2007

Implementation of a Web Service Synthesis system based on query rewriting (Web Service Specification, Signature, Query rewriting, Query planning, Service plan)

Chang Zhou
University of Windsor

Implementation of a Web Service Synthesis System Based on Query Rewriting

By Chang Zhou

A Thesis
Submitted to the Faculty of Graduate Studies
Through Computer Science
In Partial Fulfillment of the Requirements for
The Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2007

Your file Votre référence
ISBN: 978-0-494-34922-9
Our file Notre référence
ISBN: 978-0-494-34922-9

Abstract

This thesis describes the implementation of a Web Service Synthesis system, which synthesizes the implementation of a web service from its specification. A web service specification consists of a syntactic part, which follows WSDL style, and a semantic part, which is a Datalog clause. The synthesizer has access to a repository of web services; each contains both syntactic and semantic descriptions. Web services in this thesis are restricted to be the ones corresponding to database queries. In the semantic specifications of web services, the Datalog clauses or SQL queries contain tables from global schemas only. An implementation is correct with respect to its specification if the answer set of the implementation is the same as the specification. To generate the implementation, we apply query planning to generate a service execution plan and its execution codes. We implemented the prototype system that can automatically generate the Java code that implements a web service.

Keywords: Web Service Specification, Signature, Query, Query Rewriting, Query Planning, Service Plan

Acknowledgements

First I would like to thank my advisor, Dr. Jianguo Lu, for his constant encouragements, guidance and helps on my work. Dr. Lu's expertise and professional attitudes helped me to push myself to a higher level for excellence. I truly appreciate his great patience on me during this work.

I would like to extend my appreciation to my colleague, Minghao Li, for his excellent work on Query Rewriting research and the development of Query Rewriting software module. Also, I would like to thank Ms. Chunjiao Ji for her precious advices and discussions on my thesis.

Most of all, I would like to express my special gratitude to my family. Specially I thank my wife, Ms. Zhewei Chen, for her unconditional support and love on me. I also thank my mother, Ms. Ruihua Xia, for her great love and being around to help and support in my family.

Table of Contents

Abstract I
Acknowledgements II
List of Figures V
1. Introduction 1
2. Web Service Specification 7
2.1 What is Web Service Specification? 7
2.2 Global Schema 10
2.3 Signatures in the specification 12
2.4 Semantics of web service 13
2.4.1 Datalog 13
2.4.2 Parameterized Query in Datalog 15
2.4.3 The Query in Web Service Specification 16
2.5 Summary of Web Service Specification 17
3. Web Service Synthesis 19
3.1 Web Service Synthesis and the System Architecture 19
3.1.1 Web Service Synthesis 19
3.1.2 System Architecture 23
3.2 Query Rewriting Module 27
3.2.1 Query Rewriting Using Views 27
3.2.2 Query Rewriting System Implementation 30
3.3 Service Planning Module 33
3.3.1 Query Planning 34
3.3.2 Service Planning Algorithm 35
3.3.3 The Service Plan 37
3.4 Service Composition Module - Concrete Implementation 38
3.4.1 Overview 39
3.4.2 Implementations for Service Composition 42
3.4.2.1 Overall Service Plan Execution 42
3.4.2.2 Component Service Invocation 45
3.4.2.3 Result Extraction and Join operation 50
3.4.3 Summary of the concrete implementation 51
4. Experiments 53
4.1 Experimental Web Service Repository 53
4.2 Constructing Web Service Synthesis Environment 54
4.3 Test Web Service Synthesizer 55
4.4 Publish the web service 57
4.5 Experimental Results 58
5. Conclusion and Future work 59
5.1 Conclusion 59
5.2 Contribution 60
5.3 Future Work 61
Appendix 1: A Experimental Global Schema 63
Appendix 2. Two views: "v_amazon" and "v_shipping" 65
Appendix 3: WSDL of "Amazon" web service 66

Appendix 4: WSDL of "Shipping" web service 68
References 70
Vita Auctoris 76

List of Figures

Figure 1: The framework of the service composition system 2
Figure 2: Mediator Based Approach 3
Figure 3: Zeng's Approach 4
Figure 4: Example and Illustration 5
Figure 5: Web Service and Web Service Specification 8
Figure 6: Web Service Specification of Amazon service 9
Figure 7: Global Schema 11
Figure 8: Shipping Service and its query in Datalog 15
Figure 9: Missing access capabilities on parameters 15
Figure 10: Query with Binding Patterns 16
Figure 11: Web Service Specification 18
Figure 12: Synthesize a new service from specification "BookShippingFee" 21
Figure 13: Web Service Synthesis Process Flow 23
Figure 14: Web Service Synthesis Overall Picture 25
Figure 15: Two View Definitions 29
Figure 16: A User Query 29
Figure 17: Equivalent Rewriting of Q 30
Figure 18: Query Rewriting System 31
Figure 19: Service Planning Algorithm 36
Figure 20: Web Service Composition Module 38
Figure 21: Generated Java Core Class 40
Figure 22: Class Template for Synthesis Service 40
Figure 23: Codes to Generate the Class Template of Synthesis Service 41
Figure 24: The generated class for Q(ISBN, ZIP2, FEE) 41
Figure 25: Codes to generate the return type class 42
Figure 26: Code Generation Mapping with Service Plan 44
Figure 27: Abstracted Amazon Service Operation 46
Figure 28: Classes Generated from Amazon's WSDL 47
Figure 29: Classes Generated from Shipping's WSDL 47
Figure 30: Codes Segment of Amazon service 49
Figure 31: Codes Template 49
Figure 32: Generation of Implementation Codes 50
Figure 33: Result Extraction while Executing Services 51
Figure 34: Web Service Synthesis system 56
Figure 35: Generated Java Codes 57
Figure 36: Service plan tree 62

1. Introduction

Web Service is an application that is accessible to other applications over the Web. It defines a programmable interface between applications [3], and is published on the Web. Web Service consists of a series of technologies and concepts. W3C is establishing industry standards for Web Service technologies, such as web service description language WSDL [4], web service registration standard UDDI [5], and messaging standard SOAP [6]. Web service technologies are expected to become the standard for Web applications in the future.

One goal of using web services is to consume and integrate them over the Internet, instead of consuming them separately as standalone services. Having many web services running on the Internet, we need an easy and feasible method to automatically integrate existing web services and perform more complex user's requests. In many situations web service consumers don't know what web services are available on the Web, and how they should be integrated. User only knows a function to be performed or a specific query to be answered by some web services on the Web. There should be a system to automatically search for relevant services available, invoke these services, and integrate the services according to user's requirements.

To solve this problem, a lot of researches have been done in the area of automated web service composition. A general framework of the service composition system was proposed in [43], which is illustrated in Figure 1. The *service requester* submits a service specification to describe the information or services he needs; and the *service provider* proposes the web services for use. The *translator* translates between *external specification* and *internal specification*. The *process generator* tries to find out if existing services can fulfill the requester's service specification. If yes, the *process generator* generates one or more *service plan* to compose available services in the *repository*. When multiple processes are generated, the evaluator evaluates all service plans and finds the best one for execution. The execution engine executes the generated service plan and returns the results to the service provider.

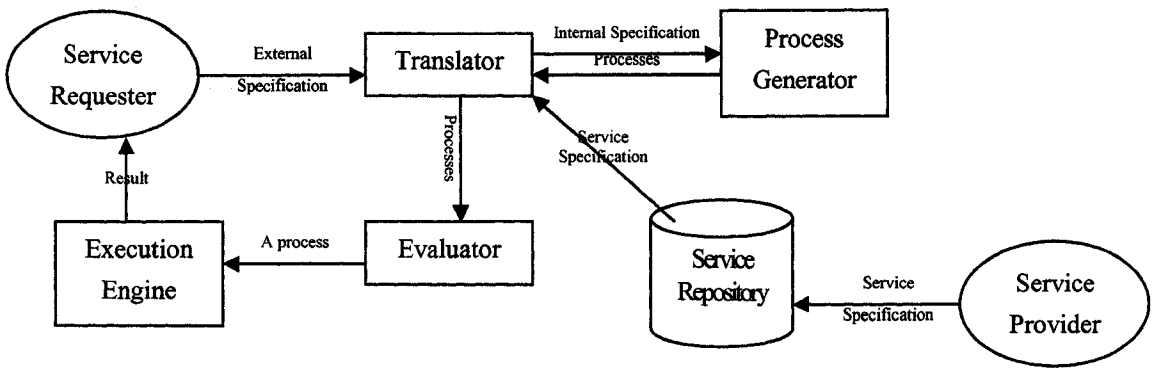


Figure 1: The framework of the service composition system

The service composition is actually performed inside the process generator. A lot of research has been done on how the *process generator* generates the process, and most of them fall into two major categories: workflow composition and AI planning [43, 39].

Workflow-based composition techniques include static and dynamic workflow generation. The static approach, such as BPEL4WS [26] or EFlow [48], requires the requester provides the tasks list and their data dependency before the composition starts [43] [45]. The dynamic approach requires automatic discovery and composition of atomic services and their business data. There is no fully automated tool to perform workflow composition.

Using AI planning to solve web service composition problem is another major direction. OWL-S [15] is such an approach. Its ServiceProfile has precondition and effect properties to produce the state change of a service. Other web service composition methods based on AI planning include Situation Calculus[50], PDDL[51], Rule-Based planning [49] [43]. However these research efforts are mostly theoretical achievements, and are lack of industry and product support.

Having all these approaches above, we propose another approach to web service composition. The process generator can create the process of component services as a composite service. However there are still lots of work ahead when we come into the service execution part. As a matter of fact, one of the biggest issues in dynamic web

service composition is in the execution of composite service. There are so many detailed descriptions inside the composite service, such as the description of control and data flow, and exception handling. How to dynamically generate the execution codes of the composite service is one important and difficult part in the dynamic web service composition.

To web service consumers, it would be a great benefit if there is an automated way of service process generation and execution. We observed a large portion of web services are migrated from database-based web applications. A novel approach has been proposed to automate dynamic Web Service Synthesis [11]. Researchers abstract the core database schema and queries behind these web services, and propose a query-based approach for automated Web Service Synthesis. The general idea is to use database query to reveal the semantic relation of web services, and so that the implementation part of a composite service can be generated from its specification by synthesizing relevant services. Java programs will be generated as the implementation of synthesis service.

There are two relevant approaches that generate the detailed implementation codes from a service specification. In [47], researchers proposed a mediator-based web service composition approach that utilizes the Inverse Rules algorithm to generate a Datalog program.

