further details of BNF grammars specific to the above predicates are given in [Papazoglou et al. 2002a].

A typical XSRL request would look like [Papazoglou et al. 2002a],

```
<XSRL>
   <XQUERY> {
      FOR $a in document(PkgTravelSegment.xml)//AirSegment
          [CarrierName =  "Alitlaia"| "United Airlines" AND
           DepartureAirport = "NewYork" AND
           ArrivalAirport = "Rome" | "Venice" AND
           (Price <= 800 AND Price >=500) AND
           SeatQty = 3 AND
           ArrivalDate = "1 June, 2002" AND
           DepartureDate = "10 June, 2002"]
      RETURN
          <ArrivalAirport>{ $a/ArrivalAirport}</ArrivalAirport>
          <price>{ $a/price}</price>
          <ArrivalDate>{ $a/ArrivalDate}</ArrivalDate>
          <DepartureDate>{ $a/DepartureDate}</DepartureDate>
          <HotelList> {
             FOR $h in document (hotelReference.xml)//HotelReference
                 [ChainHotel = "Hilton"]
             WHERE ($h/Area =$a/ArrivalAirport AND
                    $h/HotelArrivalDate = $a/ArrivalDate + 1 AND
                    $h/HotelDepartureDate = $a/DepartureDate 1)
             RETURN
                <HotelName>{ $h/HotelName }</HotelName>
                <HotelAddress>{ $h/HotelAddress }</HotelAddress>
          }</HotelList>
   }</XQUERY>
   <GOAL>
      <Then><Vital>receive_confirmation($a)</Vital>
      <Optional> receive_confirmation ($h)</Optional></Then>
   </GOAL>
</XSRL>
```

Figure 3.1: An XSRL Example Request

We can see clearly here that this request is directed to a travel agency, aimed at getting confirmation for 3 *Alitalia* or *United Airlines* air tickets from New York to (Rome or Venice) departing on the 1st of June and returning on the 10th. It also requests a list of hotels , Hilton if possible, in that area, which have rooms available in that time period.

The request is encoded into a constraint satisfaction problem with all the information in the request described by different variables of the problem. This is then passed to the AI planner which generates an execution plan. The plan is then passed on to the

executor which invokes the required services to produce a result.

Alexander Lazovik continued research with Papazoglou and Aiello on XSRL and submitted his doctoral thesis on "Interacting with Service Compositions" in 2006 [Lazovik 2006]. He considerably extended the previous work done and provided full grammars and constructs related defining XSRL. He has also developed parts of the XSRL request system which includes :

- *A Monitor :* Coordinates the planner and executor, i.e. interleaves planning and execution.
- *A Planner :* Produces the service execution plan
- *An Executor :* Execute the plan to generate the result

The architecture is also dynamic, in that it takes into account changes made in the request, or environmental constraints and user interaction. The monitor is specifically designed to interleave planning and execution, which provides feedback to the user during execution, which is also critical to the service usage process.

3.1.2 Framework for Web Service Query Algebra and Optimisation

Bouguettaya and Yu present a different view point to the same topic in their paper published last year [Bouguettaya and Yu 2008]. They present a query algebra which is similar to a request language and based on logic predicates. The query algebra proposes to provide optimised access to web services based on their "functionality" and "quality". The queries are highly service-oriented. The algebra is based on a formal service model that provides a high level abstraction of web services across a generic application domain.

The product generated after a query is supplied, is a service execution plan or an SEP. This plan could be used by a consumer to access the desired services to get the desired result. The approach taken to generate this plan is graph based. Directed service graphs with service operations as the nodes are created and the appropriate work flow is extracted from the graph, based on the given query. Details of all the algebraic operators used in the entire process are provided. The paper also includes an optimisation algorithm that extends the dynamic programming approach to efficiently select the SEP's with best user-desired quality.

Their main aim was to layout a theoretical background for the development of a web service management system(WSMS) that would be to web services as Database management systems are to data. Specific contributions of the paper include :

- *A Service Query Model :* Based on a formal service model that captures functionality, behaviour, quality of services

- *A Service Query Algebra :* Based on the service query model, enables generation of SEP's
- *Service Query Optimisation :* Score function to calculate quality of SEP's, algorithms to implement it

The service query model is based on two concepts :

1. *Service Schema* : captures key features of all web services across an application domain
2. *Service Relation* : composed of sets of instances conforming to a service schema

Core concepts of a service schema include :

1. *A Service Graph :* A directed acyclic graph with service operations as vertices, and dependencies between two operations as the edges.
2. *Service Path :* A set of operations from a service graph
3. *Operation Graph :* The union of all service paths that lead to a particular service operation
4. *Operation Set Graph :* The union of all service paths that lead to all operations in a set of service operations
5. *Accessible Operation :* If an operation is a part of a service graph and all its dependent operations are also a part of the service graph, then it is an accessible operation
6. *Accessible Graph :* If each operation in a service graph is an accessible operation, it is an accessible graph

The paper also defines a QoS model to capture the quality features of web services. This model categorises the QoS attributes into two sections :

1. *Run-time Quality :* latency, reliability, availability
2. *Business Quality :* fee, reputation

The service algebra defined on the basis of the query model has 3 major operators :

1. *Functional Map :* to locate and invoke desired functionality
2. *Quality Based Selection :* to locate service instances with desired quality
3. *Composition :* enables service composition to access multiple services

To summarise, this paper provides a graph based predicate logic approach to generate service execution plans (essentially work flows) based on a query (essentially a request) provided by a user (service client).

3.2 Comparison with The Web Service Request System

XSRL provides a fairly rich XML syntax for a request language but takes an AI constraint satisfaction planning approach. The user's request is translated into business process variables that are encoded into the constraint satisfaction problem. As the user expresses a more detailed request (i.e. more informations is expressed in the request which may not relate to the eventual functionality but may related to the expected quality and other business assertions), the number of process variables grow and hence the time complexity associated with generation of the execution plan using the planning algorithms, increases.

Again, considering the fact that we are trying to create a practical system that generates execution plans on the fly i.e. during a user's interaction session, it is infeasible to expect a constraint satisfaction solver to be time optimal and deliver quick results. No pre-computing mechanisms are adopted in this approach to reduce the time span between a user's delivering a service request, and the system returning a result.

On the other hand, [Bouguettaya and Yu 2008] provides a solid mathematical predicate logic and graph based approach to planning execution of services. It has significant pre-computing benefits incorporated in its approach which reduces the time complexity and the actual time span between a service query and the generation of the execution plan. However, it does not include a normal user friendly language for expressing the service queries or requests as it is focused on developing a *web service management system* rather than a request system.

The main advantage that our approach has over XSRL is the fact that we distinguish between different types of business process variables in the planning problem, and therefore cut down heavily on planning time complexity. Since we adopt a graph based approach for generating the service execution plan, similar to the one used in [Bouguettaya and Yu 2008], we introduce a pre-computing step for generating the graph, which reduces the request processing time complexity.

We are now able to see that neither of these approaches were complete with respect to the needs of a request system expressed in section 2.6. Therefore, we can now move on to describing the WSRS and all of its parts in Chapter 5.

Chapter 4

Our Service Example

Before we move on to describing the WSRS, we must introduce a suitable set of examples that can be used to demonstrate the new concepts that we have developed. As implementation of the system has been delayed, we use these examples to test the design of the system by providing data flow and entity relationship diagrams in the next chapter. We model the case of the Online Travel Agency. We list five fundamental web services used by the OTA :

- *The Travel Agency Service :* This service coordinates all the payment issues.
- *The Hotel Service :* This service provides hotel room availability, pricing information retrieval and room booking facilities.
- *The Flight Service :* This service provides flight availability, pricing information retrieval and flight booking facilities.
- *The Train Service :* This service provides train availability, pricing information retrieval and train booking facilities.
- *The Payment Agency Service :* This service provides credit card validity checking and credit card billing facilities.

The business transactions that the OTA must support are as follows :

- **Creation of Travel/Holiday Package**
  - *Input required from customer:* Budget, Origin, Destination, FromDate, ToDate, PreferredFlightCompany, PreferredTrainCompany, PreferredHotel, CreditCardNumber
  - *Initial output :* TravelPlan
  - *Final output :* TravelPackage
- **Booking a Hotel**
  - *Input required from customer:* FromDate, ToDate, PreferredHotel, CreditCardNumber
  - *Initial output :* HotelAvailability, HotelPrice

  - *Final output :* HotelReservation

- **Booking a Flight**
  - *Input required from customer:* Origin, Destination, PreferredFlightCompany, CreditCardNumber
  - *Initial output :* FlightAvailability, FlightPrice
  - *Final output :* FlightBooking

- **Booking a Train**
  - *Input required from customer:* Origin, Destination, PreferredTrainCompany, CreditCardNumber
  - *Initial output :* TrainAvailability, TrainPrice
  - *Final output :* TrainBooking

We provide the information that the customer must supply i.e. Input web service messages for each of the transactions. The initial output represents the possible options that must be presented to the customer before his/her credit card is charged and a final output is generated. We now provide details of the service operations and the input and output messages associated with each operation for each of the five services. Each of the services we list along with its operations, describes a service interface i.e. the implementation descriptions of these services and quality of service parameters associated with the implementations are not listed here.

Travel Agency Service

- **payForHotel()**
  - *Input :* CreditCardNumber, Price
  - *Output :* HotelPaymentStatus

- **payForFlight()**
  - *Input :* CreditCardNumber, Price
  - *Output :* FlightPaymentStatus

- **payForTrain()**
  - *Input :* CreditCardNumber, Price
  - *Output :* TrainPaymentStatus

- **checkFinances()**
  - *Input :* CreditCardNumber, Price
  - *Output :* FinancialValidity

- **generateTravelPlan()**
  - *Input :* Origin, Destination, Budget, FromDate, ToDate
  - *Output :* TravelPlan , PlanPricing
- **createTravelPackage()**
  - *Input :* TravelPlan
  - *Output :* TravelPackage , PackagePricing

Hotel Service

- **checkHotelAvailability()**
  - *Input :* FromDate , ToDate
  - *Output :* HotelAvailability
- **getHotelPrice()**
  - *Input :* FromDate, ToDate
  - *Output :* HotelPrice
- **reserveHotel()**
  - *Input :* FromDate , ToDate
  - *Output :* HotelReservation

Flight Service

- **checkFlightAvailability()**
  - *Input :* Origin , Destination , Date , Time
  - *Output :* FlightAvailability
- **getFlightPrice()**
  - *Input :* Origin , Destination , Date , Time
  - *Output :* FlightPrice
- **bookFlight()**
  - *Input :* Origin , Destination , Date , Time
  - *Output :* FlightBooking

Train Service

- **checkTrainAvailability()**
  - *Input :* Origin , Destination , Date , Time
  - *Output :* TrainAvailability
- **getTrainPrice()**
  - *Input :* Origin , Destination , Date , Time
  - *Output :* TrainPrice
- **bookTrain()**
  - *Input :* Origin , Destination , Date , Time
  - *Output :* TrainBooking

Payment Agency Service

- **checkCreditCard()**
  - *Input :* CreditCardNumber , Price
  - *Output :* CreditCardValidity
- **chargeCreditCard()**
  - *Input :* CreditCardNumber , Price
  - *Output :* CreditCardChargeStatus

Not all services described in this example are used in the next chapter, but however we thought it was better to provide the entire example rather than a subset of it. We now move on to describing and defining each part of our request system, using some of the services described above as examples to demonstrate the concepts.

Chapter 5

The Web Service Request System

To address the research challenges expressed in section 2.6.1 and to fulfil the needs expressed in section 2.6, we introduce and define the web service request system in this chapter.

5.1 Conceptualising the system : A Software Cloud

The first step in creating the request system is to understand how it might be used and who it might be used by. We shall be referring to the *web service request system* every time we use the word *system*. We now list the *roles* of people associated with the system and how these people use the system:

- **The User/Consumer/Customer :** Any individual/business/machine could implement this role. This is the entity that sends a service request to the system and waits for a response/result from the system.
- **The System Designer :** This person/business/organisation is responsible for creating the service interface descriptions for the services that the system shall provide.
- **The Service Provider :** This person/business/organisation shall provide service implementation descriptions for the services that it wants to provide to consumers via the system.

We now see that the system acts as a software cloud that a service consumer interacts with as shown in Figure 5.1. The system virtually realises the vision of cloud computing and SOC in an integrated manner. By providing a cloud of services to users, the system ensures that all the services shall be available to the users as long as they have access to the system. This potentially rules out connectivity issues that users might have to the internet in particular, as access to the system may be provided via any network (not only the internet).

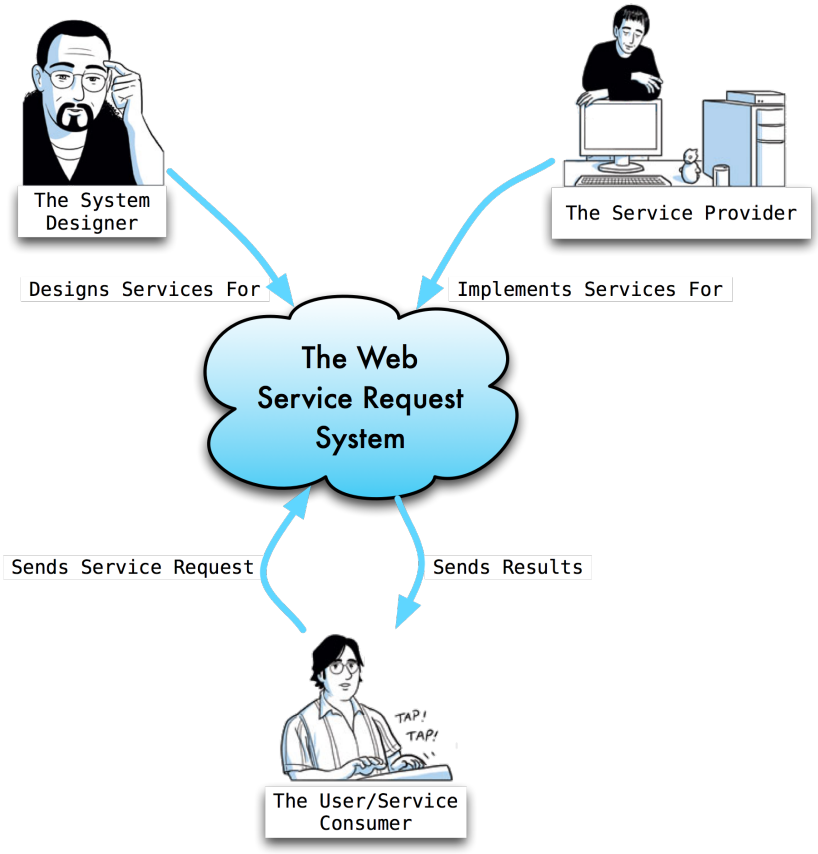

Figure 5.1: The Request System As A Software Cloud

5.2 Defining parts of the system

The next step is to clearly define constituents of the system. We subdivide the system into distinct parts and subparts :

1. **The Model :** This can be considered as an ontology that provides semantic markup/information/meta-data about web services and the user's requirements.
   - *The User State Model :* The possible requirements of a service consumer and the state of the consumers request in the system, with respect to multiple services, are described by this model.
   - *The Service Description Model :* As the name suggests, this model describes parts of web services provided by the system and also provides meta-information about these services that are not included in the service descriptions.
2. **The Language :** The service consumer uses this to specify a service request. Requests are described in terms of variables defined in the model.

3. **The Framework :** This is the part of the system which accepts the user request, performs all the necessary tasks and returns a result to the user.

   - *The Processes :* The data/information flows in the system are modelled by these processes. The algorithms required to achieve automatic composition of atomic services, and obtaining results from them are part of these precesses. There are 3 main processes involved in the framework :

     (a) *The Initialisation Process :* This describes the process of service design and implementation and all the pre-computing steps done before the system starts accepting requests from users.

     (b) *The SEP Generation Process :* This describes the data flows involved in the service execution plan generation.

     (c) *The SEP Execution Process :* This describes the service execution process according to a SEPlan and the users possible involvement in the process.

Figure 5.2 describes the web service request system as stated.

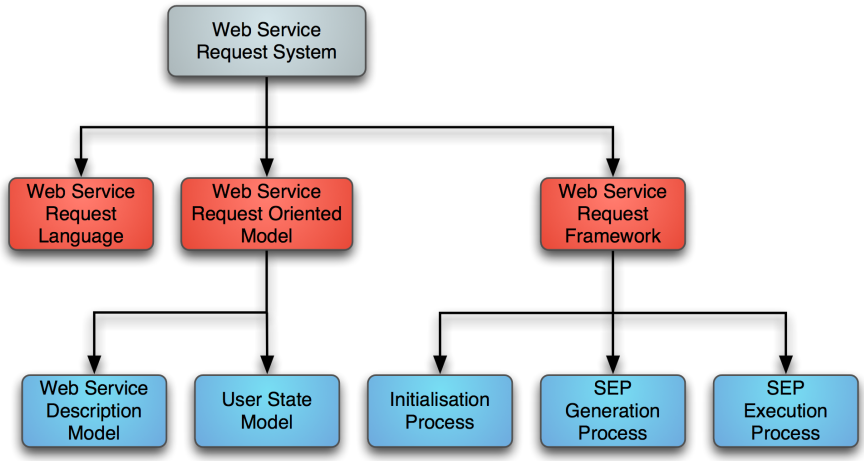

Figure 5.2: The Web Service Request System

We now move on to describing in detail, parts of the system, starting with the web service request oriented model.

5.3 The Web Service Request Oriented Model

There are several models that describe parts of the web service architecture from different view points. [Booth et al. 2004] describes four important models that can be used to describe web services, from a unique view point :

1. *The Message Oriented Model :* Explains web services from a message passing perspective. It does not relate the messages to the services and focuses on the message structure, transport etc.
2. *The Service Oriented Model :* Sits on top of the message oriented model and explains the purpose of the messages from a service's point of view.
3. *The Resource Oriented Model :* Focuses on resources that exist and the owners of the resources. Services can be considered as resources.
4. *The Policy Oriented Model :* Focuses on the constraints applied to the behaviour of resources.

Therefore, we see that each of these models builds on the previous one and provide an extra layer of information or semantic mark-up on top of the previous model. It is easy to see that we could use all of these 4 models together to describe the behaviour of a web service. But these models do not provide us with enough information to describe web services from the view point of a service request given by a user. That is precisely why we create a separate *request oriented model* to describe the behaviour of services w.r.t a service request. This model is an additional model to the other four described above. It does not attempt to replace any of the previous models, but tries to incorporate further semantic information about web services that pertain to service requests.

The request oriented model is primarily aimed to describe two separate entities :

1. *The state of the user with respect to the system :* This represents the fact that the user wants a product from (via) the system which might require the invocation of one or many services. During the service invocation phase, the system obtains output messages or results from these services. The user is concerned only about these results and therefore his state (w.r.t the system) is an indication of the type of results he is after. Essentially a service request given by the user is an expression of the state that he wants to achieve after the invocation of services.
2. *The description of a web service w.r.t the parameters of user requests :* As we have mentioned before, the existing models of web services do not provide any information pertaining to user requests. the request oriented model aims to create service descriptions that describe web services using the information that users might provide in a service request.

To describe these two entities, we divide the request oriented model into two separate models, the *user state model* and the *service description model*.

5.3.1 The User State Model

In this thesis, we consider business processes equivalent to service execution plans, that represent the control flow of business logic. It is important to understand a business process from the point of view of the user interacting with the business process

and awaiting its result. The user is primarily concerned with three things, the *functionality* (at the service operation level) achieved by the business process, the *resulting output* (at the service message level) that is returned by the business process and the *quality* of the output. Therefore it is safe to say that the user's requirements from the business process would also be centred around these three things.

The user state model is aimed at allowing the user to describe his service request as a particular state described by the model. The request language allows the user to describe the user state that he wants to arrive at. We now introduce the concept of a *user state*. It models the state of data (relevant to the user) during execution of the business process. A user state consists of a combination of *web service variables*. Web service variables achieve two objectives :

1. They are used to model the possible execution sequences of services that compose a business process i.e. they are used to model the behaviour of business processes.

2. They are used to track the progress of obtaining results based on user requirements during execution of a business process i.e. they track and model the data flow in a business process that leads to a result or output.

We then create a *user state graph* which models the state transitions of user states, which in turn describe the possible execution paths of a business process. The transitions in the states i.e. the edges of the graph or the *state dependencies* each reflect the execution of an atomic service. The idea is that when we execute an atomic service (with a single operation), we expect that the execution shall return a result or output that shall change the value of a web service variable in a user state. Therefore, *execution of an atomic service changes the user state* and hence progresses it to the next or successor state.

Figure 5.3 provides an outline of the user state model.

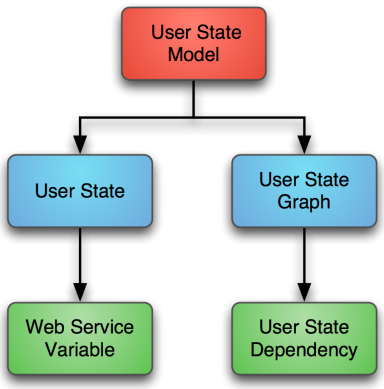

Figure 5.3: The User State Model

Before we go on to describe the user state and the user state graph, it is important that we describe the web service variable.

5.3.1.1 The Web Service Variable

We have previously talked about the system designer being responsible for designing the services present in the request system. Now we describe exactly how the designer goes about that task. If we look at the initial phase of creating a web service interface, we find that the process is *top down* and consists of the following steps :

1. *Functionality Identification :* Identify what functionality needs to be offered by the service. If a service provider (who already has an implementation of the business functionality) is trying to create a web service, it is fairly easy for him to provide both the WSDL interface and the implementation descriptions. But, when we consider the case of our system designer, we see that it is primarily the identification of functionality required by the service consumer.

2. *Service Operation Definition :* Define service operations that can provide the required functionality.

3. *Service Message Definition :* Define all the input and output messages required by the operations.

4. *Data Type Definition :* Define all the data types required by the messages.

At this point we have enough information to write a WSDL service interface description i.e. create a service interface. The *web service variable* provides the system designer with an easy way to model a web service w.r.t its behaviour and capability. The designer essentially follows the same top down approach listed above to create a set of web service variables, which together reflects how the web service is supposed to behave and what exactly its capable of providing. The quality of service and other implementation specific information can also be represented by service variables. Formally, the web service variable is defined as follows :

Definition 1 (Web Service Variable)
A web service variable can be described as a representation of service capability, quality and behaviour with respect to a service consumer. It is defined as a tuple, $var =< ID, T, Name, Val, S >$ *where :*

- ***ID*** *is a unique global identification number that is assigned to each variables.*
- ***T*** *is the type of the service variable. There are 5 different types of variables :*
    1. ***Identification Variable*** $< ident >$ ***:*** *Represents core identity information about a service i.e. service name, abstract data types used, xml namespaces used.*
    2. ***Functionality Variable*** $< fn >$ ***:*** *Represents the state of a service operation in the business process execution sequence. (Boolean)*

3. ***Information Variable*** $< info >$ **:** *Represents data or payload i.e. service messages used by a service.*
4. ***Policy Variable*** $< pol >$ **:** *Represents a WS-Policy attribute of a single service or common to a set of services. This includes quality of service, security and versioning information.*
5. ***Implementation Variable*** $< impl >$ **:** *Represents the service provider and service implementation elements such as network address and port bindings.*

- ***Name*** *is the name of the variable which includes a service namespace for the service that this variable describes.*
- ***Val*** *is defined as a double,* $Val =< D_T, D >$ *where :*
  - D_T *is the data type of the payload of the service variable.*
  - D *is data of type* D_T *and is the payload of the service variable.*
- ***Status*** *is defined as a tuple,* $S =< I_S, I_T, I_V >$ *where :*
  - I_S *is the instantiation status of the variable. It depicts whether the variable has a value yet or not, i.e. has it been instantiated yet or not. (Boolean)*
  - I_T *is the instantiation time of the variable. It contains the last time that the variable was instantiated and its value changed. (Time)*
  - I_V *is the instantiation validity of the variable. It contains the time span that the value of the variable is valid from the time of last instantiation i.e.* I_T*. (Time Span)*

Figure 5.4 describes the attributes present in a web service variable.

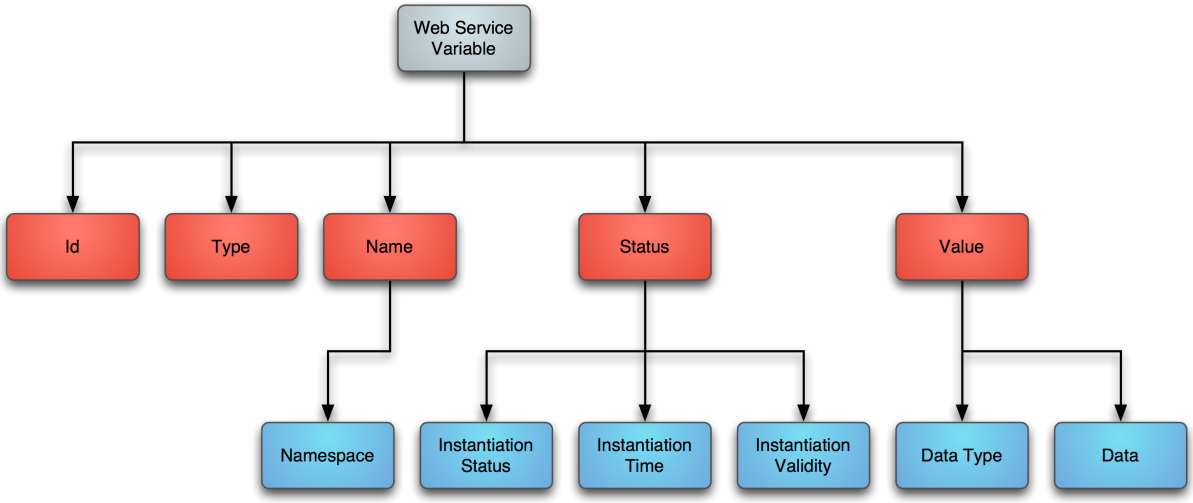

Figure 5.4: The Web Service Variable

In the *user state model* we are only concerned about 3 of the 5 types of web service variables i.e. the identification, information and functionality. These variables represent abstract service definition elements only and can therefore be referred to as *abstract service variables*. The other two types, implementation and policy represent the concrete service description elements and are also described as *concrete service variables* which we use in the *web service description model* described in Section 5.3.2.

We now provide a simple example of the process of describing the capability and behaviour of a web service using abstract service variables. We describe the **Train Service** provided in our service example in chapter 4. We do not include the *namespace* for each variable in this example as they all belong to the same service and hence the same namespace i.e. *Train*. However, if we were to refer to the variables using the namespace, we would express each variable *Var* as *Train*::*Var*. We shall now go through the process step by step :

1. *Functionality Identification :* The train service must implement 3 functions :
   (a) Check the availability of seats
   (b) Get the price of tickets
   (c) Book seats on the train
2. *Create Functionality Variables :* Define 3 functionality variables that represent the state of each of the required functions : **checkedAvailability**, **gotPrice**, **gotBooking**. Here we see that each of these variables are designed to represent the state of each of the functions of the service, and each of them have a value of either **true** or **false** i.e. they are of data type boolean. They are given the data value = *false*.
3. *Create Information Variables :* Define 7 information variables that represent the information or data that is required as input and output to each of the functions. The input variables are **Origin**, **Destination**, **Date**, **Time**. The output variables are **TrainAvailability**, **TrainPrice**, **TrainBooking**. Each of the variables shall contain either data provided by the user, or returned by the service after invocation.
4. *Create Identification Variables :* Define 7 identification variables corresponding to the name of the service and to the data types that each of the information variables require : **ServiceName**, **PlaceType**, **DateType**, **TimeType**, **AvailabilityType**, **PriceType**, **BookingType**. ServiceName is given the value, *Train*.
5. *Assign Data Types :* Now assign each service variable their data type. So, all functionality variables are *boolean*. **Destination** and **Origin** are of type **PlaceType**. **Date** and **Time** are of type **DateType** and **TimeType** respectively. **ServiceName** and **PlaceType** are of type *string*. **DateType** and **TimeType** are abstract types and consist of several fields each. **DateType** has 3 fields : *Day (integer)*, *Month(integer)* and *Year(integer)*. **TimeType** represents time of day and has 2

fields : *Hours(integer)*, *Minutes(integer)*. The **AvailabilityType** is *boolean*, **PriceType** is *float* and **BookingType** is *string*.

6. *Assign Status Values :* The status values of each variable reflect whether the variables have been assigned any values or not. Therefore, all the identification and functionality variables are given *Instantiation Status = true* and the time at which this is done as the *Instantiation Time*. The information variables are given *Instantiation Status = false* as they don't hold any data yet.