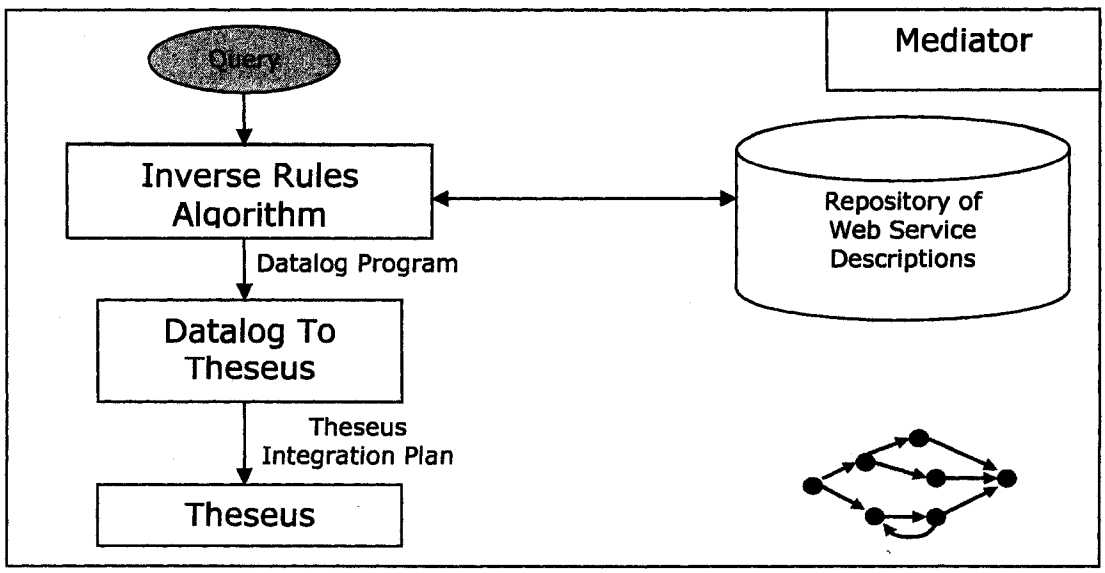


Figure 2: Mediator Based Approach

In [36], researchers advocate a rule-directed approach to dynamically generate and execute composite services. The process schemas are dynamically created via runtime business rules inference; it is a form of service implementation.

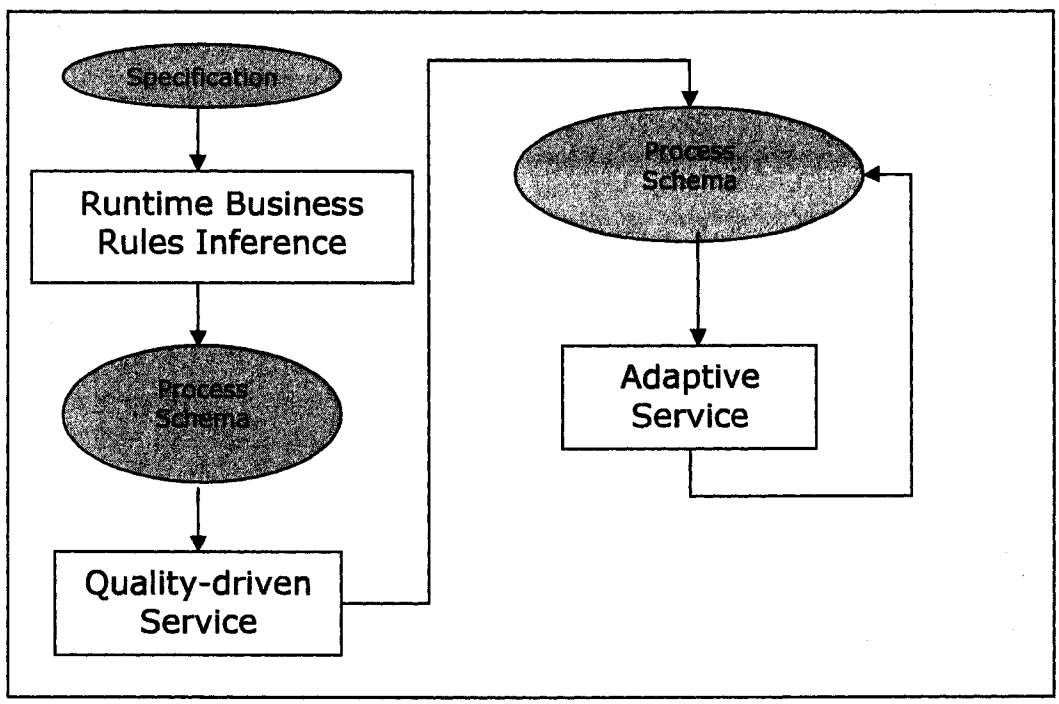


Figure 3: Zeng's Approach

First let's start from an example to understand what Web Service Synthesis does. Suppose that there are two web services "*Amazon*" and "*Shipping*" in service repository. "*Amazon*" service provides a book's Weight, Title, Price, WID (warehouse ID of Amazon) and ZIP1 (the warehouse' zip code) by its ISBN; and "*Shipping*" service provides the goods' standard shipping cost by its Weight and shipping Zip Codes. Each of those services has a semantic description. For example, Amazon service has the following query to describe its semantics, which inquires information from three tables.

```
v_amazon(ISBN, TITLE, WEIGHT, PRICE, WID, ZIP1) :-
        Amazon(ISBN, PRICE, WID),
        BookInfo(ISBN, AUTHOR, TITLE, WEIGHT),
        Warehouse(WID, Addr, ZIP1)
```

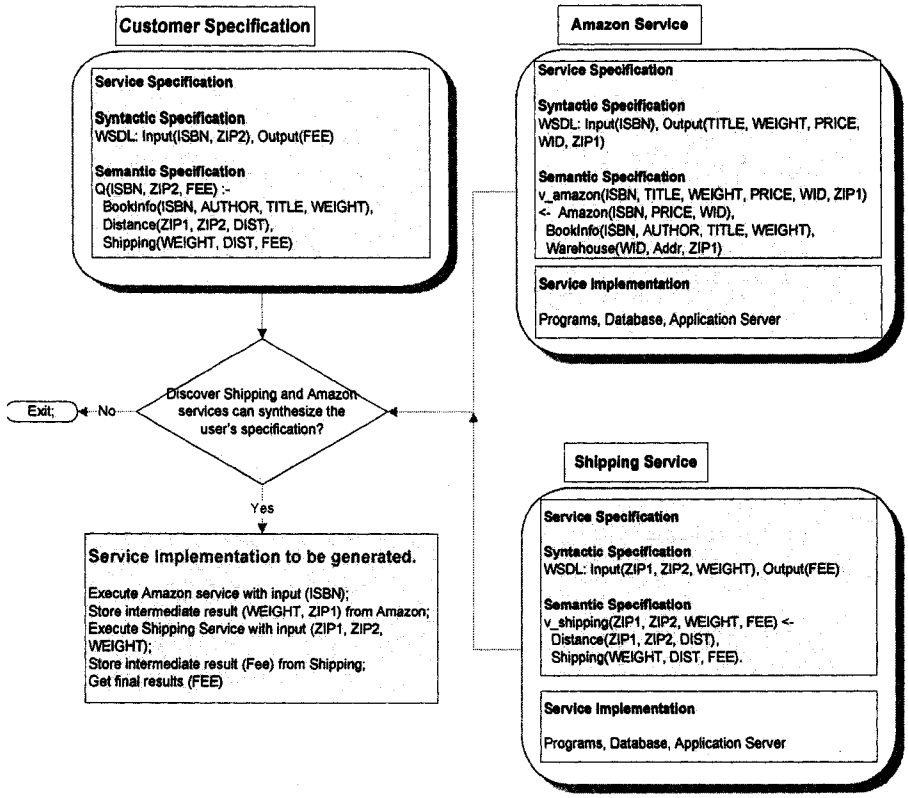


Figure 4: Example and Illustration

Suppose that there is a customer who wants to create a new service that can answer the shipping cost of a book on the internet by giving the book's ISBN and "ship to" zip code as input. The starting shipping zip code is unknown before a specific book vendor is discovered. The query can be:

$Q(ISBN^b, ZIP2^b, FEE^f)$:-
BookInfo(ISBN, AUTHOR, TITLE, WEIGHT),
Distance(ZIP1, ZIP2, DIST),
Shipping(WEIGHT, DIST, FEE).
Where ISBN and ZIP2 are the input condition of the query.

Note that the problem specification does not indicate which bookstore we are using.

First Web Service Synthesizer tries to search a web service that can answer this query directly. If there is no existing single web service that can answer this query directly,

Web Service Synthesizer needs to find out if there are several existing web services that can work together to answer this query. In this example, Web Service Synthesizer's task is to find out "*Amazon*" service and "*Shipping*" service and use these two services to fulfill the customer specification. In doing so, the synthesizer will generate Java programs to invoke "Amazon" and "Shipping" services and gets results for user.

This thesis describes the design and implementation of such a Web Service Synthesis system, the Synthesizer, which synthesizes the implementation of a web service using existing web services from a user *web service specification.* This thesis designs the overall architecture of the Synthesizer, and implements two major components: service planning module and service composition module.

In this work, we assume that web services are annotated with *specifications* that describe semantics as well as the syntax of a web service [11]. The web service's *implementation* part is separated from its *specification* and is to be generated dynamically using existing web services. The Synthesizer can access a repository of available web services which are defined by web service specifications; and to a given specification, the Synthesizer will utilize the web services in the repository to generate its implementation, if there is an answer. The whole service discovery and synthesis process is automated and dynamic.

This thesis is organized as follows: Chapter 2 introduces the web service specification. Chapter 3 is the main part of the thesis, including Web Service Synthesis and its system implementation details. Chapter 4 introduces the experimental settings in this thesis. Chapter 5 draws conclusions and projects the future of semantic-enabled dynamic Web Service Synthesis.

2. Web Service Specification

Currently, most web service description languages, such as the industry standard WSDL [4], define only the syntax of web services. They don't give explicit semantic definitions of web services. On the other hand, other web service semantic specification languages, such as OWL-S[15], which is based on web ontology languages and software agent description language [19], are mostly complex and of research value, and far from industry application. In our research, we proposed to use Datalog [18] notation to describe the semantics of a web service.

2.1 What is Web Service Specification?

Web service specification [11] defines both the semantics and syntax of a web service. The following is the definition of Web Service Specification.

Definition 1 (Web service specification) A web service specification is S(Sig, Q), where Sig is the signature of a web service and Q is the corresponding query of the web service.

A web service specification has a signature to describe web service's syntax and a query to describe web service's semantics. Web service syntax is described by WSDL, which includes the message types, operations and their input/output messages. The query in the service specification is based on a pre-defined Global Schema, which contains all the relations in an application domain. From the database schema, we abstract a database query for each operation to represent the service's semantics. Having the signature and the query, we defined a complete web service specification.

This semantic description is easy for web service consumers to understand and process, because most users are already familiar with database-based applications. Also for web service providers and composers, existing techniques in database application area can be migrated into web service area, and solve new problems in web service area efficiently.

Since we mentioned query and schema here, some people may perceive some similarity to database-based applications. In fact, a large portion of web services are based on legacy database applications. There are web service construction tools, such as DB2SQL [37], to wrap queries as web services. A web service can be implemented in various ways. For example, it can have a background database to store all business data, Java programs to process business logic, and a web server to process HTTP requests and SOAP messages.