We have now described the abstract **Train Service** via the *identification*, *information* and *functionality* variables. It can be seen that the first 4 steps if this example process correspond to the first 4 steps in the service interface creation process that we outlined in the beginning of this section. It is also safe to say that once the service variables corresponding to a service interface are created, the process of creating an actual WSDL interface description of the service becomes trivial. However, the other two types of variables i.e. the implementation and policy variables are created by the service provider, once he has received the abstract service description. We shall see examples of implementation and policy variables in section 5.3.2.2.

5.3.1.2 The User State

As we have mentioned before, the user state represents the information relevant to the user during execution of a business process. Therefore in our **Train Service** example, the user would only care about the functionality (**checkedAvailability**, **gotPrice**, **gotBooking**) and capability (**TrainAvailability**, **TrainPrice**, **TrainBooking**) provided by the service during its execution. Formally, we define a user state as follows :

Definition 2 (User State)
A user state can be defined as a triple, $S =< ID, Var_{fn}, Var_{Info} >$ *where :*

- *ID is a unique global identification number for this state. ID is 0 for the root state.*
- Var_{fn} *is a set of functionality variables.*
- Var_{info} *is a set of information variables.*

An example user state corresponding only to the **Train Service** is given in Figure 5.5. It represents the initial state of the user with $id = 0$ when no services were invoked and then the state transition to the next state with $id = 1$ caused by the invocation of the **checkAvailability**() atomic service. We see clearly how the parameters passed to the invocation (*Barcelona,Amsterdam,30/10/2009,10:00*) represented by the input information variables *Origin*, *Destination*, *Date* and *Time* are added to the next state along with the value of the output message *TrainAvailability*, which is *true*.

This state of the user represents a point in execution of the business process. In this example, the business process is one that gives train booking and information facilities and has operations of the**Train Service** as parts of it.

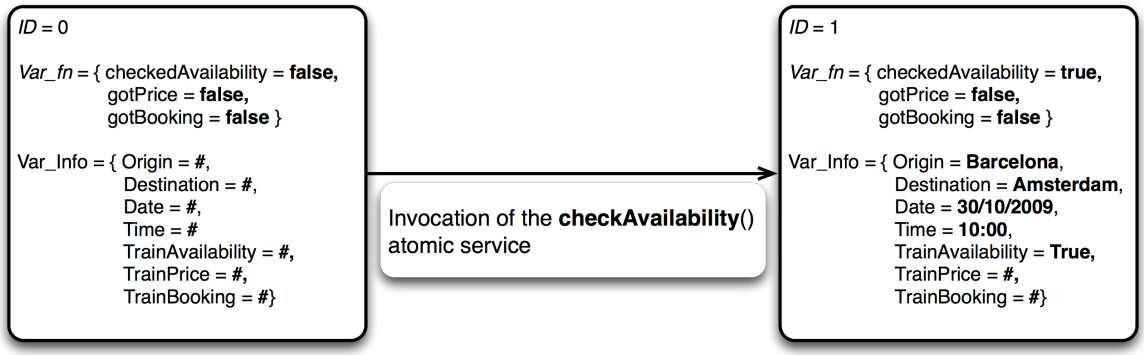

Figure 5.5: A User State Changed By Invocation Of An Atomic Service

5.3.1.3 The User State Graph

The sole purpose of the user state graph is to show all the possible execution sequences that a business process might have. A state in the graph progresses to the next state when a service operation is invoked. As a result of the invocation, the functionality variable representing the state of the service operation that was invoked is given value *true* and the information returned by the invoked service (i.e. output messages of the invoked operation) is incorporated into the information variables of the next state. However, we see that the *information variables only get values after actual invocation of a service*. Therefore, it is important to note that the user state graph is *not* a runtime graph and therefore, none of the information variables in the states of the graph have any values. The graph represents all the possible execution sequences and not an execution itself. It is generated as soon as the services offered by the system are designed by the *system designer* which represents a **pre-computing step**.

The edges of the graph, also formally known as the *user state dependencies* represent the service operation invocations and hence are depicted by service variables. Formally, we define a user state dependency between two user states as follows :

Definition 3 (User State Dependency)
A user state dependency can be defined as a triple, $S_{Dep} =< S_{pred}, S_{succ}, Comp_{fn} >$ *where :*

- S_{pred} *is the predecessor (or previous) state id for this dependency.*
- S_{succ} *is the successor (or next) state id for this dependency.*
- $Comp_{fn}$ *is a functionality component that belongs to the abstract web service description model and it represents an atomic service that takes state* S_{pred} *to state* S_{succ}*.*

The functionality component that we have referenced in this definition is stated in Section 5.3.2.1. The important thing to note about it is that it represents an atomic web service i.e. a single service operation.

We now formally define the user state graph as follows :

Definition 4 (User State Graph)
The user state graph is a directed acyclic graph (DAG) defined as a triple, $USG =< r, V, E >$ *where :*

- ***r** is a user state which is the root node of the DAG*
- ***V** is a set of user states that are the nodes of the DAG*
- ***E** is a set of user state dependencies that are the edges of the DAG*

However, to generate a user state graph, we need another layer of semantic information reflecting the pre and post conditions that apply to each service operation. This is because, a single service operation may provide more than one functionality and hence may change more than one functionality variable, which is reflected as a post-condition of the operation. Pre-conditions of an operation represent the functionalities that must have already been invoked prior to the operation being invoked. We provide this semantic data in the functionality component defined in section 5.3.2.1. Therefore, a user state graph can only be generated once the functionality component is created. We now give an example of a user state graph. Again, we use the **Train Service** and describe the users state with respect to invocation of operations of that service in Figure 5.6.

We have made some assumptions in this example regarding pre-conditions of the operations. We assume that the post-condition of each operation reflects a change in only 1 functionality variable, as shown in the graph. We also assume that the **bookTrain** operation is dependent on the **getTrainPrice** operation which is dependent on the **checkTrainAvailability** operation. In the example we can see how the states change and the values in the state dependencies, reflecting the operations invoked.

We now move on to defining the *web service description model* that reuses the web service variables to create semantic service descriptions.

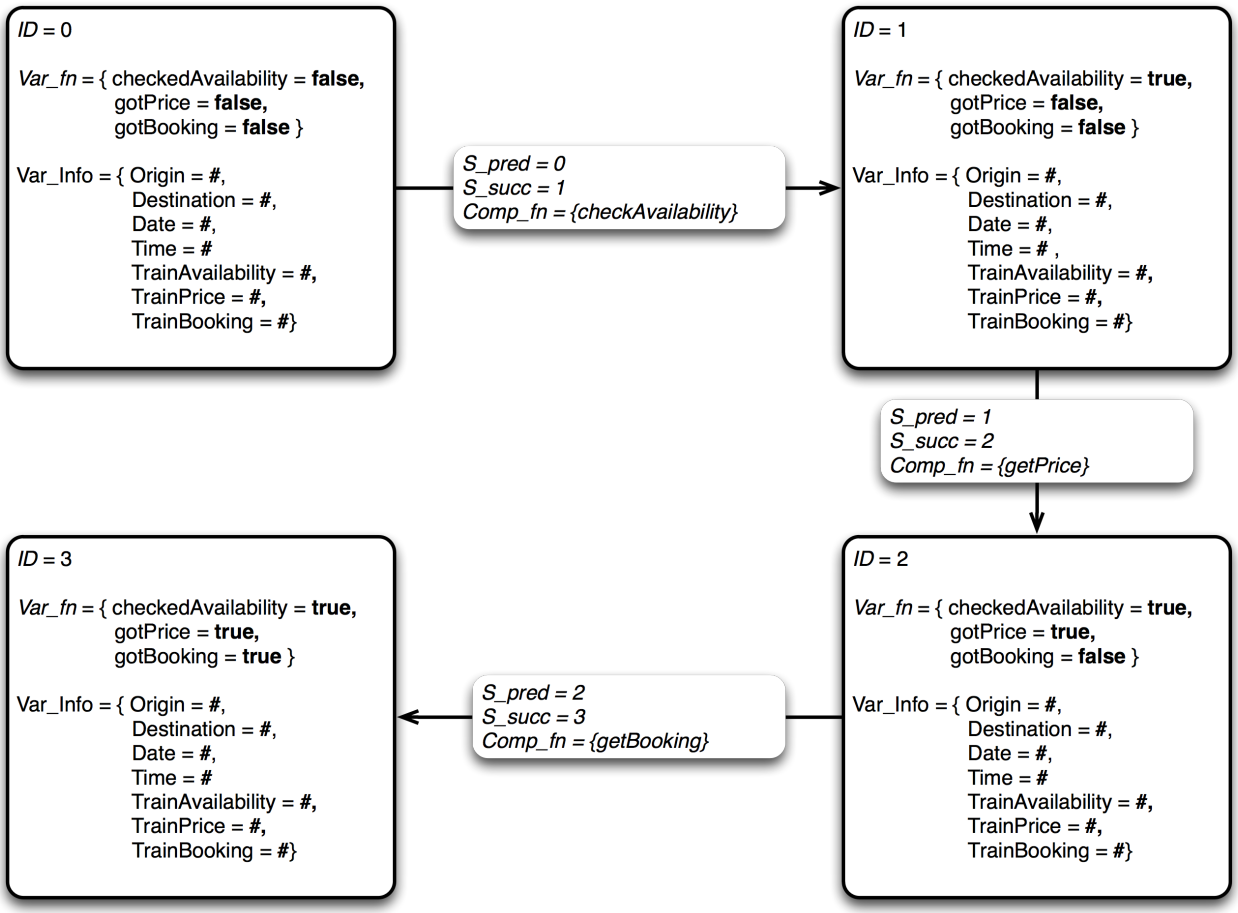

Figure 5.6: A User State Graph w.r.t The Operations Of The Train Service

5.3.2 The Web Service Description Model

As we have seen, the user state model provides a representation of the user's requirements with respect to web services. It also provides a basis on which the system designer can design the services that are required to fulfil these user requirements.When we refer to designing services, we mean :

- Creating abstract service descriptions i.e. WSDL interface descriptions for services
- Including semantic information about elements in the abstract description : This is the most important part of designing the services. Semantic information here reflects the extra meta-data or markup information that is required by the system to facilitate automatic composition of services (described in the system).

After a service has been designed, its abstract description can be given to a service provider, who can then implement the service functionality and provide the concrete

service description based on the implementation and the network address where it is available. The provider also follows a process similar to the one followed by the system designer, when creating an implementation description. He describes the required information in the form of implementation variables and creates a WSDL description based on these variables. However, the service provider must also be able to express policy (i.e. QoS, security protocols, versioning) parameters in the concrete description. Normally, he would have to include a separate WS-Policy Attachment [Papazoglou 2008] with his concrete description. With the help of policy variables that we have previously defined, this task is simplified for the provider. We shall give examples shortly.

We therefore create a model that is used to design the services based on a set of service variables (*functionality,information,identification*) and then implement the design, again based on a set of service variables (*implementation,policy*). We call this model the *web service description model*. The system designer uses this model to describe a single atomic service i.e. one instance of the model describes a web service with one operation. We further subdivide this model into two parts, the *abstract* and *concrete* part. The abstract part represents the interface description of the web service while the concrete part represents the implementation description. Figure 5.5 explains the breakdown of the description model.

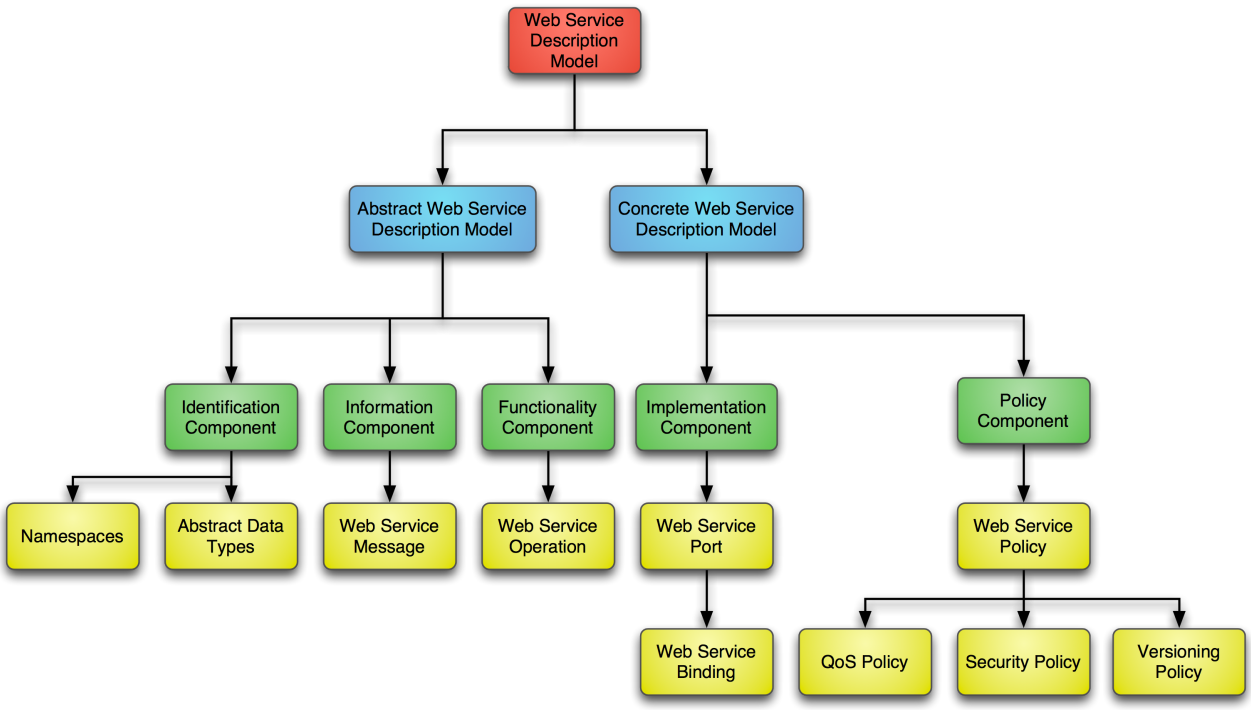

Figure 5.7: The Web Service Description Model

We force a component based architecture on this model, to provide suitable abstraction and facility for reuse of components. Each component in the description model describes an element of a WSDL description. The parts in the lowest level of the diagram (boxes in yellow) represent the WSDL elements that the components are describing. Each component also provides semantic information about the WSDL element that it describes.

The objective that we want to achieve eventually from this component based description model, is automatic creation of WSDL service descriptions by reusing the components, based on the required operations, messages, data types, port bindings and policy parameters of the services in the system. We could potentially create a composite service description automatically based on the user requirements on service variables. This however is not in the scope of this project and we shall flag it as future work.

We now move on to defining and describing in detail each of the two parts of the description model with the help of examples.