Figure 5 illustrates Web Service Specification.

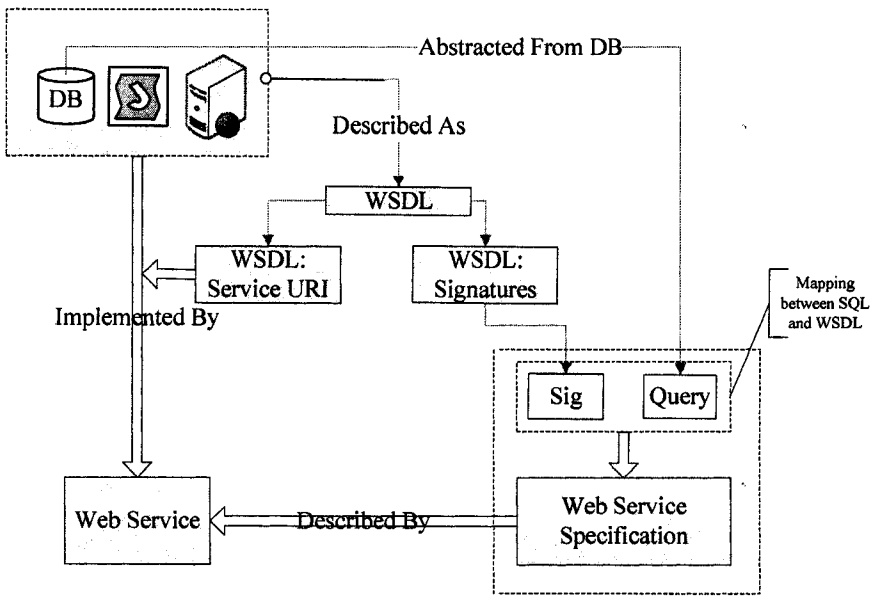


Figure 5: Web Service and Web Service Specification

Let's take a look at an example specification:

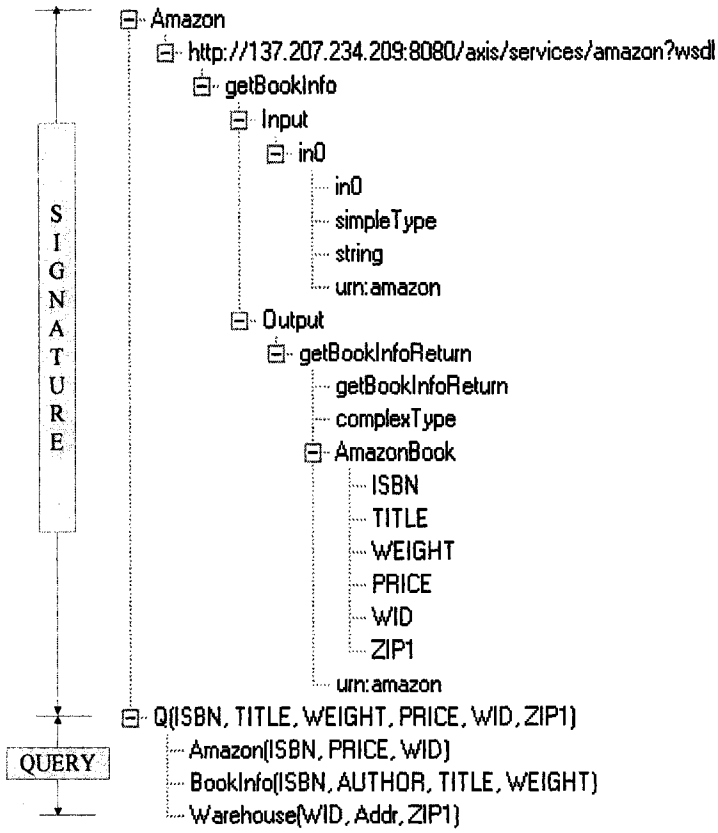


Figure 6: Web Service Specification of *Amazon* service

I use a tree structure to display the hierarchy of specification in the above figure. We see a specification includes the signature and the query of a web service. The signature is defined in XML Schema, just the same as in WSDL. The XML Schema in the signature is only partial WSDL of the service. The complete WSDL file can be downloaded from http://137.207.234.209:8080/axis/services/amazon?wsdl.

This web service specification defines *amazon* web service's *getBookInfo* operation. The input is ISBN, the output is a complex type "AmazonBook" that includes information of *(ISBN, TITLE, WEIGHT, PRICE, WID, ZIP1)*. The query *Q(ISBN, TITLE, WEIGHT, PRICE, WID, ZIP1)* obviously inquires the following information *(ISBN, TITLE, WEIGHT, PRICE, WID, ZIP1)* from three relations: *Amazon(ISBN, PRICE, WID)*, *BookInfo(ISBN, AUTHOR, TITLE, WEIGHT)*, *Warehouse(WID, Addr, ZIP1)*. For the meaning of each variable in *Q*, we have to refer to the relevant relations. For example, "WID" refers to the universal ID of a warehouse. The relation Amazon(ISBN, PRICE,

WID) includes a book's information, including its ISBN, selling price and the warehouse ID that stores this book. The detailed information of the warehouse is defined in Warehouse(WID, Addr, ZIP1), which includes the ID of the warehouse, its detail address and zip code.

This web service inquires by an ISBN for a book's information, such as its title, weight, price, and the storage information like warehouse id and zip code.

We need to understand that the web service descriptions and implementations are separated. In this example the author has implemented the service "*Amazon*" using Java programs. The specification is only a description of the service implementation, so that service consumers and composers can easily understand and invoke this service.

2.2 Global Schema

We assume that there is a relational database schema for all web services. Ideally the global schema contains all necessary relations, so that all business logics and operations can be built on top of the global schema. The global schema consists of relations, which is the same as the tables in database. All base parameter types in web service operations are also defined in the global schema. The global schema is defined in the format of XML Schema similar to WSDL.

The global schema in this thesis defines the following relations in figure 7. There are more details for this global schema defined in XML Schema in Appendix 1.

Amazon

Field Name	Data Type
ISBN	Text
Price	Float
WID	Integer

Distance

Field Name	Data Type
Zip1	Text
Zip2	Text
Dist	Float

BookInfo

Field Name	Data Type
ISBN	Text
Title	Text
Author	Text
Weight	Integer

Warehouse

Field Name	Data Type
WID	Integer
Addr	Text
Zip1	Text

BookReview

Field Name	Data Type
ISBN	Text
Comment	Text
Rate	Integer

Shippping

Field Name	Data Type
Weight	Integer
Dist	Float
Fee	Float

Chapters

Field Name	Data Type
ISBN	Text
Price	Float
WID	Integer

Figure 7: Global Schema

This global schema defines 7 relations, which equal to the database schema behind all web services' implementation. To simplify the real application and make the example easy to understand, the above 7 relations are abstracted and contain only the core information.

For example, any book has certain fixed information, such as its ISBN, title, author and weight, and BookInfo relation defines such information. For a online shopping website, such as "Amazon", it has a relation "Amazon" to record the book's current selling price and the corresponding storage information WID – warehouse ID. The warehouse information is recorded in a separate relation Warehouse(WID, Addr, Zip1) to include the detailed information like address and zip code. Such a global schema defines the necessary relations needed for the book shopping and shipping scenario.

In this thesis, the author manually builds the global schema according to the application scenario. We do not discuss how to build the global schema in this thesis, because the

main goal is to perform Web Service Synthesis. In real world, not every web service is built on database. We may not abstract such a schema in some cases. The focus of Web Service Synthesis approach is on database-based web services. In real-life applications, we can always define the fundamental relations in a specific application domain and compose such a global schema.

2.3 Signatures in the specification

The signature declares the syntactic part of a web service. The signature includes the input and output types of a web service. More specifically, the input or output in a signature includes the type of message (simple or complex), the name space, and all elements and their type of the message. If there is a complex type, the cascade of the data type will be divided and all elements will be listed in the specification.

For example, Figure 6 on Page 10 shows an operation *getBookInfo*, which accepts input message *in0* that has a simple XML Schema type, and returns output message that has complex type *AmazonBook* in XML Schema. The complex type has four elements:

AmazonBook {
    xsd:string ISBN
    xsd:double PRICE
    xsd:string TITLE
    xsd:int WEIGHT
    xsd:int WID
    xsd:string ZIP1
}

Compared to WSDL, the signature is defined in the same way as WSDL using XML Schema. The signature includes types and message definition in WSDL, but doesn't have implementation-related information in WSDL, such as binding/ports, etc. The signature is not concerned about the implementation part. The signature focuses on the information of a web service regarding web service synthesis.

2.4 Semantics of web service

Based on the global schema, WSS uses a query to describe a web service's semantics. The query is defined in Datalog clause in WSS.

2.4.1 Datalog

Datalog is a logic-based data model [10, 18]. Datalog adopts first-order logic as a way to represent relations and operations on relations. Datalog is similar to Prolog, but does not allow function symbols in arguments. The underlying mathematical model of data for Datalog is essentially that of the relational model. Datalog is built from atomic formulas. An atomic formula is a predicate symbol with a list of arguments. For example,

$$p(A_1, \ldots, A_n),$$

where p is the predicate symbol, and A1, ..., An are arguments which can be variables or constants. "p" is an atomic relation, and Datalog is built from the atomic relations.

For example, the atomic relation parent(X, Y) denotes that Y is one parent of X, then the relation sibling can be defined as below:

$$\text{sibling}(X, Y) :- \text{parent}(X, Z) \,\&\, \text{parent}(Y, Z) \,\&\, X \neq Y$$

X and Y has the same parent Z, and X is not equal to Y, therefore X and Y has a sibling relationship.

Datalog notation is a common practice to express conjunctive queries. Conjunctive queries are able to express select-project-join queries. A conjunctive query has the form:

$$q(X) :- r_1(X_1), r_2(X_2), \ldots, r_n(X_n)$$

where q, and $r_1, \ldots, r_n$ are predicate names. The atom q(X) is called the head of the query, and refers to the answer relation. The atoms $r_1(X_1), r_2(X_2), \ldots, r_n(X_n)$ are the subgoals in

the body of the query. The predicate names r_1, ..., r_n refer to database relations. The tuples X, X_1, ..., X_n are vectors of parameters, and vector X_i contains variables or constants elements in relation r_i. ($1 \leq i \leq n$)

Queries may also contain subgoals whose predicates are arithmetic comparisons, such as $<$, $\leq$, $\geq$ and $\neq$. In this case, we require that if a variable X appears in a subgoal of a comparison predicate, then X must also appear in an ordinary subgoal. We refer to the subgoals of comparison predicates of a query Q by C(Q). A conjunctive query with arithmetic comparisons has the following notation:

$$q(X) :- r_1(X_1), r_2(X_2), \ldots, r_n(X_n), C_1(Y_1), C_2(Y_2), \ldots, C_n(Y_m)$$

r_i refer to the relations in database, and C_i (i=1, ..., m) are arithmetic comparison predicates that constraints the query on these relations.

We can take a look at an example query that queries the shipping cost by weight and zip codes. The query needs to gather information from two relations: Distance and Shipping. The join condition is "Distance.DIST = Shipping.DIST".

SQL	Select Distance.ZIP1, Distance.ZIP2, Shipping.WEIGHT, Shipping.FEE From Distance, Shipping Where Distance.DIST = Shipping.DIST
Datalog	v_shipping(ZIP1, ZIP2, WEIGHT, FEE) :- Distance(ZIP1, ZIP2, DIST), Shipping(WEIGHT, DIST, FEE).

Table 2-1. Datalog Example

The above query corresponds to the "Shipping" web service.

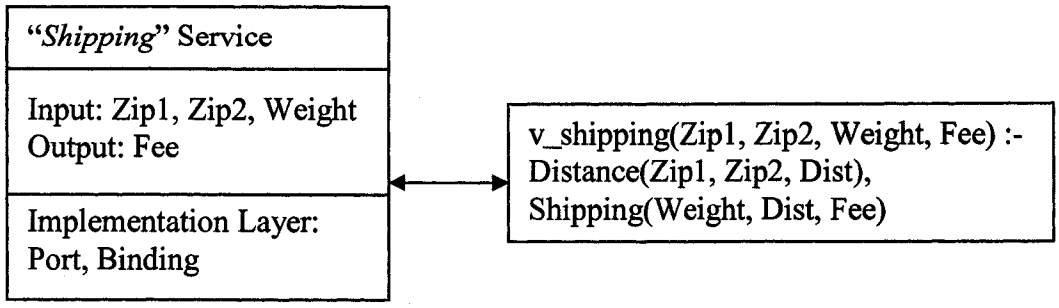


Figure 8: *Shipping* Service and its query in Datalog

2.4.2 Parameterized Query in Datalog

Each web service has certain input and output parameters. However, the query in Datalog cannot represent these constraints of web services. To use the query in web service synthesis, we must represent its access capabilities.

For example, Figure 9 is the semantics of the "*Shipping*" service, which requires three input parameters: Zip1, Zip2 and Weight. Without these inputs, the service cannot execute. However, from the corresponding query v_shipping(ZIP1, ZIP2, WEIGHT, FEE), we cannot find annotations of such constraints. It looks like four parameters (ZIP1, ZIP2, WEIGHT, FEE) are for both input and output. However three input parameters (ZIP1, ZIP2, WEIGHT) must be provided before the query can execute.

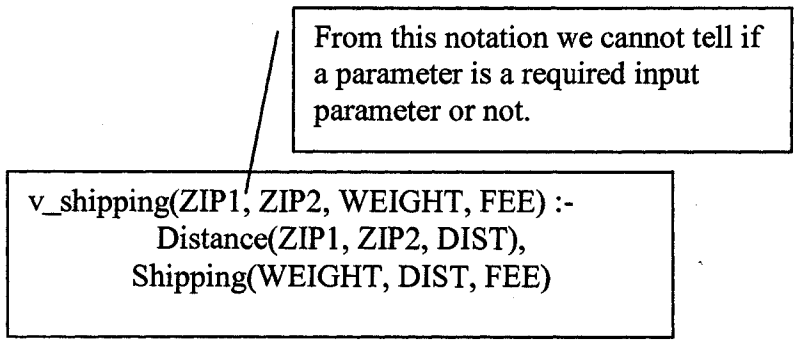


Figure 9: Missing access capabilities on parameters

Therefore we need a mechanism to represent these access constraints. Researchers had discussed this problem in database applications area [30, 31]. To a query, if a parameter

must be satisfied for a query to execute, this parameter is called a "bound" parameter; otherwise it is called "free", and the query can execute without input of this parameter. This happens a lot to diverse information sources with limited query capabilities. Web service is one of these information sources. In a query, we use the "b" to denote bound parameter, and "f" to denote a free parameter [30].

To the query in Figure 9, v_shipping(ZIP1, ZIP2, WEIGHT, FEE), the binding pattern is "b b b f". Parameters (ZIP1, ZIP2, WEIGHT) in the query must be bound when execution.

In our research, we use a superscript "b" or "f" for each parameter in a query to indicate the binding patterns. With this improvement, Datalog notation can fully represent a query and its access constraints.

The same query in Figure 9 can be rewritten as below:

$v\_shipping(ZIP1^b, ZIP2^b, WEIGHT^b, FEE^f)$:-
Distance(ZIP1, ZIP2, DIST), Shipping(WEIGHT, DIST, FEE)

Figure 10: Query with Binding Patterns

2.4.3 The Query in Web Service Specification

The query in web service specification is the database query of the web service that performs data inquiries in background. It is denoted by Datalog and corresponding binding patterns. For the definition of the query, a global schema should always be referred to. The query doesn't make sense without predefined relations and data in the global schema.

For example, "*Amazon*" service has the following query to describe what it does.

v_amazon(ISBN^b, TITLE^f, WEIGHT^f, PRICE^f, WID^f, ZIP1^f) :-
Amazon(ISBN, PRICE, WID),
Warehouse(WID, Addr, ZIP1),
BookInfo(ISBN, AUTHOR, TITLE, WEIGHT).

The query is to obtain the title and weight of a book from BookInfo, its price from Amazon, and its warehouse id and zip code of the warehouse from Amazon and Warehouse tables. Here Amazon, Warehouse and BookInfo are tables in global schema. ISBN is the join condition for Amazon and BookInfo, WID is the join condition for Warehouse and Amazon.

The advantage of using query here is its simplicity. Full fledged web service semantic specification languages such as OWL-S [15] need the top level ontology of service to define semantics. Our "global schema + query" combination is easy for application. In addition, because this method has the same nature as database queries, many database-supporting tools can be used to process web service semantics. For example, *Query Rewriting* technique is used in this thesis to process semantic relation of web services.

2.5 Summary of Web Service Specification

In the real world, one web service may have multiple operations. Each operation has its own functionality, and has its own input and output parameters described by WSDL. Therefore, every operation should have a corresponding query to describe its semantics. We can expand our web service specification definition to contain multiple operations. We can expand the previous definition 1 as below:

Web Service **S** has n operations: op[1], op[2], …, op[n]
To any operation op[i] (1<i<n), there are definitions of Signature[i] and Query[i]
We use:

A vector **Signature** to store Signature [1], …, Signature[n] (1<i<n)

And

A vector **Query** to store Query [1], …, Query[n] (1<i<n).

We have extended web service specification definition:

S(Signature, Query)

Figure 11: Web Service Specification

In the extended definition a signature is a vector of signatures, which defines the inputs and outputs of all web service operations. In addition, query "*Q*" is a vector of queries, which defines the underlying database views of all web service operations. *Signature* vector and *Q* vector have one-on-one relationship.

A web service specification defines what the service is. The implementation part is separated, and it could be done through any available approach. A service specification helps both service provider and service consumer to understand each other. The service provider implements web service, and publishes this service by giving a specification. The service consumer and integrator only needs to know the service specification so that they can understand this service completely and even synthesize it into new service. In this thesis, the specification helps us to define a user request, and leave the implementation details unanswered. Our goal is to use existing services to synthesize the implementation part as user requested.

3. Web Service Synthesis

In this chapter, I will introduce Web Service Synthesis and my design and implementation of such a Web Service Synthesis system.

3.1 Web Service Synthesis and the System Architecture

3.1.1 Web Service Synthesis

Web Service Synthesis (hereafter WSS) is the process to generate a web service implementation from its web service specification, which is carried out by Web Service Synthesizer in a dynamic way [11].

More specifically, the input of the synthesizer is a web service specification *S* to be implemented, and a set of implemented web service denoted by specifications **{S1, S2, …, Sn}**. The output of the synthesizer is an implementation of **S** using one or more implemented services from the service repository **{S1, …, Sn}**.

Let's see an example. Suppose there are two web services "*Amazon*" and "*Shipping*", and their specifications are abstracted as below:

```
Amazon: {
[ Input=(ISBN); Output=(Title, Weight, Price, Wid, Zip1) ],
[ v_amazon(ISBN^b, Title^f, Weight^f, Price^f, Wid^f, Zip1^f):-
        Amazon(ISBN, Price),
        Warehouse(WID, Addr, Zip1),
        BookInfo(ISBN, Author, Title, Weight) ]   }
```

```
Shipping: {
[ Input=(Weight, Zip1, Zip2); Output=(Fee)],
[ v_shipping(Weight^b, Zip1^b, Zip2^b, Fee^f) :-
        Distance(Zip1, Zip2, Dist),
        Shipping(Weight, Dist, Fee) ]   }
```

Suppose a user submit such a specification:

```
BookShippingFee: {
[ Input=(ISBN, Zip2); Output=(ISBN, Zip2, Fee) ],
[ Q(ISBN^b, Zip2^b, Fee^f) :-
    BookInfo(ISBN, Author, Title, Weight),
    Distance(Zip1, Zip2, Dist),
    Shipping(Weight, Dist, Fee) ]   }
```

The user needs to query the shipping cost of a book regardless the book vendor. Shipping from different book vendor will have different shipping cost. There is no existing single web service that can answer user's query. The Synthesizer can discover that two services "*Amazon*" and "*Shipping*" together can answer the query. User only wants to know the shipping cost when knowing ISBN, starting and ending shipping zip codes. Other information, such as Title and Price of a book, are not requested by the user, and will be ignored by the Synthesizer. Finally as the implementation of the user specification, the Synthesizer will generate Java programs that synthesize two services *Amazon* and *Shipping* to find a book's shipping cost.

Figure 12 demonstrates this example and its synthesis process. We will use this example to explain the details of the Synthesizer.