5.3.2.1 The Abstract Web Service Description Model

The abstract description model, as mentioned before describes the public interface part of a web service. It has three components, namely the identification, information and functionality components. Before we go on to describe each component, we formally define a web service description component (or simply a component). Each individual component conforms to this definition :

Definition 5 (Web Service Description Component)
A web service description component provides syntactic and semantic information about a WSDL description element. It can be defined as a tuple, $C =< ID, Type, Name, Desc, SemInfo >$ *where :*

- ***ID*** *is a unique global identification number for this component.*
- ***Type*** *is the type of functionality variable that this component represents.*
- ***Name*** *is the name of this component.*
- ***Desc*** *is the WSDL description of the element that this component describes.*
- ***SemInfo*** *is the extra semantic information that this component provides for the WSDL element that it describes. Each component may have different representations of this semantic information.*

As we have mentioned before, each of these components corresponds to an atomic service i.e. a web service with a single operation. For example, if we were to describe the **Train Service** in the form of components, we would have to define separate information and functionality components for each of its three operations. However, only

one identification component would be required to describe the data type etc as a list of web service variables and their corresponding WSDL description. We now move on to describing the individual components of the description models.

The Identification Component

This component contains the service name, the required namespaces, the abstract data types, that apply to the atomic service that this component is part of. It also contains the WSDL description of the information that it represents (i,e, the WSDL <**types**> element, the *targetNamespace*, *name* attributes of the <**definition**> element) via information variables (which is the semantic information part of this component). In our train service example, the identification components representing service names would be **TrainAvailabilityService**, **TrainPriceService** and **TrainBookingService**, Figure 5.8 describes this component.

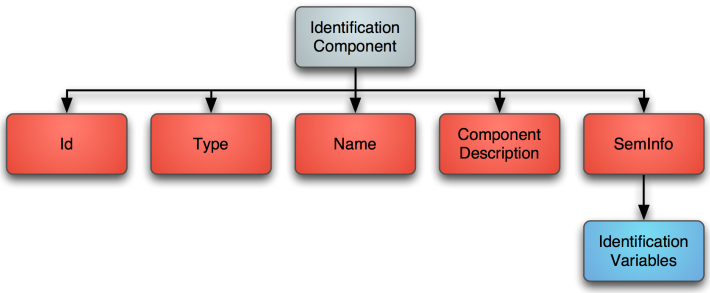

Figure 5.8: The Identification Component

The Information Component

This component is used to describe the input and output messages for the atomic service that this component represents, in the form of information variables. The WSDL <**message**> elements are described by this component. The specification of exactly which information variables are required by this atomic service as input and which ones are required as output, represents the semantic information given by this component, as shown in Figure 5.9. The corresponding train service examples of this component are provided in Figure 5.11.

The Functionality Component

This is the most important component as it represents the only operation of the atomic service that it represents. The WSDL elements described by this component are <**portType**> and <**operation**>. This component provides the pre and post conditions of the operation in the form of Fist Order Logic propositions composed of functionality variables. This is precisely the semantic information required to generate the user state graph. Figure 5.10 describes this component. The corresponding train service examples of

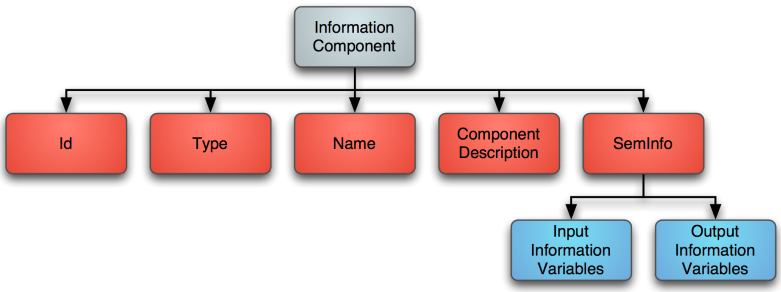

Figure 5.9: The Information Component

this component are provided in Figure 5.11.

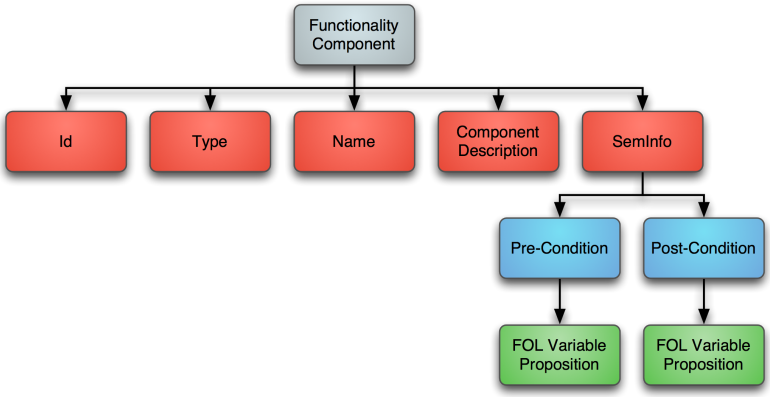

Figure 5.10: The Functionality Component

We now demonstrate the use of the information and functionality components to provide semantic information about the three operations (atomic services) that compose the **Train Service** in Figure 5.11. We do not provide the *Name*, *Type*, *Component Description* and *Id* of these components in the Figure.

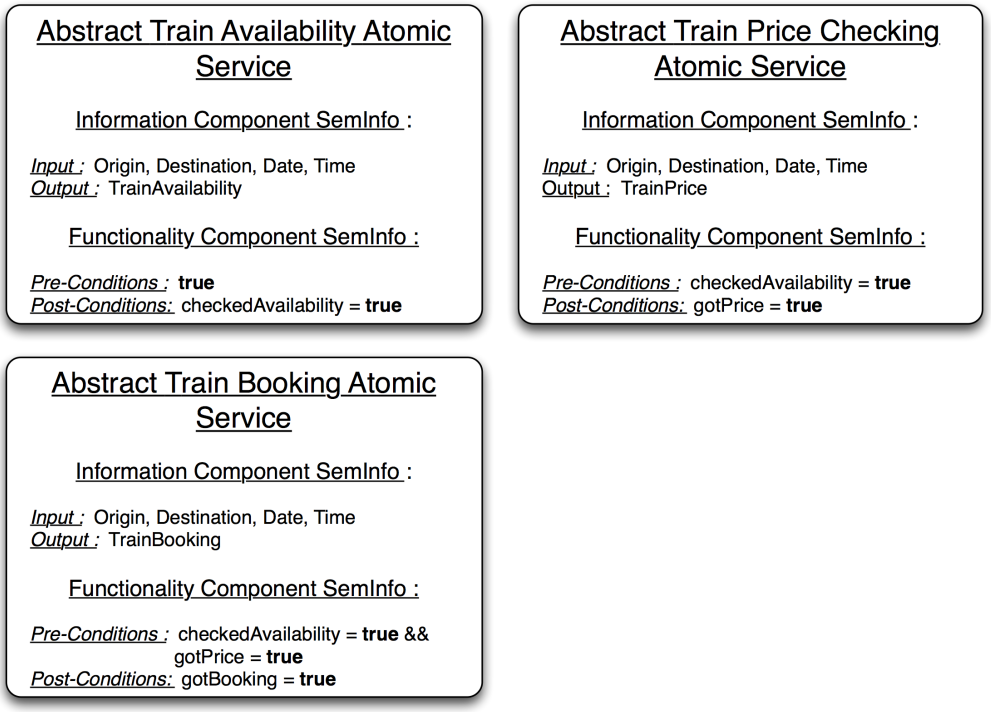

Figure 5.11: Use of Information and Functionality components to describe the Train Service

5.3.2.2 The Concrete Web Service Description Model

As soon as all the identification, functionality and information components of a web service are defined, the work of the system designer is over. These components (containing the WSDL descriptions) are passed on to service providers who now have the option of implementing atomic services from many separate services. For example, say a service provider wishes to provide only informational services i.e. in our OTA example, he decides to provide availability information for hotels, flights and trains. Therefore he wishes to implement the **checkHotelAvailability**(), **checkFlightAvailability**() and the **checkTrainAvailability**() operations. This is now an easy possibility as all he needs to begin implementing these atomic services are the abstract description components corresponding to each of them.

The service provider first creates the implementation and policy variables that model the concrete parts of the services and then creates a concrete description based on our *concrete web service description model*. We now look at the 2 components that compose this model.

The Implementation Component

This component contains the implementation variables that contain the name of the service provider, and port binding information for the service implementation. The WSDL description elements described by this component are <**binding**>, <**port**> and <**service**>. The implementation variables provide the required semantic information for this component. Figure 5.12 describes the structure of this component.

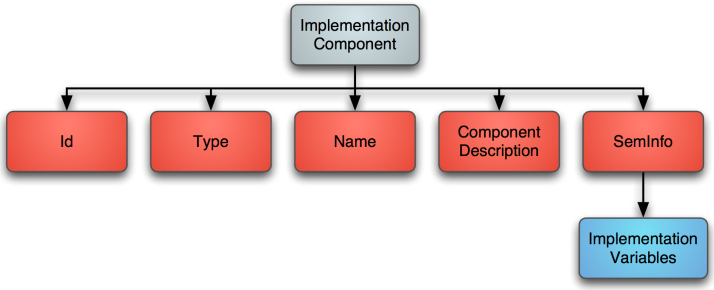

Figure 5.12: The Implementation Component

The Policy Component

This component describes the Quality of Service, security protocols, and service versioning information in a list of policy variables. The WSDL <**Policy**> elements specified in the WS-Policy Attachment [Papazoglou 2008] are described by this component. Figure 5.13 describes this component.

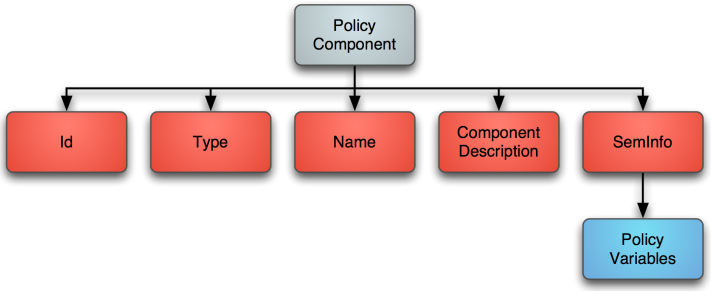

Figure 5.13: The Policy Component

We now demonstrate the use of these two components by the service provider to describe a concrete atomic web services. We again use the example of the three atomic services composing the **Train Service** in Figure 5.14.

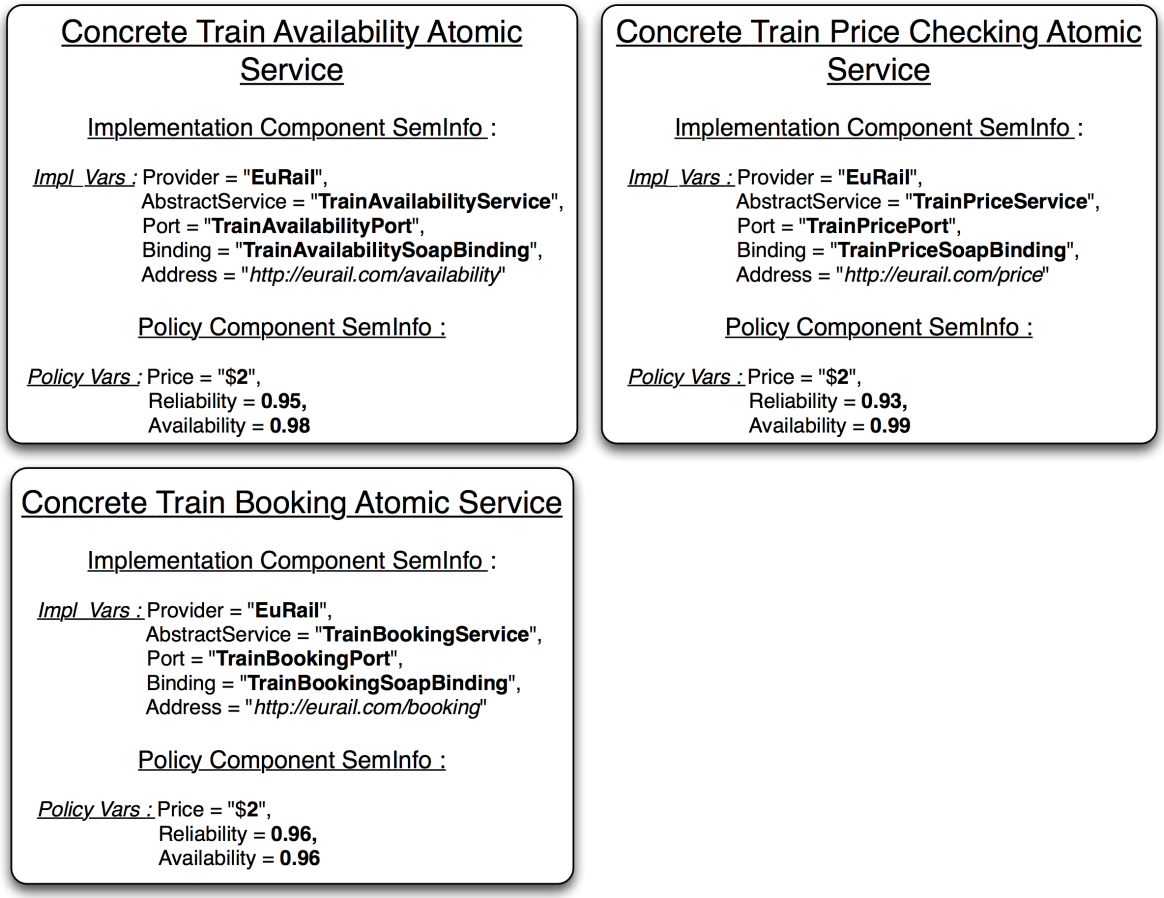

Figure 5.14: Use of Implementation and Policy Components To Describe The Concrete Train Service

In this example we see that each of the three policy components have 3 QoS variables, **Price**, **Reliability** and **Availability**. Price represents the fee that is required to invoke each of these services and is set to $2 each. Reliability represents the reliability of the output given by each of these services. We can see here that each of these services has greater than 93% reliability. Availability represents the percentage of time that the services are likely to be available over the internet. We see here that each of these services is available at least 96% of the time. The provider is free to specify any other policy variables as required. We have only specified 3 in this example due to reasons of brevity.

We also see in this example, the implementation variables that correspond to each of these atomic services i.e. **Provider**, **Service**, **Port**, **Binding**, **Address**. These variables together represent the concrete implementation part of the service description. However, the service provider is free to create any other implementation variables that might be required to represent a particular concrete service description.

We see therefore, that the entire web service description model with its five components together, describes the total behaviour of an atomic web service. We now have enough semantic information to automatically compose several atomic web services described using web service variables and components. The system designer is responsible for creating the service variables and the abstract service descriptions, and the service provider must create the concrete service descriptions. Once we have the abstract descriptions of all the atomic services required, we can then proceed to the automatic generation of the user state graph.

Once the user state graph is generated, we can begin accepting service requests from users. The requests described in the web service request language are then encoded into constraints on service variables. Based on the functionality variable constraints in the encoded request, we automatically find the user state in the state graph that corresponds to it. In the next step, we find the path from the root state to the requested state, again automatically. This path is then used to generate the service execution plan using the concrete description components by the SEP generation engine. The service execution plan is then used by the execution engine to obtain the results required which are then passed on to the user. The processes and algorithms involved in these steps are described in the framework.

We first define and describe our service request language used to express constraints on service variables.

5.4 The Web Service Request Language

The primary goals of the request language are to :

- Allow the user to express the service request in terms of service variables spanning several web services
- Allow the user to express an ordering of service operations based on conditional selection
- Allows the user to describe the state in the user state graph that the user wants to arrive at i.e. the goal state or target state
- Be easy and intuitive enough for the user to understand and use
- Minimise the amount of constructs required to express a request
- Be easy enough for representation in XML syntax

We assume the fact that the users are presented with all the service variables in the system and that the user knows about the five types of variables and their respective purposes. This is a reasonable assumption as far as a normal user is concerned as the variables mostly represent terms that the user would normally be well versed in. For e.g. an information variable *TrainPrice* represents a web service message that is received after invoking the **getTrainPrice**() operation for the **Train Service**, but it also makes meaningful sense to any service consumer as well. If the consumer is after getting the price of a train ticket, the purpose of an information variable *TrainPrice* would be reasonably easy for the consumer to identify.

The primary purpose of the request language is to express constraints on service variables which form a service request. These constraints reflect the service requirements of the user. We must now formally define the meaning of a variable constraint.

Definition 6 (Web Service Variable Constraint)
*A web service variable constraint is defined as a logical expression described in propositional calculus. A Web service variable and its possible value are atomic propositions that are combined by a equational logic connective (either one of $<$ (less than), $>$ (greater than), $=$ (equal to), $<=$ (less than or equal to), $>=$ (greater than or equal to)) to form an atomic variable constraint. Two atomic variable constraints are combined using a propositional logic connective (either one of **and** (Conjunction), **or** (Disjunction), **not** (Negation)) to form a compound variable constraint. A variable constraint is either **mandatory** or **optional**.*

As an example, let us express a constraint on information variables : $TrainAvailability =$ *"Available"* **and** $TrainPrice <$ \$50. In this compound variable constraint, **TrainAvailability**, *"Available"*, **TrainPrice**, *"$50"* are atomic propositions. The equational logic connectives are '=' and '<'. The propositional logic connective is **and**. It must also be noted that a variable constraint on a type of service variable is composed only of service variables of that type. We can now go on to formally define a web service request in terms of variable constraints.

Definition 7 (Web Service Request)
A web service request is defined as a tuple $R = < Var_{id}, Var_{info}, Var_{fn}, Var_{impl}, Var_{pol}, Ret >$, *where :*

- Var_{id} *is a set of web service variable constraints on identification variables.*
- Var_{info} *is a set of web service variable constraints on information variables.*
- Var_{fn} *is a set of web service variable constraints on functionality variables.*
- Var_{impl} *is a set of web service variable constraints on implementation variables.*
- Var_{pol} *is a set of web service variable constraints on policy variables.*
- *Ret is a set of information variables that the request defines as return values.*

We now formally define the request language which is used to express requests for web service compositions in the form of constraints on web service variables.

Definition 8 (Web Service Request Language)
WSRL is defined as a language using which a normal user can express constraints on web service variables. A mandatory constraint on a service variable of type var is expressed using the unary operator ***require***<var> *p, where p is a web service variable constraint composed of service variables of type var. Optional constraints are specified using the unary* ***optional*** *p operator and preferential constraints are expressed using the binary* ***prefer*** p_1 ***to*** p_2 *operator. Conditional constraints are expressed using the binary* ***if*** p_1 ***then*** p_2 *operator. An ordering in constraints is achieved using the n-ary* ***request*** $p_1, ..., p_n$ *operator. The information or data expected as a return value or result by the user is expressed using the unary* ***return*** *p operator.*

The semantics of WSRL operators are clearly expressed in Table 5.1 and we provide a full Backus-Naur Form(BNF) definition of WSRL in Appendix B.

5.4.1 Examples of WSRL Requests

The medium of expression of requests can either be XML or simply a list of operators with constraints. We demonstrate both these forms of request expression in the following examples.

5.4.1.1 Getting a Train Ticket Booking

The requirements of the user, that must be expressed in this request are as follows :

1. Must get a train ticket from *Barcelona* to *Amsterdam*
2. Ticket must cost less than $300
3. The train company must be *Eurail*
4. Prefer to travel on 30/10/2009, but 31/10/2009 is also fine

WSRL Operator	Purpose
n::*var*	Express the service variable *var* as part of the service namespace *n*
request $\{p_1, ..., p_n\}$	Express ordering of variable constraints p_1 to p_n in the service request
require<*var*> p	Express a mandatory variable constraint p on a service variable of type *var*. Here $var \in \{info, fn, id, pol, impl\}$
if p_1 **then** p_2 [**else** p_3]	Express a conditional binary (**if** p_1 **then** p_2) or ternary (**if** p_1 **then** p_2 **else** p_3) variable constraint
optional p	Express a variable constraint as optional i.e. not mandatory for the satisfaction of the request
prefer p_1 **to** p_2	Express a variable constraint as a preference to another i.e. if p_1 is satisfied then p_2 does not hold, else p_2 needs to be satisfied
return p	Express a variable constraint p on information variables, as a return value for the satisfaction of a service request. A service request is only satisfied if the information variables contained in p are returned to the user

Table 5.1: The Operators of WSRL

5. Any time on these dates is good
6. The booking fee must be less than $10

The corresponding WSRL request as a list of operators with variable constraints is given below :

```
request{

  require<info> Train::Origin = "Barcelona"
                and Train::Destination = "Amsterdam"
                and prefer Train::Date = "30/10/2009"
                       to Train::Date = "31/10/2009"
                and Train::TrainPrice < "$300"

  require<impl> Train::Provider = "EuRail"

  require<pol> Train::Price < "$10"

  require<fn> Train::gotBooking

  return Train::TrainBooking
}
```

We see in this example that all the variables are referenced to the variable namespace *Train* by the **::** operator.

Now we show the WSRL request expressed in XML :

```
<WSRL>
    <REQUEST>
        <REQUIRE varType="info">
            Train::Origin = "Barcelona"
            and Train::Destination = "Amsterdam"
            and prefer Train::Date = "30/10/2009"
                    to Train::Date = "31/10/2009"
            and Train::TrainPrice < "$300"
        </REQUIRE>

        <REQUIRE varType="impl">
            Train::Provider = "Eurail"
        </REQUIRE>

        <REQUIRE varType="pol">
            Train::Price < "$10"
        <REQUIRE>

        <REQUIRE varType="fn">
            Train::gotBooking
        </REQUIRE>

        <RETURN>
            Train::TrainBooking
        </RETURN>
    </REQUEST>
</WSRL>
```

5.4.1.2 Getting a Holiday Package

This example demonstrates how WSRL can be used to express service requests that span multiple services. The requirements of the user, that must be expressed in this request are as follows :

1. Must go for holiday from *Barcelona* to *Amsterdam*
2. Total budget must be less than $3000
3. Prefer to fly, but train travel is also ok
4. If flying, then *Alitalia* is preferred
5. If going by train, *Eurail* is preferred
6. Preferred hotel is *Hilton*
7. Holiday must be between 30/10/2009 and 10/11/2009
8. The holiday booking fee must be less than $100

The corresponding WSRL request as a list of operators with variable constraints is given below :

```
request{

  require<info> Travel::Origin = "Barcelona"
                and Travel::Destination = "Amsterdam"
                and Travel::FromDate = "30/10/2009"
                and Travel::ToDate = "10/11/2009"
                and Travel::Budget < "$3000"

  require<impl> optional Flight::Provider = "Alitalia"
                and optional Hotel::Provider = "Hilton"
                or optional Train::Provider = "EuRail"

  require<pol> Travel::Price < "$100"

  require<fn> prefer Flight::gotBooking to Train::gotBooking
              and Hotel::gotReservation
              and Travel::gotPackage

  return Travel::TravelPackage
}
```

The XML version of the request is as follows :

```
<WSRL>
      <REQUEST>
            <REQUIRE varType="info">
                Travel::Origin = "Barcelona"
                and Travel::Destination = "Amsterdam"
                and Travel::FromDate = "30/10/2009"
                and Travel::ToDate = "10/11/2009"
                and Travel::Budget < "$3000"
            </REQUIRE>

            <REQUIRE varType="impl">
                optional Flight::Provider = "Alitalia"
                and optional Hotel::Provider = "Hilton"
                or optional Train::Provider = "EuRail"
            </REQUIRE>

            <REQUIRE varType="pol">
                Travel::Price < "$100"
            <REQUIRE>

            <REQUIRE varType="fn">
                prefer Flight::gotBooking to Train::gotBooking
                and Hotel::gotReservation
                and Travel::gotPackage
            </REQUIRE>

            <RETURN>
                Travel::TravelPackage
            </RETURN>
      </REQUEST>
</WSRL>
```

We therefore see from these request examples, that WSRL allows a user to express his request clearly in terms of constraints on web service variables. It might seem that the **if** p_1 **then** p_2 **else** p_3 operator is not used, and may not even be required. This is not the case. The conditional operators are provided to allow the user more control in expressing the service request. The user might decide to express a condition that can only be validated at runtime. For e.g. Lets say the user wanted to make use of a special price discount offer given by the *Radisson* hotel in *Amsterdam*, but does not know if the offer is available. If the offer is not available, then he would prefer staying at the *Hilton*. In this case, the request would be expressed as follows :

```
request{

  require<info> Travel::Origin = "Barcelona"
                and Travel::Destination = "Amsterdam"
                and Travel::FromDate = "30/10/2009"
                and Travel::ToDate = "10/11/2009"
                and Travel::Budget < "$3000"

  if require<impl> Hotel::RadissonSpecialPriceOffer = "Available" then
      require<impl> Hotel::Provider = "Radisson"
  else
      optional require<impl> Hotel::Provider = "Hilton"

  require<impl> optional Flight::Provider = "Alitalia"
                or optional Train::Provider = "EuRail"

  require<pol> Travel::Price < "$100"

  require<fn> prefer Flight::gotBooking to Train::gotBooking
              and Hotel::gotReservation
              and Travel::gotPackage

  return Travel::TravelPackage
}
```

We see here that the implementation variable *RadissonSpecialPriceOffer* (given by service provider *Radisson*) is used to express this condition. We have now covered the expression of service requests and must move on to how these requests are processed and the results delivered to the user. The *web service request framework* deals with the entire process.

5.5 The Web Service Request Framework

In previous sections we have seen how information regarding the user's requirements and semantic web service descriptions are modelled. We have also seen how a user expresses a service request. In this section we present the phases that the system is associated with, processes that are part of these phases and algorithms that are part of these processes. We also clearly define how the people associated with the system feature in these phases.

There are two distinct phases associated with the request system :

1. **The Initialisation Phase :** This phase represents the period when design and implementation of the services provided by the system are done, and the user state graph is generated. This is the only phase that requires the involvement of the system designer and the service provider and consists of a single *initialisation process*.

2. **The Operational Phase :** This phase represents the period when the system is operational i.e. accepting requests from users. There are two main processes associated with this phase :

   (a) *The SEPlan Generation Process :* This process is used to generate the service execution plan.

   (b) *The SEPlan Execution Process :* This process is used to execute the generated plan i.e. invoke the services and obtain results to pass on to the user. This process may require involvement of the user.

We now explain in detail, the processes and algorithms involved in each phase.

5.5.1 The Initialisation Phase

As mentioned before, this phase represents request oriented model instantiation by the system designer and service implementation by the service provider. There are however several dependencies in this design phase. We express all these dependencies in Figure 5.15.

We now explain each step of the phase described in Figure 5.15 :

1. The *system designer* elicits user requirements and creates all the identification, information and functionality variables.

2. The *system designer* creates the abstract service description components and finishes the abstract service design.

3. The *service provider* is obtains the abstract service description components from the system and starts implementing services.

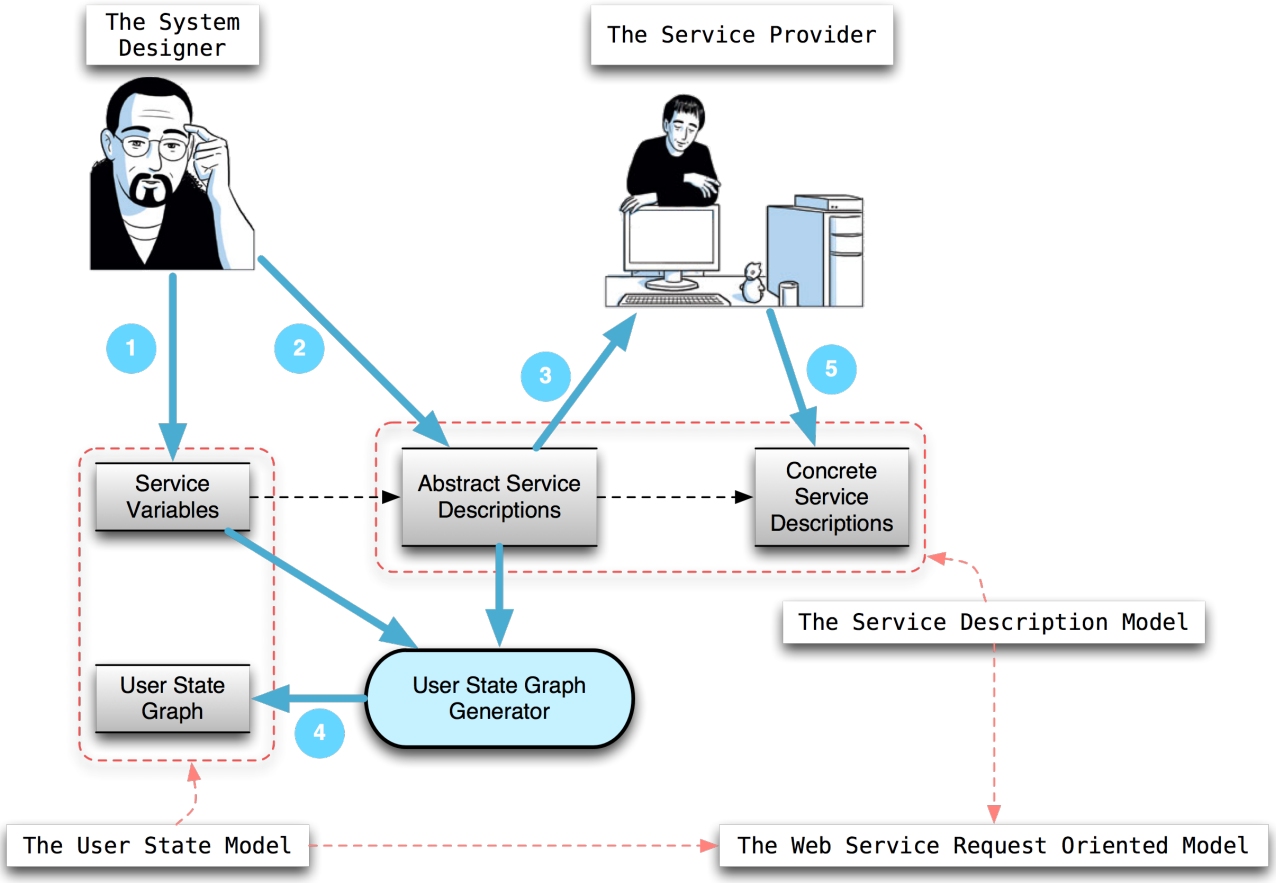

Figure 5.15: The Initial Service Design Phase

4. The **user state graph generator** algorithm is used to generate the user state graph. The inputs required for this algorithm are the abstract service variables that are part of the initial user state and the functionality components.