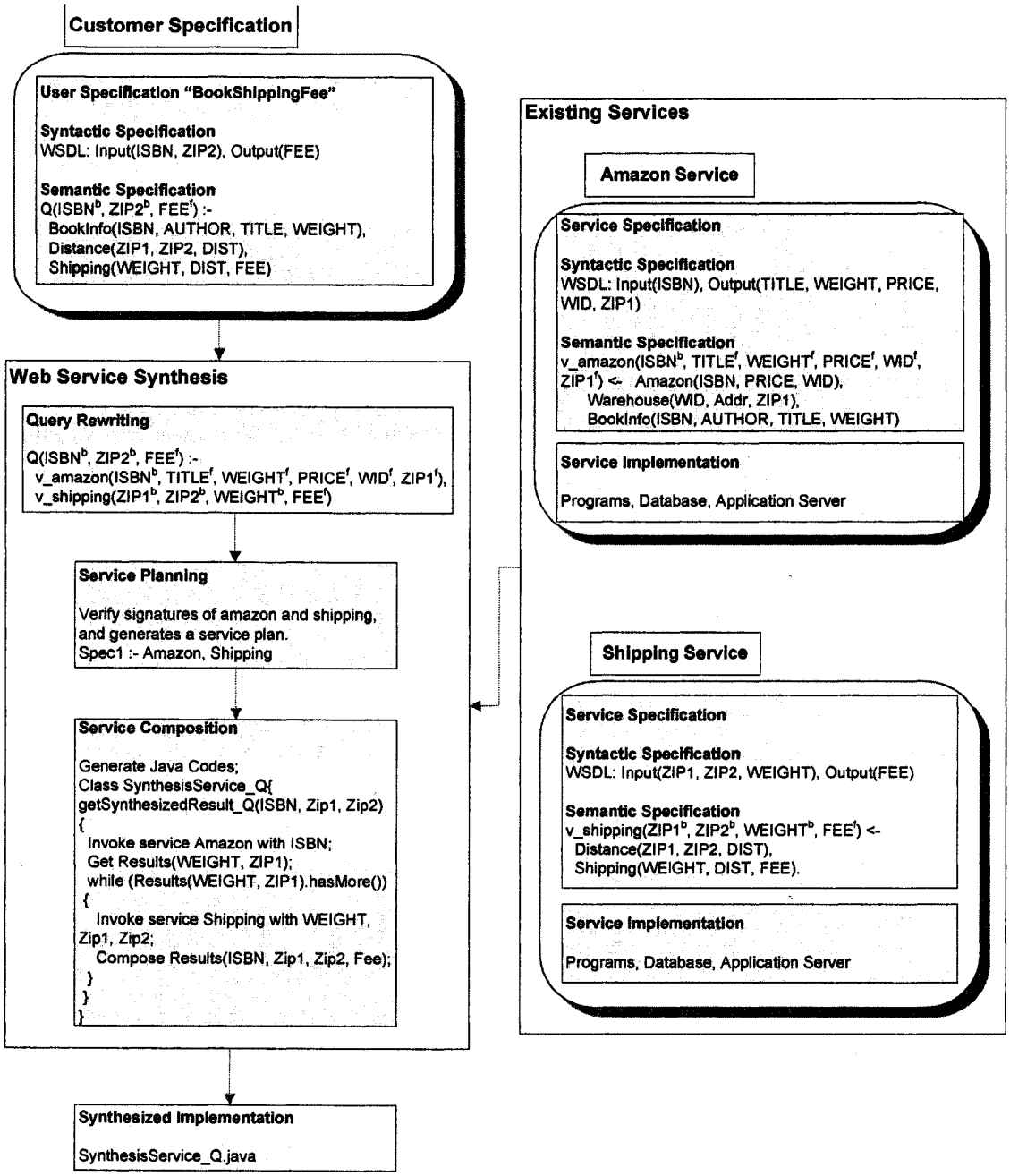


Figure 12: Synthesize a new service from specification "BookShippingFee"

We use *Web Service Specification* (thereafter specification) in the definition of WSS. The specification defines a web service's syntax and semantics, and leaves the service's implementation part unanswered. There is a repository of web services that are already

implemented, and these services are also defined by their corresponding specifications and are available for synthesis in WSS.

When user needs any information from the repository, he can write a service specification to define his requests. The corresponding implementations of this specification are expected to be generated by WSS. The implementation is correct with respect to the specification if its answer set is equivalent to that of the specification for any database instances. The answer set of a query is the set of answers produced in the database. The answer set of the implementation is the set of answers generated by the query Q' in the abstract implementation as defined in below:

Definition 2

(*Abstract implementation*) Given a web service specification $S(Sig, Q)$, and a web service repository consists of n web services: $S_1(Sig_1, Q_1)$, $S_2(Sig_2, Q_2)$, ..., $S_n(Sig_n, Q_n)$. An abstract implementation of $S(Sig, Q)$ using $S_1, S_2, \ldots S_n$ is $S(Sig, Q')$ where $Q' :- Q_i \ldots Q_j$, where $(Q_i \ldots Q_j) \subseteq (Q_1, \ldots, Q_n)$.

When $Q'(Q_i, \ldots, Q_j)$ $(1 <= i, j <= n)$ is a complete rewriting of Q using $(Q_1, \ldots, Q_n)$ (denoted by $Q':- Q_i \ldots Q_j$), and signature "Sig" also matches the combination of $(Sig_i, \ldots, Sig_j)$, we can conclude that web services $(S_i, \ldots, S_j)$ can be synthesized together to implement S(Sig, Q).

In the abstract implementation above, we use the concept "Query Rewriting" in advance. In brief, "Query Rewriting" finds out a set of queries which is equivalent to a specific query Q. We will talk about query rewriting in details in 3.2. Every web service has a query to define its semantics. To any two web services, their semantic relations are actually equivalent to the relation of the queries in their specifications. WSS uses query rewriting techniques to reveal semantic relation of web services. This is the first step in Web Service Synthesis.

From the query rewriting result, the corresponding web services which are semantically relevant can be discovered. Since every service has a signature to define syntax, the signatures in the specification list are used for syntactic verification for these services –

see if they can collaborate with each other syntactically. This is step two – Service Planning - in WSS. After this step, we have found out the a web service S can be synthesized by existing services (S_i, …, S_j) in a certain order.

In step three, the Synthesizer synthesizes the implementation codes that invoke each component service in (S_i, …, S_j) and answer the user query. The implementation codes are Java programs that can be executed independently by user. Figure 13 displays the process flow of WSS.

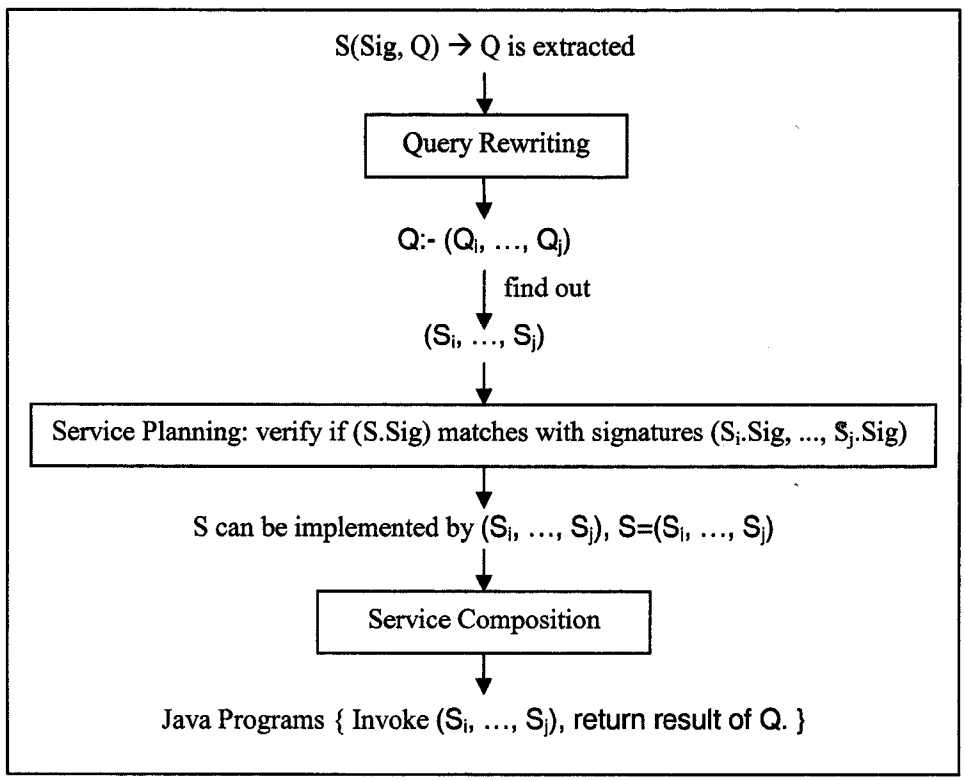


Figure 13: Web Service Synthesis Process Flow

3.1.2 System Architecture

The biggest contribution of this thesis is to implement the theoretical research of WSS into a prototype software system – the Synthesis system. I design the system architecture and implement the whole system, and I also developed two necessary core components: service planning module and service composition module. Because query rewriting

algorithms had been widely proposed and researched, I didn't develop my own query rewriting module; I used the query rewriting package QRW developed by my colleague Minghao Li. Development tools include Java, Axis [35], DOM [38], Javassist [32]. Java codes are finally generated by the Synthesizer. Figure 14 demonstrates the architecture of the Synthesizer.

The Synthesizer takes three inputs: a Global Database Schema, a set of Web Service Specifications, and a user specification for a web service. The Synthesizer will finally generate the user specification's implementation that contains the actual programs to process client requests.

The Synthesizer has three core modules: (1) Query Rewriting module; (2) Service Planning module; (3) Service Composition module.

The query rewriting module finds the equivalent rewriting of user's query. It extracts the queries list (Q_1, Q_2, ..., Q_n) from specification set (S_1, S_2, ..., S_n) of previously defined services. Based on the global schema and the queries list, for any user query Q, it will generate the equivalent rewriting of Q if there is a correct answer; or it will return null if the user query Q doesn't have an equivalent rewriting.

The service planning module takes three inputs: the rewriting result, the global schema and the specification list. From the rewriting result, we can find out relevant services for synthesis. This module verifies if these relevant services can collaborate with each other. Equivalent rewriting only guarantees the semantic correctness, and the service planning module verifies the syntactic correctness. Once succeeds, this module returns a service plan to record the execution order of each component service.

The service composition module takes two inputs: a valid service plan and the specification list. It generates the actual codes that invoke the services in the plan and return the results as user requested.

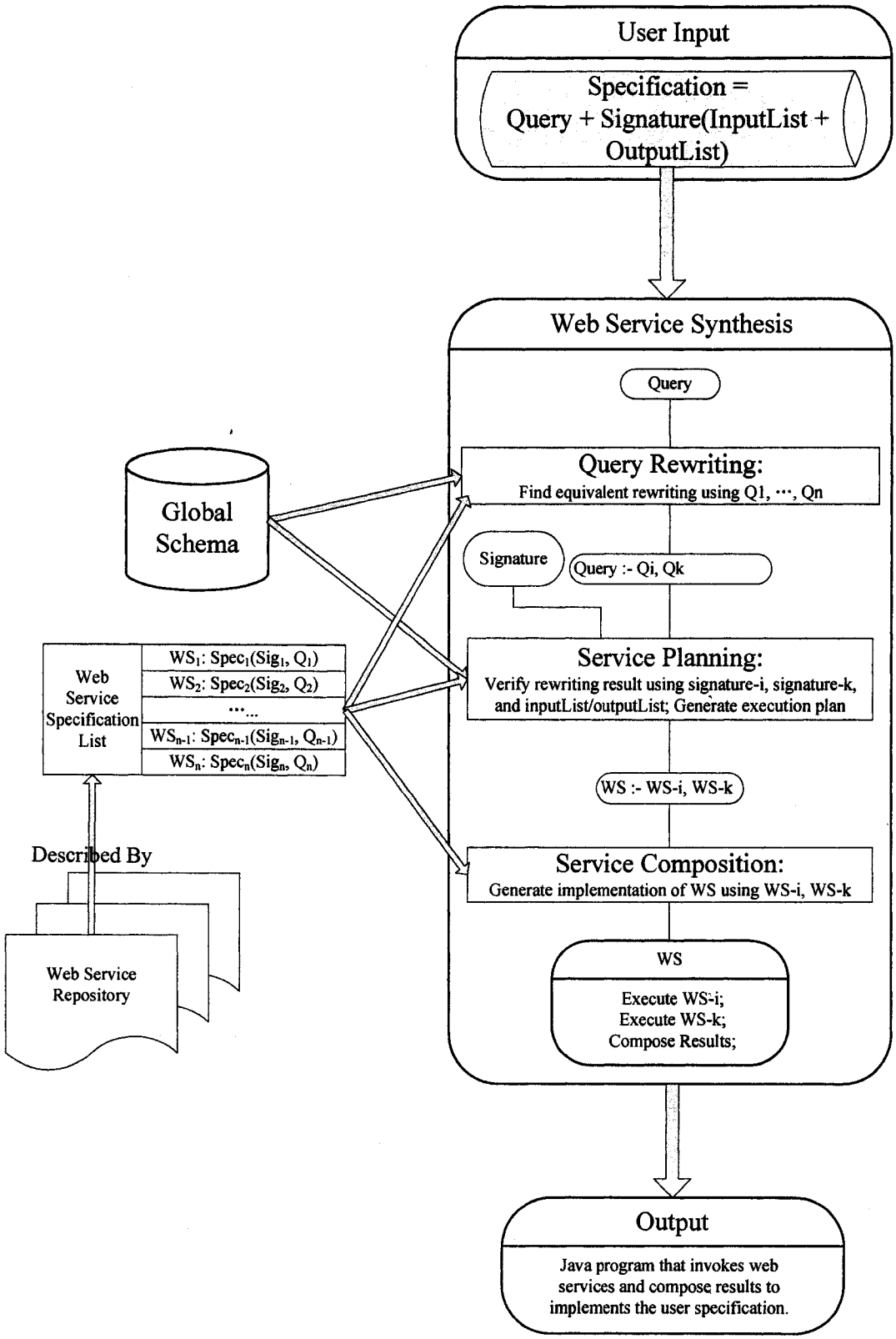


Figure 14: Web Service Synthesis Overall Picture

These three modules are sequential: the output of query rewriting module is the input of service planning; and the output of service planning is the input of service composition. The final outputs are the cascading Java programs.

There are two types of users of the Synthesizer. One is the service consumer who wants to search useful services and integrate them into their applications. The generated Java codes are very practical and can be easily integrated into applications. The other is the service synthesizer who wants to provide new synthesized web services implemented by existing web services. Most parts of the new web service's WSDL can be generated from Java programs. The deployment part is the only part that is not processed by WSS, and has to be done manually by user.

The web services in the repository are just like building blocks, the user query can be synthesized by these building blocks. If user query happens to be equivalent to the query of another single service, WSS will locate the service and return Java codes that invoke this service.

Every time WSS generates a new implementation for a different specification, WSS will add this specification into the specification set, so that this new service can be used for future users. If future user submits any query that had been previously submitted, WSS simply reuses the generated Java codes without synthesis.

WSS generates programs that synthesize component services and dynamically fetch results. When user constantly needs to query the same type of information with different inputs values, user can integrate these programs into their web-based application system and obtain results dynamically.

3.2 Query Rewriting Module

Query rewriting module is responsible for finding out semantically-relevant web services from the repository. From the user specification, the query part is extracted first for the purpose of query rewriting. The query rewriting module needs three inputs parameters: the user query, a global database schema, and the query list that is extracted from the specification list in the repository. In this chapter I will first introduce query rewriting using views in theory level, then I will introduce the query rewriting module in WSS.

3.2.1 Query Rewriting Using Views

The problem of query rewriting is to find the equivalent rewriting of a query Q using previously defined views over the database, rather than accessing the database directly [10]. In web service synthesis research, we can consider a web service as a query over a global database schema, and web service can be regarded as a view over a global database. Therefore query rewriting techniques can be used in web service synthesis to answer user queries using existing web services. Query rewriting is the first step in WSS, which discovers semantically-relevant services.

In this research, we first introduce the concepts: "query containment" and "query equivalence". These two definitions provide the semantic basis for comparison between queries and their rewritings.

Definition 3 Query containment and equivalence: A query $Q1$ is said to be contained in a query $Q2$, denoted by $Q1 \subseteq Q2$, if for all database instances D, the set of tuples computed for $Q1$ is a subset of those computed for $Q2$, i.e., $Q1(D) \subseteq Q2(D)$. The two queries are said to be equivalent if $Q1 \subseteq Q2$ and $Q2 \subseteq Q1$. We denote query equivalence as $Q1 = Q2$.

The definition above indicates that we must compute the tuples of a query Q based on a database instance. In theory, the query containment and equivalence relations must be computed on all database instances. However, in our real world applications, it's

impossible to test on all database instances. We can only compute the query containment and equivalence relation based on the database instances we have. And it is still valuable to know the containment and equivalence relation hold on the available database instances.

Having defined query containment and equivalence, we can define equivalent rewritings of a query. In our research, we only query data on a specific database schema.

Definition 4 Equivalent rewritings: On a database D, let Q be a query and $V = \{V1, ..., Vm\}$ be a set of view definitions. The query Q' is a rewriting of Q using V if Q' refers only to the views in V, and Q' is equivalent to Q (denoted $Q(V) = Q'(V)$).

$V_1, \ldots, V_m$ are views defined on a database instance D. To a query Q, an equivalent rewriting QR_1 is computed on a set of view definitions V. Equivalent rewriting QR_1 will inquire the same tuples as Q. If there is a different set of view definitions V' on the same database D, the same query Q may have different equivalent rewriting QR_2. When we talk about equivalent rewritings, we must include a specific database instance and a set of view definitions.

There are many algorithms to compute the equivalent rewritings, and this thesis uses an expanded bucket algorithm which was introduced in [17, 23]. The original bucket algorithm computes the complete rewriting of a query Q, and the expanded algorithm verifies the query equivalence in addition. There are four steps in this algorithm:

(1) The algorithm constructs a bucket for each subgoal of the query Q. A view V_i is added to the bucket if one of the subgoals of V_i matches the owner subgoal (from the query Q).

(2) The algorithm computes the Cartesian products between views in all the buckets. Each combination of views is a candidate rewriting, which is a conjunctive query joining these views together.

(3) The algorithm checks the containment relation on each candidate rewriting RW against the query Q, to make sure $RW \subseteq Q$, and removes the ones that are not contained.

(4) For each contained rewriting, the algorithm conducts containment checking with Q. If the conjunctive rewriting contains Q, it is a complete and equivalent rewriting.

Let's study an example. There are two views: one provides book information, and the other provides shipping information.

```
v_amazon(ISBN^b, TITLE^f, WEIGHT^f, PRICE^f, WID^f, ZIP1^f) :-
    Amazon(ISBN, PRICE, WID),
    Warehouse(WID, Addr, ZIP1),
    BookInfo(ISBN, AUTHOR, TITLE, WEIGHT)
v_shipping(ZIP1^b, ZIP2^b, WEIGHT^b, FEE^f) :-
    Distance(ZIP1, ZIP2, DIST),
    Shipping(WEIGHT, DIST, FEE)
```

Figure 15: Two View Definitions

There is a user query *Q(ISBN, ZIP2, FEE)* that inquires a book's shipping information.

```
Q(ISBN^b, ZIP2^b, FEE^f) :-
    BookInfo(ISBN, AUTHOR, TITLE, WEIGHT),
    Distance(ZIP1, ZIP2, DIST),
    Shipping(WEIGHT, DIST, FEE)
```

Figure 16: A User Query

There are three subgoals in this query, therefore we can construct three buckets as following:

BookInfo(ISBN,AUTHOR, TITLE, WEIGHT)	Distance(ZIP1, ZIP2, DIST)	Shipping(WEIGHT, DIST, FEE)
v_amazon(ISBN, TITLE, WEIGHT, PRICE, WID, ZIP1)	v_shipping(ZIP1, ZIP2, WEIGHT, FEE)	v_shipping(ZIP1, ZIP2, WEIGHT, FEE)

Because *BookInfo(ISBN, AUTHOR, TITLE, WEIGHT)* is a subgoal of *v_amazon(ISBN, TITLE, WEIGHT, PRICE, WID, ZIP)*, we put *v_amazon* in "*BookInfo*" bucket; because *v_shipping(ZIP1, ZIP2, WEIGHT, FEE)* has two subgoals *Distance(ZIP1, ZIP2, DIST)*

and *Shipping(WEIGHT, DIST, FEE)*, we put *v_shipping* in the bucket of "*Distance*" and "*Shipping*".

By computing the Cartesian products of the views in all buckets, a candidate rewriting *Q'* is found, and denoted in conjunctive query as below:

```
Q'(ISBN^b, ZIP2^b, FEE^f) :-
        v_amazon(ISBN^b, TITLE^f, WEIGHT^f, PRICE^f, WID^f, ZIP1^f),
        v_shipping(ZIP1^b, ZIP2^b, WEIGHT^b, FEE^f)
```

Figure 17: Equivalent Rewriting of Q

By containment checking between *Q* and *Q'*, we can decide if it is an equivalent rewriting of *Q* using views *v_amazon* and *v_shipping*. From this result, there is extra tables in Q', so we know Q' is a contained rewriting. The user query *Q* contains more information than the JOIN of these two views "*v_amazon*" and "*v_shipping*". However this result is still usable for web service synthesis.

In this thesis, views "*v_amazon*" and "*v_shipping*" each corresponds to a web service. In other words, these two web services can compute and answer query Q.

Query Rewriting has many applications, such query optimization, data warehousing, content distribution networks, etc. In this thesis, we apply query rewriting in web service synthesis to reveal the semantic relation of web services.

3.2.2 Query Rewriting System Implementation

The query-rewriting module loads a query, a database schema and a query list, and computes equivalent rewritings of the query using queries in the query list, or it returns null if there is no equivalent rewriting of the query. User inputs the query in a specification; the query list is extracted from the specification list of existing services, and the global schema is predefined.

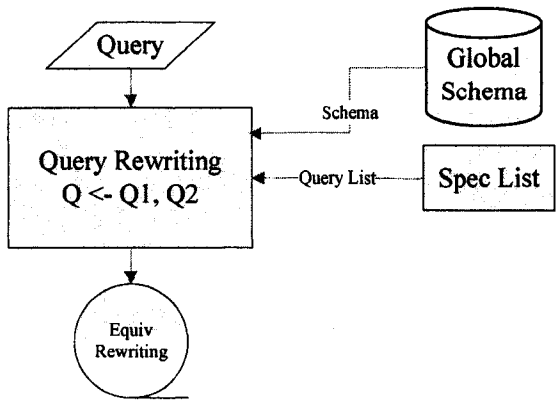


Figure 18: Query Rewriting System

The rewriting results are in Datalog format. An equivalent rewriting has a head predicate, and one or multiple body predicates. For example, to the user query defined in figure 16, the Synthesizer computes the following rewriting result:

Rewriting #1: $Q(ZIP1^b, ZIP2^b, ISBN^b, FEE^f)$:-
$v\_amazon(ISBN^b, TITLE^f, WEIGHT^f, PRICE^f, WID^f, ZIP1^f)$,
$v\_shipping(WEIGHT^b, ZIP1^b, ZIP2^b, FEE^f)$.