5. The *service provider* sends the implementation and policy components to the system i.e. sends the concrete service descriptions.

We now describe the user state graph generator algorithm that is involved in this phase and provide an example of its execution. It must be noted that the algorithms described in the following sections have several comments within them to provide clarity to the function calls that are made in the statements. These function calls are assumed to be clear and understandable with minimal complexity and therefore have not been expanded and explained further.

The User State Graph Generator

The user state graph is the most important part of our system as it is required for calculating the goal state path stated in *Definition 9*. The user state graph generator(Algorithm 1) is given the root state , *root* and a list of all the functionality components in the system , $op[]$. It initialises an empty user state graph which has the first state as *root*. It then iterates through each of the functionality components in $op[]$ and checks whether any of them have pre-conditions that are satisfied in the current state (which is the root state on the first iteration).

As an example of this, let us consider the *Train Service* again. Only one functionality component in the train service has a precondition that is met in the root state. This is the **checkAvailability** component whose pre-condition is always *true*. This implies that this component can be invoked at any stage of the process. We are careful in such cases as we also check whether the post-conditions of the component are already satisfied in the current user state. If they are, then we do not use this component to create the next state to avoid cyclic dependencies. We also do this to avoid duplication of user states. The user's request must correspond to only one state in the graph.

If these conditions are met, the functionality component *op* is a candidate for becoming the next user state dependency or graph edge. We apply the post conditions described in *op* to generate the next state. We add this *nextState* to our user state graph node list $USG.V$ and then create a new user state dependency *dep* which is added to the user state graph dependency list $USG.E$. The algorithm is then recursively called with *nextState* as root and the same list of functionality components $op[]$. This recursive call returns a *branch* of the USG which is another USG with a list of nodes, dependencies and root node. This *branch* is then simply added to the the main one by adding its nodes and dependencies to the main node and dependency list. The algorithm then goes to the next iteration and considers the case for the next functionality component. In our *Train Service* example we clearly see that the initial loop creates only one branch for the **getAvailability** component, as the other 2 components **getPrice** and **bookTicket** have preconditions that are not satisfied for the initial *root* state, which does not change with each iteration.

Analysing the time complexity of this algorithm we find that this is a recurrence with time complexity $T(n) = O(n.log(n))$, where n is the number of functionality components that have preconditions satisfied in the root state. Therefore, the algorithm scales well with the increase in functionality components in the system i.e. an increase in the atomic services supported by the system. However, because this is a pre-computing step, it isn't time critical w.r.t the user.

Algorithm 1 USG Generator

Input: *root* /** The Root User State **/ $op[]$ /** List of Functionality Components **/
Output: USG /** The User State Graph **/

```
1: USG ← intUSG(root) /** Initialise USG **/
2: /** For each functionality component in op[] **/
3:
4: for-each op in op[] do
5:    current ← root /** Current user state **/
6:    /** Check if the preconditions are true in current user state **/
7:    if arePreTrue(op, current.var[]) AND arePostTrue(op, current.var[]) = false
      then
8:       varCopy[] = current.var[]
9:
10:      /** Apply the postconditions to the current state's variables **/
11:      current.varFn[] ← applyPost(op, current.var[])
12:
13:      /** Check if the nextState is the same as this state **/
14:      /** If yes, then move on to next op **/
15:      if noChange(varCopy[], current.var[]) then
16:         continue
17:      end if
18:
19:      /** Get a global identifier for the next user state **/
20:      i ← getNewStateIdentifier()
21:
22:      /** Create the next user state **/
23:      nextState ← createNewState(i, current.var[])
24:
25:      /** Add the next state to the USG **/
26:      USG ← addState(USG, nextState)
27:
28:      /** Create a user state dependency **/
29:      dep ← createNewDep(op, root, newState)
30:
31:      /** Add dep to the user state dependency list **/
32:      USG ← addDep(USG, nextDep)
33:
34:      /** Recursively call USG-Generator with nextState as root to get branch **/
35:      branch ← USG − Generator(nextState, op[])
36:
37:      /** Add branch to this USG **/
38:      USG ← addBranch(USG, branch)
39:   end if
40: end for-each
41: return USG
```

5.5.2 The Operational Phase

This phase represents the actual running of the system. As described before, it consists of two important processes, the *SEPlan generation process* and the *SEPlan execution process*. Each of these processes might be implemented as *daemons* that run while the request system is operational. We now look at each of these processes in detail.

5.5.2.1 The SEPlan Generation Process

The purpose of this process is to accept a service request expressed in WSRL from the user, and generate a service execution plan from that request. Figure 5.16 describes the steps involved in this process. We go through each step as follows :

1. The user describes his request in WSRL and sends it over the network to the request system where it is accepted by the **request encoder** algorithm.
2. The request encoder algorithm encodes the WSRL request into a set of service variable constraints as state in *Definition 7* in section 5.4.
3. The encoded service request is passed on to the **request mapper** algorithm, which maps the user state that the functionality variables in the request correspond to. This state is the goal state that the execution plan must eventually reach.
4. The goal state is then given to the **target state path finder** algorithm, which finds a path in the user state graph that takes the root user state to this goal user state.
5. This goal state path is then given to the **execution plan generator** algorithm which uses the concrete service descriptions in the system to create the service execution plan.

We must now formally define what we mean by a goal state path and a service execution plan.

Definition 9 (Goal State Path)
A goal state path for a user state g is defined as a set $P_g = \{D_1, D_2, ..., D_n\}$ *where* D_i *is a user state dependency,* $1 < i < n$ *and* $D_1.S_{pred} = r$ *(root state) and* $D_n.S_{succ} = goal$ *(goal state).*

Definition 10 (Web Service Execution Plan)
A web service execution plan is defined as a set $ExP = \{ExI_1, ExI_2, ..., ExI_n\}$ *where* ExI_i *is an execution item,* $1 < i < n$*. An execution item is defined as a double* $ExI =< Comp_{fn}, Comp_{impl} >$*, where :*

- $Comp_{fn}$ *is a functionality component.*
- $Comp_{impl}$ *is an implementation component that provides concrete service description for* $Comp_{fn}$

We now describe the algorithms that are involved in this phase.

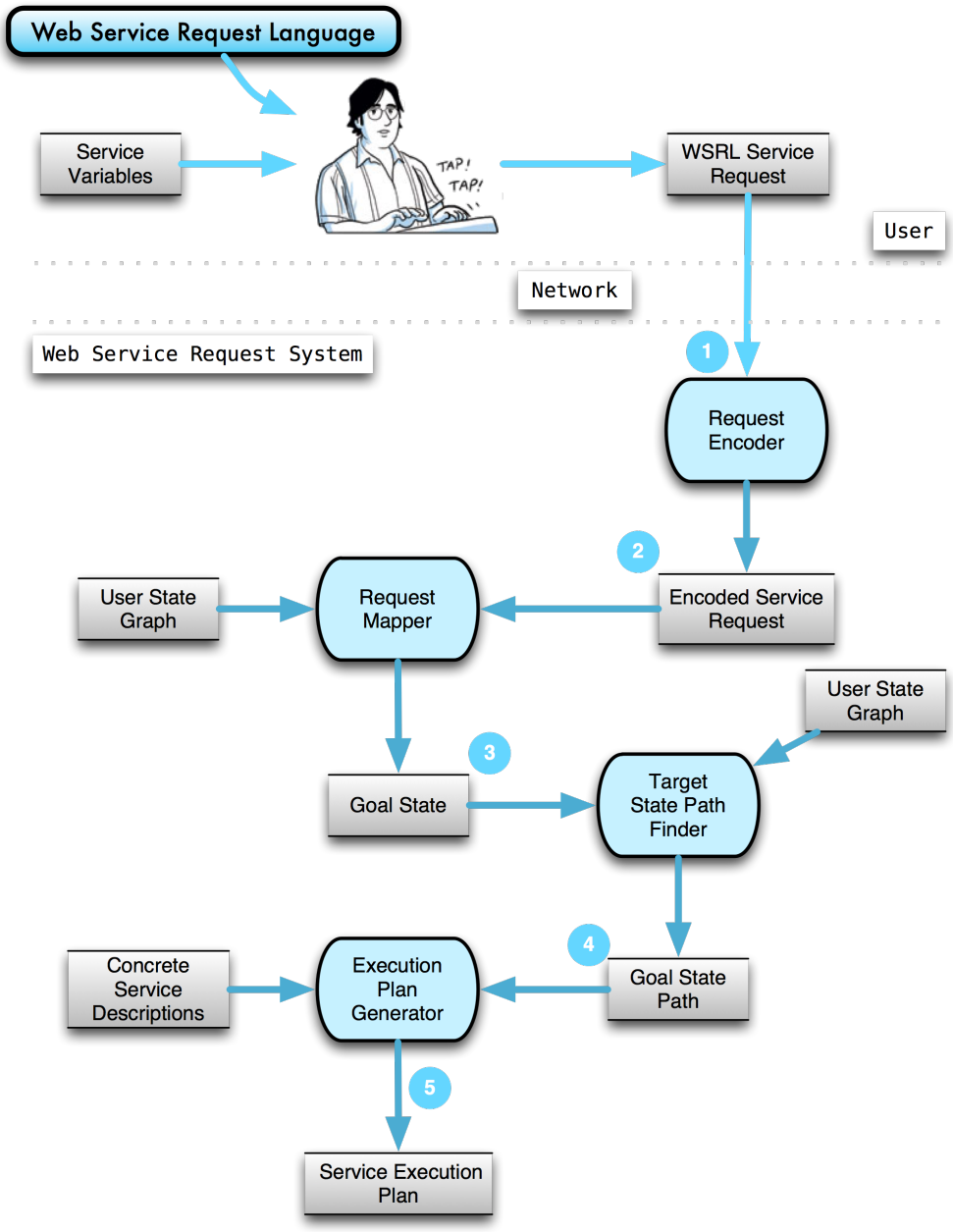

Figure 5.16: The Service Execution Plan Generation Process

The Request Encoder

The request encoder (Algorithm 2) is used to encode the WSRL request provided by the user, into a web service request, stated in *Definition 7*. This is a very trivial and simplistic algorithm as it performs a single pass on the WSRL service request, parsing the request for each type of variable present in the request, and encoding the variables into service variable constraints as stated in Definition 6. It then creates and returns

the service request. Time complexity for this algorithm is simply $O(n)$, where n is the number of variables constrained in the given WSRL request.

Algorithm 2 Request Encoder

Input: UR /** The User Request **/
Output: SR /** The Service Request **/

```
1:  /** Find all "id" variable constraints and add them to id[] **/
2:  id[] ← parse(UR, "id")
3:
4:  /** Find all "info" variable constraints and add them to info[] **/
5:  info[] ← parse(UR, "info")
6:
7:  /** Find all "fn" variable constraints and add them to fn[] **/
8:  fn[] ← parse(UR, "fn")
9:
10: /** Find all "pol" variable constraints and add them to pol[] **/
11: pol[] ← parse(UR, "pol")
12:
13: /** Find all "impl" variable constraints and add them to impl[] **/
14: impl[] ← parse(UR, "impl")
15:
16: /** Get the return info vars and add them to ret[] **/
17: ret[] ← getReturnVars(UR)
18:
19: /** Create the service request **/
20: SR ← createSR(id[], info[], fn[], pol[], impl[], ret[])
21:
22: /** Return the service request **/
23: return SR
```

An example encoded request that represents the WSRL request for train ticket booking specified in section 5.4.1.1 is given in Figure 5.17.

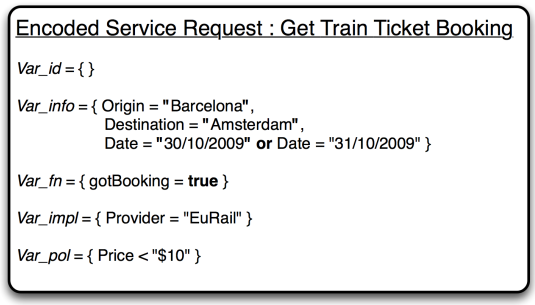

Figure 5.17: An Example Encoded Service Request.

The Request Mapper

This request mapper (Algorithm 3) is used to find the goal user state in a user state graph. This is also a very simple algorithm which goes through each node n in the list of nodes in the graph $USG.V$ and checks whether the functionality variable constraints expressed in the given service request SR apply to the current node (user state). If it does, then it returns the required node G. If none of the nodes are applicable then the algorithm returns an error message ERR. This algorithm has time complexity of $O(n.V)$ where n is the number of functionality constraints in SR, which shall always be trivial w.r.t V which is the number of nodes (user states) in the graph.

Algorithm 3 Request Mapper

Input: SR /** The Service Request **/ USG /** The User State Graph **/
Output: G /** A User State **/

```
1: /** For each node in user state graph **/
2: for-each v in USG.V[] do
3:    G ← v
4:    F ← getFnConstraints(R)
5:    if fnConstraintsApply(F, G) then
6:       return G
7:    end if
8: end for-each
9: return ERR
```

An example goal user state that corresponds to the WSRL request for train ticket booking specified in section 5.4.1.1 is given in Figure 5.18.