Each query in body part has a corresponding web service in the repository. Therefore, two semantic-relevant web services "*Amazon*" and "*Shipping*" are found out by WSS, and their core queries are equivalent to the user's query.

Multiple rewriting results are possible, and any one is equivalent to the user's query, and can answer the user query completely. We can generate multiple solutions of the same user specification from multiple rewriting results.

In WSS, QRS software package is a third party package. The query-rewriting module is developed by Minghao Li in his master thesis. Minghao implements and optimizes the core query rewriting algorithms Bucket [17]. The implementation details of the query-rewriting module are not the focus of this thesis. Any external query-rewriting modules can be plugged into WSS. The Synthesizer only assumes the following interface:

RewritingList = QueryRewriting(Query, GlobalSchema, ViewList)

I integrate the QRS software package into WSS system, and design an interface to present the query, the global schema, the query list and rewriting results.

3.3 Service Planning Module

We can find out semantically-relevant web services from the rewriting results that could possibly answer user's query. However equivalent rewriting doesn't guarantee these web services can work with each other to compute the final answer. Due to the syntactic constraint, WSS must verify the signature of each web service. Even if signatures are verified successfully, these web services still have to be invoked in a certain order, not in random order. This brings the service planning problem.

For example, in Rewriting #1, query *v_shipping* in the body cannot be executed first. There are three bound parameters in *v_shipping*: Weight, Zip1 and Zip2. These parameters must have input values when *v_shipping* is executed. Original inputs from head predicate include ISBN and Zip2, and Zip1 and Weight are not direct inputs from user, therefore this query *v_shipping* cannot be executed first.

Service planning validates the rewriting results and finds out the correct execution order, and finally generates a service plan from the rewriting result. There are bound parameters in the body predicates of the rewriting result, and binding patterns are extracted for planning. These binding patterns reveal the dependencies of body predicates in rewriting result, and these dependencies are irrelevant to the semantics of a web service. The service planning problem essentially is the same as query planning problem. The execution order of queries is the same as service execution order.

Firstly we introduce query planning in 3.3.1. Then we introduce the algorithm and implementation of Service Planning Module in 3.3.2. The service planning module has two steps: (1) Extract binding patterns from service signature; (2) Use "Query Planning" to generate service plan.

3.3.1 Query Planning

We have introduced "Parameterized Query in Datalog" in 2.4.2. We can use binding patterns to denote the access capabilities of parameters in a query. Using b and f to describe binding patterns, the queries in the body of Rewriting #1 are denoted as:

1. v_amazon(ISBN^b, TITLE^f, WEIGHT^f, PRICE^f, WID^f, ZIP1^f)
2. v_shipping(WEIGHT^b, ZIP1^b, ZIP2^b, FEE^f)

In the first query *v_amazon*, ISBN is bound; in the second query *v_shipping*, WEIGHT, ZIP1 and ZIP2 are bound. The user's request Q(ISBN^b, ZIP2^b, FEE^f) has 2 input parameters: ZIP2 and ISBN. Therefore, v_amazon must be executed first, and then v_shipping will be executed, because v_amazon will return (WEIGHT, ZIP1) as outputs, which are the inputs of query v_shipping.

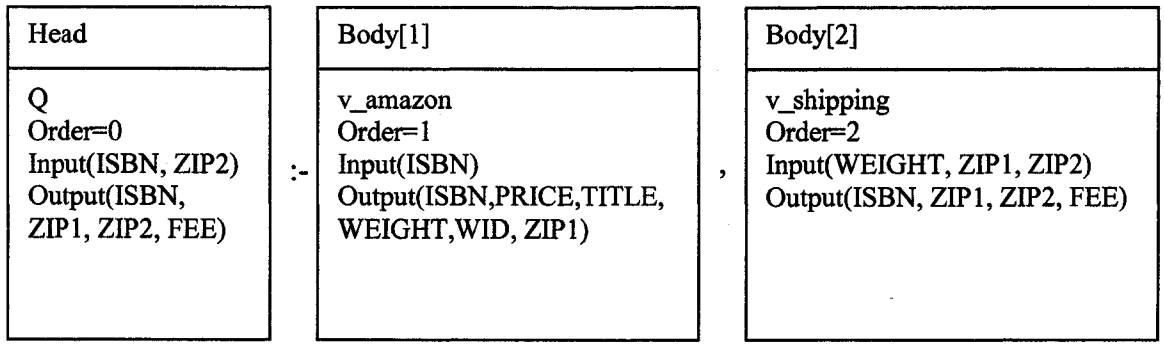
In theory, the body of rewriting result includes conjunctive queries (CQ), and the subgoals must be executed in a certain order. We call the valid execution order as "the feasible order of subgoals", which is defined as below:

Definition 5 (feasible order of subgoals) Some subgoals $g_1(X_1)$, ...,$g_k(X_k)$ in a conjunctive query, CQ, form a feasible order if for each subgoal $g_i(X_i)$ in the order, given the variables that are bound by the previous subgoals, subgoal $g_i(X_i)$ is answerable; that is , there is a binding pattern p_{ij} of the relation g_i, such that for each argument X in subgoal $g_i(X_i)$ that is adorned as b in p_{ij}, whether X is a constant , or X appears in a previous subgoal. A CQ is feasible if it has a feasible order of *all* its subgoals.

The binding patterns are extracted from the signature part of the specification. They denote the constraints of the service signature to the web service query. The feasible order of subgoals is the same as a valid execution order of component web services in the synthesis service. We only give definitions here, and how to find the feasible order of subgoals will be discussed in the next chapter.

3.3.2 Service Planning Algorithm

Let us start from the example. In our example "Rewriting #1", there are two body predicates. It is straightforward to find its feasible execution order. "*Amazon*" can be executed first because bound parameter "ISBN" is in the input list of head predicate. "*Shipping*" must be executed after "*Amazon*" because it waits for bound parameter WEIGHT from output of "Amazon" service. The generated execution plan is as below:



Plan #1: Q:- v_amazon(order=1), v_shipping(order=2)

This example only has two predicates in the body of the query; therefore the planning process is very simply. But this example exposes two principles for service planning:

- Subgoals whose inputs are subset of Head's inputs must be executed first. In this example, Amazon's input "ISBN" is subset of Head's inputs, therefore Amazon is planed first.
- Subgoals whose inputs can be extracted from previous subgoals can be executed after those executed subgoals.

There are more details in [30, 31] regarding this problem. In this thesis, we apply the similar techniques to solve the web service planning problem. WSS will sort body predicates in order and generate a service execution plan using the following algorithm:

```
Service Planning Algorithm:
    Input = bound variables in head predicate
    Plan = null
    BoundPred = all bound predicates
    If exists free predicate then {
        Add free predicate to Plan;
        Input = Input + (input and output variables in this free predicate);
    }
    Changed = true;
    While ((BoundPred != null) && (changed == true)) {
        Changed = false;
        For (i=1, ..., number of BoundPred) {
            If ((BoundPred[i].InputList) in (Input)) {
                Plan = Plan + BoundPred[i];
                Input = Input + (input and output of BoundPred[i]);
                Remove BoundPred[i] from BoundPred;
                Changed = true;
            }
        }
    }
    If (BindingPred == null) then
        return Plan;
    else
        return (PlanNotExist);
```

Figure 19: Service Planning Algorithm

The algorithm starts from free predicates. The inputs of free predicates are not confined by any constraints, and they are executed first. After free predicates are planned, the bound predicates will be planned. A bound predicate can be planned only if its inputs can be extracted from previously planned predicates. The algorithm has two stop conditions: (1) All predicates are planned successfully; or (2) There are at least one bound predicate that cannot be planned any more.

Condition (1) stops when a valid service plan is generated, and condition (2) stops when the rewriting result cannot be planned.

In summary, the service planning algorithm above performs two tasks: (1) check the feasibility of the rewriting result; (2) generates a service plan to record the component service execution order and the location of input parameter (or bound parameter) from previously executed predicate's output list when the rewriting result is feasible.

3.3.3 The Service Plan

The feasible service plan is similar to the conjunctive queries in the rewriting body, but in sorted order. Below is the service plan in general:

Head(QName, Order, InputList, OutoutList) :-
Body[i](QName[i], Order = i, InputList[i], OutputList[i]) (1<= i <= n, n is the number of nodes in the execution plan)

The service plan contains each query's name, its execution order, and its input and output parameter list. The head predicate has "Order=0". When the execution plan is generated, the Synthesizer knows which component services to be composed together and they should be executed in the same order that each predicate is added into the execution plan.

Current implementation of service planning module generates a linear service plan. All nodes are planned in sequential order, and the component services will be invoked in the same order. But a more complex planning is possible, which generates a graph as a service plan. Due to current Synthesizer only compiles linear service plan into corresponding service invocation programs, a graph service plan cannot be processed in WSS anyway. I didn't implement this complex case of service planning. I will leave this as further enhancements of Synthesizer after the prototype Synthesizer is finished.

WSS generates the service plan as an abstract implementation for user. This implementation is not concrete in the sense that the scaffolding codes to integrate the outputs from different services are not generated. We still need to invoke the services in the service plan in a concrete implementation to dynamically return the results to user.

3.4 Service Composition Module - Concrete Implementation

Service composition module generates Java programs as the concrete implementation, which performs the following tasks: (1) invoke the services in the generated service plan; (2) get the final results as user requested.

The input of the service composition module is a feasible execution plan, and the corresponding outputs are Java programs. This module must load the specification list and the global schema to be able to execute. Figure 20 is the process flow of the service composition module.

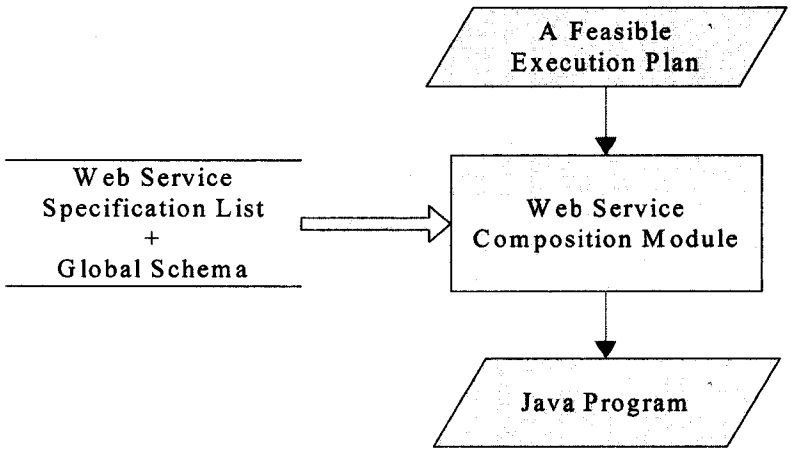


Figure 20: Web Service Composition Module

In our example, given Plan #1 (in 3.3.2) as the input, WSS will generate a Java program that first invokes "*Amazon*" service with ISBN as input, and gets a list of (ISBN, TITLE, WEIGHT, PRICE, WID, ZIP1) as output. It then executes "*Shipping*" service with input (ZIP1, ZIP2, WEIGHT) obtained from Amazon service, and get output (ZIP1, ZIP2, WEIGHT, FEE). Next the program will select (ISBN, ZIP2, FEE) as the output of the composite service.