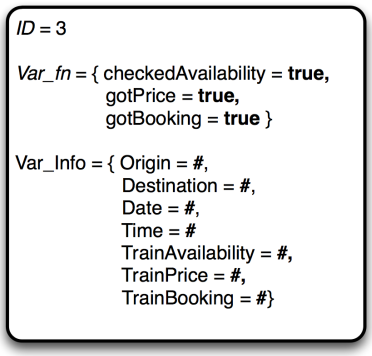

Figure 5.18: An Example Goal State

The Target State Path Finder

This target state path finder (Algorithm 4) is used to find the goal state path, stated in *Definition 9*. It initialises a list of user state dependencies P and then gets the target dependency $target_{dep}$ which points to the given target state $target$. This is a matter of matching state id's and is therefore performed in $O(E)$ time, where E is the number of state dependencies. This is the the dependency that corresponds to the last dependency in the final goal state path, and is now added to P. The algorithm then backtracks its way from the first node of $target_{dep}$ i.e. S_{pred} to the root state. In each step of the way, the next dependency is added to list P. When it is done, P contains the goal state path in reverse order. A simple reversal of order is done and P is returned. We see here that the worst case time complexity for this algorithm is $O(E^2)$.

Algorithm 4 TS-Path-Finder

Input: *root* /** The Root User State **/ *target* /** The Target User State **/
Output: P /** The Target State Path **/

1: /** Backtrack from the target state by looking at the user state dependencies **/
2: /** Initialise a user state dependency list to hold the target state path **/
3: $P \leftarrow initDependencyList()$
4:
5: /** Locate the target dependency in the list **/
6: $target_{dep} \leftarrow locateDepInList(USG.E[], target, "Successor")$
7: /** Add the dependency to the list **/
8: $P \leftarrow addDepToList(P, target_dep)$
9: /** Backtrack until the root user state is reached **/
10: **while** $target_{dep}.S_{pred} \neq root$ **do**
11: $target_{dep} \leftarrow locateDepInList(USG.E[], target_{dep}.S_{pred}, "Successor")$
12: $P \leftarrow addDepToList(P, target_{dep})$
13: **end while**
14: /** Reverse the order of P **/
15: $P \leftarrow reverseOrderOfDependencyList(P)$
16: **return** P

The example target state path that corresponds to the goal state given in Figure 5.18 is given below in Figure 5.19.

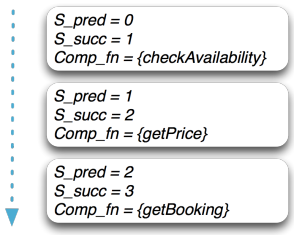

Figure 5.19: An Example Goal State Path

The Service Execution Plan Generator

The service execution plan generator (Algorithm 5) uses the goal state path, implementation & policy components in the system, and the service request to generate a service execution plan as stated in *Definition 10*. The execution plan Exp is first initialised and then the algorithm iterates through each state dependency in the goal state path P. In each iteration checks are made to see if any implementation and policy constraints have been specified in the service request SR. If there are any such concrete service constraints then an implementation component is searched for in $Impl[]$, which satisfies these constraints. An example of this is given in our train ticket booking request. The constraints $Provider = EuRail$ and $Price < \$10$ are specified and hence the algorithm finds the implementation components supplied by the company that books *EuRail* train tickets and which offers the service for less than $10.

If no such concrete service variable constraints are specified, then any implementation component is chosen that provides the functionality of $fnComp$. The implementation component $implComp$, along with its corresponding functionality component $fnComp$ is composed into an execution item $ExpItem$, which is then added to the plan Exp. After iterating through all the state dependencies in P, the algorithm returns Exp. We see here that the worst case time complexity for this algorithm is $O(E.n)$ where n is the number of implementation components in the system and E is the number of nodes in the USG or the maximum length of P.

The example execution plan that corresponds to the goal state path in Figure 5.19 is given below in Figure 5.20.

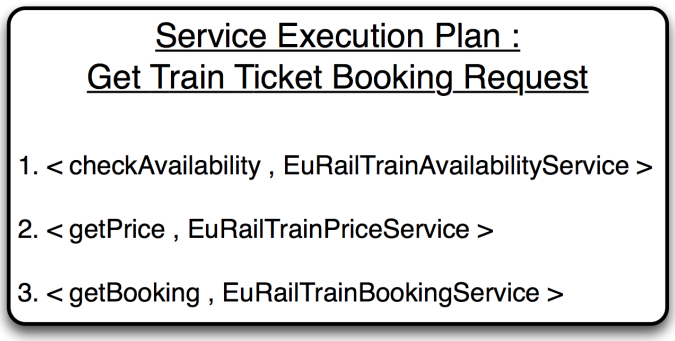

Figure 5.20: An Example Service Execution Plan

Algorithm 5 Execution-Plan-Generator

Input: $P, Impl[], Pol[], SR$

Output: ExP **/**** The Service Execution Plan - An ordered list of functionality and implementation component doubles ****/**

1: **/**** Get the impl and pol constraints from the SR ****/**
2: $ImplConst[] \leftarrow getImplConstraints(SR)$
3: $PolConst[] \leftarrow getPolConstraints(SR)$
4: $Exp \leftarrow initExp()$
5: **/**** For each state dependency in the goal state path, get the fn component associated ****/**
6: **for-each** p in P **do**
7: **/**** Get the associated functionality component ****/**
8: $fnComp \leftarrow getFnComp(p)$
9:
10: **/**** Check whether any impl var constraints and pol var constraints are present ****/**
11: **if** $ImplConst[] \neq$ empty OR $PolConst[] \neq$ empty **then**
12: **/**** Get the impl comp. that agrees with the constraints and corresponds with the fn comp ****/**
13: $implComp \leftarrow getImplComp(fnComp, Impl[], Pol[], ImplConst[], PolConst[])$
14: **else**
15: **/**** Get any impl comp that correspond with the fn comp ****/**
16: $implComp \leftarrow getImplComp(fnComp, Impl[])$
17: **end if**
18: **/**** Create an Execution Plan Item from these two components ****/**
19: $ExpItem \leftarrow createExpItem(fnComp, ImplComp)$
20: **/**** Add it to ExP ****/**
21: $Exp \leftarrow addToExp(ExpItem)$
22: **end for-each**
23: **return** ExP

5.5.2.2 The SEPlan Execution Process

This process is responsible for invoking the web services according to the service execution plan, and obtaining results from them. In the case that the invoked services do not return proper results, this process sends an error message to the user as described in Figure 5.17. We shall now go through this process step by step.

1. A service execution plan generated by the SEP generation process is passed to the **SEPExecutor** algorithm.
2. This step is non-deterministic as it involves invocation of web services. In the case that the invoked services return a valid result, it is passed on to the user and the process ends. Whereas in the case that one of the invoked services fails to return a valid result, an error message is generated, which is passed on to the user.
3. If the user decides to re-send a different WSRL request in response to the previous error message, he sends it to the SEP Generation process.
4. Another SEP is generated and the process goes back to step 1.

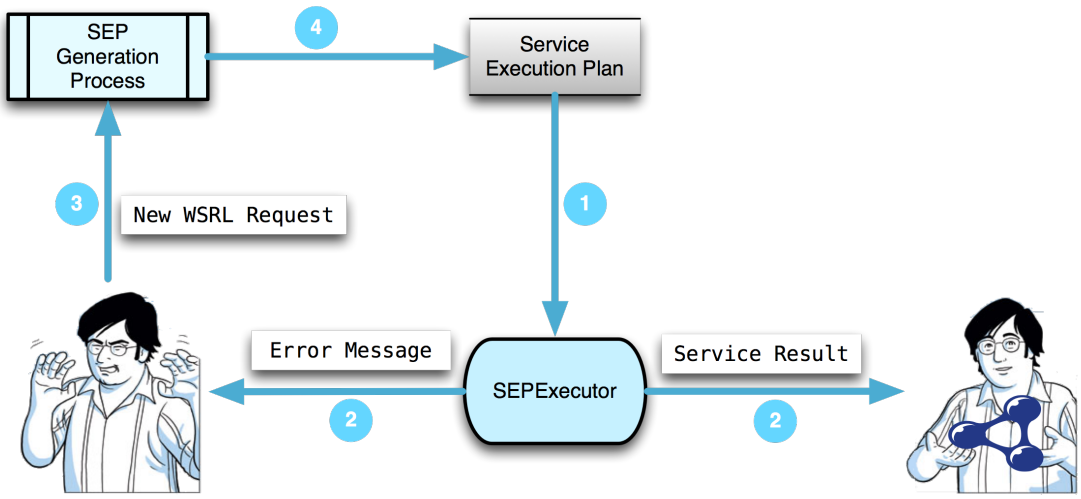

Figure 5.21: The SEPlan Execution Process

We shall now move on to describing the SEPlan Executor Algorithm used to invoke services according to the service execution plan.

The SEPlan Executor

The SEPExecutor (Algorithm 6) is responsible for invoking web services to obtain results based on a service execution plan. The result of this algorithm is often non-deterministic. This is because it is responsible for actually invoking web services, which might not always return correct results. In fact, they might not return any results at all, in which case the algorithm stalls. A timeout must be specified on service invocations in the actual implementation of the algorithm, but it is not specified here. This algorithm also supports dynamic lookups for other service implementations that provide the same functionality in case the invoked services return errors.

The inputs to this algorithm are the information components $Inf[]$, the implementation components $Impl[]$, the user state graph G, the service execution plan SEP and the service request R. Two counters are initialised in this algorithm. The *errorCounter* counts the number of times a service invocation returns an error instead of a result. and the *changeCounter* counts the number of times a different service implementation is invoked to get the same functionality. Maximum thresholds are also defined for each of these counters.

The algorithm runs until each execution item $SEPitem$ in SEP has been used to invoke a service and get a successful result. For each execution item, the corresponding information component $infoComp$ that describes what messages are required as inputs for the $fnComp$ in the item, is obtained. The $SOAP$ messages corresponding to the input information variables in $infoComp$ are created in $soapMsgs[]$. The service implementation described in $SEPItem.implComp$ is then invoked by sending the SOAP input messages in $soapMsgs[]$ to the network address specified in $SEPItem.implComp$.

At this stage we determine whether to invoke a service from a different provider. If the previous invocation has returned an error message in Res then the errorCounter is incremented and the the rest of the iteration is skipped to go to the next iteration. The $SEPItem$ is the same in this iteration as it hasn't yet been popped from SEP. Therefore if this happens for $maxErrors$ times we change the service provider at line 18 of the algorithm. The change counter counts exactly how many times changes have been made. If the $changeCounter$ reaches $maxImplChanges$ then the algorithm returns an error ERR.

If no errors are encountered and Res has the required result for $SEPItem$, then we add the info in Res to user state graph G. This is the stage where the information variables in the user state graph actually get values. If a change to any information variable is made, we check whether the changed value agrees with the constraints that the user has already mentioned in the service request R. For example, if the train price returned is \$400, then it does not agree with the user constraint $TrainPrice < \$300$ in our *train ticket booking* request example. In such a case, an error ERR is returned to the user which explains that the requirements could not be met. Another possibility

at this stage could be the changing of service providers to check better rates, but we have not included that in our algorithm.

If the changes made to information variables agree with the constraints in R, then the $SEPItem$ is popped from the SEP and the next $SEPItem$ is handled. At the stage when the SEP has no more items i.e. it is exhausted, the results specified as desired return values in the service request $R.Ret$ are obtained from the USG and returned to the user.

We can only measure the time complexity of a deterministic execution of this algorithm i.e. when no web services do not return any value or error message. The complexity for this case is $O(n.V)$, where V is the maximum number of items in SEP and n is the number of implementation components in the system.

5.5.3 Algorithm Time Complexities

We can see in Table 5.2 that the worst case time complexities of the algorithms scale well (i.e. linearly) with the increase in number of atomic services in the system. This implies that the system shall be able to support an increasing number of services provided the adequate hardware requirements are met (for running the system implementation).

Algorithm 6 SEPExecutor

Input: $Inf[], Impl[], USG, SEP, R$
Output: Res /** The Result **/

1: $errorCounter \leftarrow 0, changeCounter \leftarrow 0, maxErrors \leftarrow 3, maxImplChanges \leftarrow 3$
2: **while** $SEP \neq empty$ **do**
3: $SEPitem \leftarrow getNextItem()$ /** Get next item in SEP **/
4: $infoComp \leftarrow getCorrInfoComp(SEPitem.fnComp, Inf[])$ /** Get info comp **/
5: /** Determine which information variables from the service request represent the input messages for this item **/
6: $msg[] \leftarrow getInputMessages(infoComp, R)$
7: /** Construct the SOAP input messages for this item **/
8: $soapMsgs[] \leftarrow constructSOAPmsgs(msg[])$
9: /** Invoke the service by sending the SOAP message to the required network address **/
10: $Res \leftarrow invokeService(SEPItem.implComp, soapMsgs[])$
11: /** If the service returns an error message then increment error counter and continue to next iteration **/
12: **if** $isErrorMsg(Res)$ **then**
13: **if** $errorCounter < maxErrors$ **then**
14: $errorCounter \leftarrow errorCounter + 1$
15: $continue$
16: **else**
17: **if** $changeCounter < maxImplChanges$ **then**
18: $SEP \leftarrow changeImplComp(SEPitem)$
19: $changeCounter \leftarrow changeCounter + 1$
20: $continue$
21: **else**
22: **return** ERR
23: **end if**
24: **end if**
25: **end if**
26: $errorCounter \leftarrow 0, changeCounter \leftarrow 0$
27: /** Add the info returned to the next user state in the USG **/
28: $USG \leftarrow addInfoToNextState(SEPItem, Res, P)$
29: /** Check whether any information variables have been changed w.r.t the previous state **/
30: **if** $anyInfoChange(USG)$ **then**
31: /** Check whether the change is good w.r.t the info variable constraints in the service request **/
32: **if** NOT$(isChangeGood(R, USG))$ **then**
33: /** if change not good return error message stating the inconsistency **/
34: **return** ERR
35: **end if**
36: **end if**
37: /** Remove the current item from the SEP **/
38: $popItem(SEP)$
39: **end while**
40: **return** $results(USG, R.Ret)$

Algorithm	Time Complexity	Worst Case Analysis
User State Graph Generator	$O(n.log(n))$	n is the number of functionality components that have preconditions satisfied in the root state. In the worst case n = total number of functionality components in the system
Request Encoder	$O(n)$	n is the number of variables constrained in a WSRL request. In the worst case, n = total number of variables in the system, but this is very very unlikely.
Request Mapper	$O(n.V)$	n is the number of functionality variable constraints in a WSRL request and V is the number of nodes in the user state graph. In the worst case, n is the total number of functionality variables in the system.
Target State Path Finder	$O(E^2)$	E is the number of user state dependencies in the user state graph. This is the worst case complexity.
Service Execution Plan Generator	$O(n.E)$	n is the number of implementation components in the system and E is the number of nodes in the user state graph. This is the worst case complexity.
SEPlan Executor	$O(n.V)$	n is the number of implementation components in the system and V is the maximum number of items in the SEPlan. In the worst case V = total number of nodes in the user state graph.

Table 5.2: Worst Case Time Complexity Analysis of all Algorithms

Chapter 6

Conclusion & Future Work

In this thesis we presented a new concept of a web service request system that serves as a single entry point for service consumers to the web service domain. It provides an approach that greatly increases the *usability* of web services, provides enough *flexibility* to users of the system in terms of requesting multiple services, and provides enough *expressiveness* to users for expressing their requests. We have presented a request language (WSRL) based on which users can express service requests. We have created a request oriented model for web services, that represents services from the view-point of service requests. We have also presented the framework responsible for handling the service requests and returning results to the users.

Our work meets all the needs of a request system that we initially described in *Section 2.6* and therefore realises our vision of the request system given in Figure 2.5. The research challenges mentioned in *Section 2.6.1* have also been addressed by our work. Our attempt at minimising time complexities of algorithms required to process requests, has also been successful. This implies that we have managed to design a system that can actually be used in the real world.

We see that our system is the first step in realising the service oriented and cloud computing vision of a service cloud which we described in Figure 2.1. What our system has achieved is an approach to creating *semantic web services* rather than regular ones. Our concentration on creating solid models to represent data in the system also ensure that there aren't any inconsistencies in understanding for any of the people associated with the system i.e. the user, system designer and service provider. By providing full semantics of WSRL and proposing it as the language of choice for expressing service requests, we have also potentially addressed the business to business transaction problem that the Internet has today. That is a definite step forward for service oriented computing.

Our approach involves the user in the service invocation process when non-deterministic failures may occur. This ensures that the system's method of handling non-determinism is well defined and the user makes the final call in case of any excessive errors in service invocation. Apart from this limited interaction, the user of the system is essentially abstracted from the entire web service domain. If proper user interfaces are im-

plemented that make the request expression process intuitive and expressive enough, the users can potentially treat this system interface as a regular internet market website. The only difference to a normal website in this case is that the system is abstracted from any specific business and acts as an intelligent service aggregator that spans multiple business domains.

Graph path finding provides a sound, easy, efficient and practical approach to creating service execution plans. Our initial plan of adopting a graph based approach in the system, similar to the ones provided by [Bouguettaya and Yu 2008] was also followed. We have also provided enough options for business collaboration via our system at an atomic service level. The abstract services provided by the system may be implemented by many service providers and therefore many service implementation choices also potentially exist for users of the system. It must also be noted that our approach cuts out the need of having another service registry such as the UDDI which must be queried to obtain relevant services. This significantly cuts out communication time between the system and an external registry and hence provides better QoS to users of the system with fast transactions. The implementation and policy components provided to the system by service providers creates an internal service registry in the system which also contains semantic information. Therefore, querying our internal registry for particular implementation components shall have minimal time complexity.

Our work has concentrated on the design of the system and the underlying models for knowledge representation and language expression. Although, the implementation of the system had begun, time limitations did not permit completion. Therefore, in order for the system to be realised in the future, the following work elements must be carried out :

- *Full implementation* : This is imperative to the acceptance of the system in the web services community and provides for proper analysis of the time complexity of the algorithms.
- *User Interface* : This is extremely important as it shall determine how a majority of Internet users may use the system.
- *Testing of the system* : The implementation of the system must be tested in several ways including :
  - Unit testing : Test each algorithm in isolation.
  - Integration testing : Test each process and then all processes together.
  - Scalability testing : Test the scalability of the system i.e. how does the implementation compare to the theoretical scaling analyses.
  - User Interface testing : Test the user interface of the system by allowing actual users, experimentation with it

- *Possible Improvement of Design* : The results from each phase of testing the system shall give enough scope for improvement of the design if required.
- *Reusing SEP's for common requests* : This is also an important topic that must be approached. Majority of Internet marketplace users require similar products and are highly likely to send the same or very similar service requests to the system. Research must be done to create algorithms that determine which SEP's are worth storing in the system so that they can be readily reused.
- *BPEL Specification Generator* : The next step for the system would be to be able to generate BPEL4WS specifications that could be automatically reused in the future for corresponding service requests. This would also minimise time complexity by cutting out the plan generation phase.
- *Algorithm Optimisation* : The testing of the system would provide feedback on the performance of the algorithms and hence help in optimising them. In particular the optimality of the generated service execution plan would be a crucial one to approach first. Time optimality of all algorithms also need to be achieved.
- *Industry Acceptance of WSRL* : After the potential of the system is proven by testing, WSRL and WSRS must be put out for acceptance by the web service community and in particular the *World Wide Web Consortium* (W3C).

In conclusion, we would like to say that the *Web Service Request System* has enormous potential in the Internet marketplace, furthers the cause of service oriented and cloud computing and deserves to be implemented, optimised, tested and deployed in the future.

Appendix A

High Level Requirements Specification

Description of Project

This requirements specification was created during the literature review period in semester 1 and reflects the initial goals of the project. The initial description of the project is as follows : *There is a need to design and develop a service-centric request infrastructure that goes beyond keyword-based, UDDI-based service discovery. Querying or requesting services involves complex structures, semantics, and relationships that cannot be supported by keyword-based querying. We propose to design and implement a service request language that leverages ontologies to better map user service queries or requests to the available service space.*

General Requirements

WSRL shall :

- be created for a normal user.
- be intuitive and easy to use for a normal user.

Functionality

WSRL shall :

- be created for a generic service domain.
- allow the user to query the service space based on
    1. Service functionality
    2. Service quality
- allow the user to invoke service operations
- allow the user to query multiple services

- have dynamic service composition facilities
- shall have service query optimisation facilities
- shall provide a formal service model
- shall have options to include a service domain ontology or a set of service domain ontologies
- shall be able to generate service work flows
- shall be able to generate service execution plans.

Attributes

WSRL shall consist of :

- The language constructs and grammar
- The query processor for parsing WSRL queries
- The framework for generating service execution plans (or work-flows)

Appendix B

WSRL in Backus-Naur Form

Here we define the WSRL notation we use in the form of BNF grammar. We make the following assumptions :

- *e*... represents multiple elements of type e
- [*e*] denotes optional element e
- $< e >$, $< /e >$ denote starting and ending XML tags

The formal WSRL language syntax is defined as follows :

```
wsrl              ::=  <WSRL> request </WSRL>

request           ::= <REQUEST> variable_constraint... </REQUEST>

variable_constraint ::= <REQUIRE varType> proposition </REQUIRE>

varType           ::= id | info | fn | impl | pol

proposition       ::= <BOOL> true | false  </BOOL> |
                      <AND> proposition... </AND> |
                      <OR>  proposition... </OR> |
                      <NOT> proposition... </NOT> |
                      <GREATER>  lval       </GREATER>
                        <THAN>   rval       </THAN> |
                      <LESS>     lval       </LESS>
                        <THAN>   rval       </THAN> |
                      <EQUAL>    lval rval  </EQUAL>

prefer a to b     ::= <PREFER> a
                      <TO> b </TO>
                      </PREFER>

optional a        ::= <OPTIONAL> a </OPTIONAL>

return a          ::= <RETURN> a </RETURN>
```

```
if a then b [else c] ::= <IF> a
                         <THEN> b </THEN>
                         <ELSE> c </ELSE>
                         </IF>

namespace::v     ::= <NS> namespace
                     <VAR> v </VAR>
                     </NS>

lval             ::= a...zA..Z[rval]
rval             ::= a..zA..Z0..9$
v                ::= a..zA..Z0..9
namespace        ::= a..zA..Z0..9
a                ::= variable_constraint
b                ::= variable_constraint
c                ::= variable_constraint
```

Appendix C

Glossary of Acronyms

- **HTTP :** Hyper text transfer protocol
- **JMS :** Java message service
- **SMTP :** Simple mail transfer protocol
- **SOAP :** Simple object access protocol
- **XML :** Extensible markup language
- **WSDL :** Web service description language
- **UDDI :** Universal description, discovery and integration
- **BPEL4WS / WS-BPEL :** Business process execution language for web services
- **CDL4WS / WS-CDL :** Choreography description language for web services
- **WSMO :** Web service modelling ontology
- **OWL :** Web ontology language
- **OWL-S :** OWL for web services
- **WSRL :** Web service request language
- **WSRS :** Web service request system
- **SEP :** Service execution plan
- **USG :** User state graph

Bibliography

ALONSO, G., CASATI, F., KUNO, H., AND MACHIRAJU, V. 2003. Web services. *Springer-Verlag*. (p. 12)

AMAZON. 2008. Amazon elastic compute cloud (ec2). *http://www.amazon.com/ec2/*. (p. 10)

ANDREWS, T. 2003. Business process execution language for web services. *http://www.ibm.com/developerworks/library/ws-bpel*. (pp. 2, 23)

BAJAJ, S. 2006. Web service policy framework (WS-Policy) version 1.2. *http://xml.coverpages.org/ws-policy200603.pdf*. (p. 22)

BELLWOOD, T., CLEMENT, L., EHNEBUSKE, D., HATELY, A., HONDO, M., HUSBAND, Y., JANUSZEWSKI, K., LEE, S., MCKEE, B., MUNTER, J., AND RIEGEN, C. V. 2002. UDDI version 3.0. *http://uddi.org/pubs/uddi-v3.00-published-20020719.htm*. (p. 20)

BOAG, S., CHAMBERLIN, D., FERNANDEZ, M., FLORESCU, D., ROBIE, J., SIMEON, J., AND STEFANESCU, M. 2002. XQuery 1.0: An XML query language. *W3C working draft 15*. (p. 29)

BOOTH, D., HAAS, H., MCCABE, F., NEWCOMER, E., CHAMPION, M., FERRIS, C., AND ORCHARD, D. 2004. Web services architecture. *http://www.w3.org/TR/ws-arch/*. (p. 41)

BOUGUETTAYA, A. AND YU, Q. 2008. Framework for web service query algebra and optimization. *ACM Trans. Web 2, 1, Article 6*. (pp. 4, 5, 31, 33, 82)

BUYYA, R., YEO, C. S., AND VENUGOPAL, S. 2008. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC-08, IEEE CS Press, Los Alamitos, CA, USA), Sept. 25-27, 2008*. (p. 10)

CABRERA, L. F. 2005a. Web service atomic transaction: (WS- AtomicTransaction). *http://schemas.xmlsoap.org/ws/2004/10/wsat*. (p. 23)

CABRERA, L. F. 2005b. Web service coordination: (WS-Coordination). *http://schemas.xmlsoap.org/ws/2004/10/wscoor/*. (p. 23)

CHRISTENSEN, E., CURBERA, F., MEREDITH, G., AND WEERAWARANA, S. 2001. Web services description language (WSDL) 1.1. W3C. (p. 16)

GOOGLE. 2008. The google app engine. *http://appengine.google.com/*. (p. 10)

JIN, L. J., MACHIRAJU, V., AND SAHAI, A. 2002. Analysis on service level agreement of web services. *Technical Report HPL-2002-180, Software Technology Laboratory,*

HP Laboratories Palo Alto. available at : www.hpl.hp.com/techreports/2002/HPL-2002-180.pdf. (pp. 20, 21)

LAGO, U. D., PISTORE, M., AND TRAVERSO, P. 2002. Planning with a language for extended goals. *Proceedings AAAI'02*. (p. 29)

LAZOVIK, A. 2006. *Interacting with Service Compositions*. PhD thesis, International Doctorate School in Information and Communication Technologies (ICT), Trento University. (p. 31)

MANI, A. AND NAGARAJAN, A. 2002. Understanding quality of service for web services. *IBM developerWorks*. available at : http://www.ibm.com/developerworks/library/ws-quality.html. (p. 21)

MARTIN, D., BURSTEIN, M., HOBBS, J., LASSILA, O., MCDERMOTT, D., MCILRAITH, S., NARAYANAN, S., PAOLUCCI, M., PARSIA, B., AND PAYNE, T. 2004. Owl-s: Semantic markup for web services. *http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/*. (p. 24)

MCGUINNESS, D. AND HARMELEN, F. V. 2004. Owl web ontology language overview. *W3C recommendation*. (p. 24)

MICROSOFT. 2008. Microsoft live mesh. *http://www.mesh.com/*. (p. 10)

PAPAZOGLOU, M. P. 2008. *Web Services: Principles And Technology*. Pearson Education. (pp. 9, 12, 13, 51, 56)

PAPAZOGLOU, M. P., AIELLO, M., PISTORE, M., AND YANG, J. 2002a. XSRL: a request language for web services. *Internet Computing, IEEE*. (pp. 4, 29, 30)

PAPAZOGLOU, M. P., AIELLO, M., PISTORE, M., AND YANG, J. 2002b. XSRL: an xml web-service request language. *Internet Computing, IEEE*. (p. 29)

PAPAZOGLOU, M. P., AIELLO, M., PISTORE, M., YANG, J., CARMAN, M., SERAFINI, L., AND TRAVERSO, P. 2002. A request language for web-services based on planning and constraint satisfaction. *Lecture Notes in Computer Science 2444*, 76–85. (p. 29)

PAPAZOGLOU, M. P. AND GEORGAKAPOULOS, G. 2003. Introduction to the special issue about service-oriented computing. *Communications of the ACM 46*, 10 (October), 24–8. (p. 9)

PELZ, C. 2003. Web services orchestration and choreography. *Web Services Journal*. (pp. 2, 23)

ROMAN, D., KELLER, U., LAUSEN, H., BRUJIN, J. D., LARA, R., STOLLBERG, M., POLLERES, A., FEIER, C., BUSSLER, C., AND FENSEL, D. 2005. Web service modeling ontology. *Applied Ontology, Vol. 1, Issue 1, Pg 77-106*. (pp. 24, 25)

W3C. SOAP version 1.2 part 1. *http://www.w3.org/TR/soap12-part1/*. (p. 15)

W3C. 2001. Wsdl 1.1. *http://www.w3.org/TR/wsdl*. (pp. 17, 18)

W3C. 2003. SOAP version 1.2 part 0: Primer. *http://www.w3.org/TR/2003/REC-soap12-part0-20030624/*. (p. 15)

W3C. 2004. Rdf. *http://www.w3.org/RDF/*. (p. 24)