From the above example, we see the major tasks for web service composition. (1) We need to execute all the services in the service plan on the overall level; (2) We need to invoke each individual service in the micro level; (3) We need to extract the results from each executed service and join the final results requested by user.

These three tasks cover different aspects of service composition. Task 1 takes care of the overall execution, which is called "Service Composition" in this thesis; task 2 takes care of the service invocation of individual service; and task 3 takes care of the results, especially when multiple results are returned in some component services, and the final results are the JOIN of all returned results from component services. Task 2 and task 3 are necessary parts of task 1.

In 3.4.1, I will overview the core Java class to be generated for service composition. Then in 3.4.2 I will talk about the service composition in details.

3.4.1 Overview

There is a core Java program to be generated for service composition, and there are several accessorial programs necessary for service composition. The generated core Java program is connected with the user directly. This program takes direct inputs from user and return final results to the user. Intuitively this Java program must conform to the user specification. In other words, the Java program must completely answer user's query, and follow the same syntactic constraints defined in the specification. Figure 21 displays the high level abstraction of this core Java program.

This class is named by "SynthesisService" plus the query name. In the above example, the class name is "SynthesisService_Q". This class has a "get" method which implements the operation of the web service. This method is named by "getSynthesizedResult" plus the query name. In the above example, the method name is "getSynthesizedResult_Q".

In figure 21 we can see how the user specification maps to a method of the core class. It is natural to have such a mapping that the input list in user specification maps to the input arguments of the method *getSynthesizedResult_Q( )*, and the output list maps to the output data type of this method, and the query is answered by the implementation codes of this method.

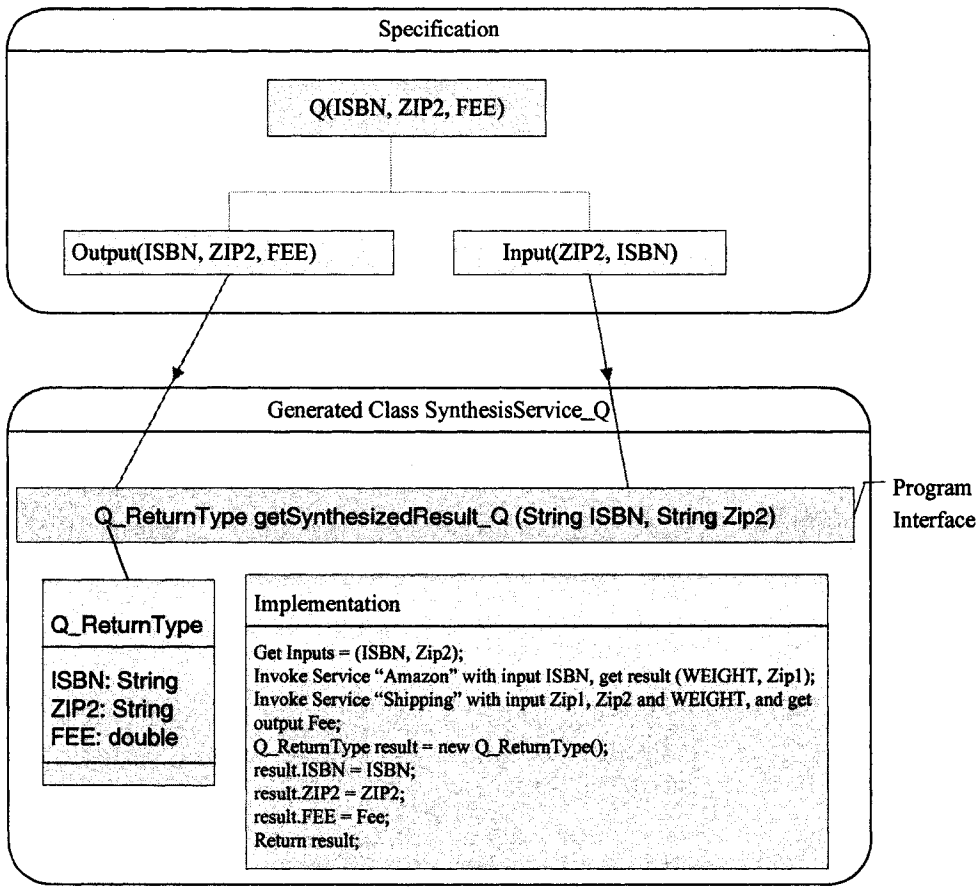


Figure 21: Generated Java Core Class

During the code generation process, we first generate a template class without any implementation contents. Then the detailed composition codes and accessorial codes are generated and inserted into corresponding location. These implementation details will be discussed in 3.4.2.

Figure 22 illustrates such a Java class template and one method that are generated in this step. We can always generate such a class if we have a user specification.

```
public class SynthesisService_Q {
    Vector getResult_SynthesisService_Q(String zip1, Stirng zip2, String isbn) {
        //Insert Method Implementation Here    }
}
```

Figure 22: Class Template for Synthesis Service

The following codes generate this frame class:

```
public void GenerateServiceClassByServiceHead(Head head) {

    String classname = "SynthesisService_" + head.QueryName;
    try {
        File file = new File(classname+".java");
        FileWriter writer = new FileWriter(file);

        String class_definition = "public class " + classname + "{\n\n";
        String method_definition = "public Vector getResult_" + classname + "(";
        Enumeration input_list = head.Input.elements();
        while(input_list.hasMoreElements()) {
            Parameter param = (Parameter)input_list.nextElement();
            method_definition = method_definition + param.getParamType() + " " +
param.getParamName().toLowerCase();
            if (input_list.hasMoreElements())
                method_definition = method_definition + ",";
            else
                method_definition = method_definition + "){";
        }
        writer.write(class_definition + method_definition + "\n\\\\Insert Method
Implementation Here \n}\n\n}");
        writer.close();
    }
    catch (IOException ioe) {
        System.out.println(ioe);
    }
}
```

Figure 23: Codes to Generate the Class Template of Synthesis Service

When output in user specification is complex type, WSS creates a new data type class for this synthesis service. In the example in Figure 21, user requires multiple elements (ISBN, ZIP2, FEE) in outputs, therefore a new data type "Q_ReturnType" (figure 24) is generated, which has members "ISBN", "ZIP2" and "FEE".

```
D:\workdir\workspace\thesis>javap Q_ReturnType
Compiled from "Q_ReturnType.java"
public class Q_ReturnType extends java.lang.Object implements
java.io.Serializable{
    public java.lang.String getISBN();
    public void setISBN(java.lang.String);
    public java.lang.String getZIP2();
    public void setZIP2(java.lang.String);
    public double getFEE();
    public void setFEE(double);
    public Q_ReturnType();
}
```

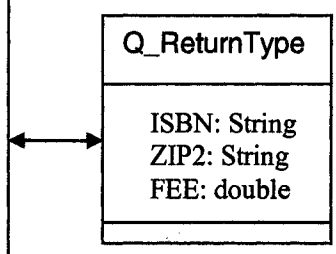


Figure 24: The generated class for Q(ISBN, ZIP2, FEE)

The codes to generate "Q_ReturnType" are as below:

```
public void GenerateOutputDataType(Head head) {
     String classname = head.QueryName + "_ReturnType";
     try {
     File file = new File(classname+".java");
     FileWriter writer = new FileWriter(file);
     writer.write("import java.io.Serializable;\n\n");
     writer.write("public class " + classname + " implements Serializable {\n\n");
     writer.write("public " + classname + "() {};\n\n");

     Enumeration output_list = head.Output.elements();
     while (output_list.hasMoreElements()) {
          Parameter param = (Parameter)output_list.nextElement();
          String member = "private " + param.getParamType() + " " + param.getParamName() +
";\n";
          String getMethod = "public " + param.getParamType() + " get" + param.getParamName()
+ "(){return " + param.getParamName() +";};\n";
          String setMethod = "public void set" + param.getParamName() + "(" +
param.getParamType() + " var) {" + param.getParamName() + " = var;};\n";
          writer.write(member + getMethod + setMethod + "\n");
     }
     writer.write('}');
     writer.close();
     }
     catch (IOException ioe) { System.out.println(ioe); }
}
```

Figure 25: Codes to generate the return type class

Another important issue here is the number of tuples returned by this method. In general, it is possible for this method to return multiple tuples. Therefore, the return type of this method is a Vector instead of Q_ReturnType (in figure 22). This vector stores all Q_ReturnType objects returned by this method.

Having generated the core class template SynthesisService_Q.java and the return data type class Q_ReturnType.java, we are going to generate the codes inside the core method *getSynthesizedResult_Q( )* which performs service composition. Functionally speaking, we need to generate codes to execute the service plan, and compose final results.

3.4.2 Implementations for Service Composition

3.4.2.1 Overall Service Plan Execution

In current version of WSS, the service plan execution is sequential because the service plan is linear. All nodes are planned in order and their executions are also in the same

order. The overall service plan execution only needs to guarantee that the input condition of a component service is satisfied when it is invoked.

For a component service to be executed, its input parameter must be provided. The initial inputs come directly from user, and they are the value of input list in the head node. Since the execution plan is feasible, the input parameters for each component service are either from user's inputs, or from previous service nodes executed before this node. The body nodes are executed in sequential order. It is impossible to have a node service that is not executable due to the lack of inputs condition; it is guaranteed by "service planning".

Component services will be invoked one by one in nested loops in the same order as the order in service plan. The key problem here is how to locate the correct value of a parameter that was returned from previously invoked service. Every pair of parameter name and its value are stored in an output list. When a component service is ready for invocation, WSS always checks its parameters in the input list in the specification, and locates these parameters from the output list which stores all output parameters and their values from the previously-invoked services. The service plan has recorded the input parameter's location in previously-executed predicate's output list during service planning. WSS uses this information to retrieve the correct value of input parameter.

The execution of each component service is nested one by one in the same order as the service plan. When there are multiple results returned, there is a while loop to perform the "JOIN" operation. The final results are saved as Q_ReturnType data type in a vector.

Now we discuss the generation of the implementation codes. Each component service corresponds to a segment of codes. Figure 26 shows the mapping between the service plan and the codes segments that are generated.

Core codes have been generated in this step, and we still need to generate the accessorial codes, such as codes for service invocation and results extraction. We will cover these topics in 3.4.2.2 and 3.4.2.3.

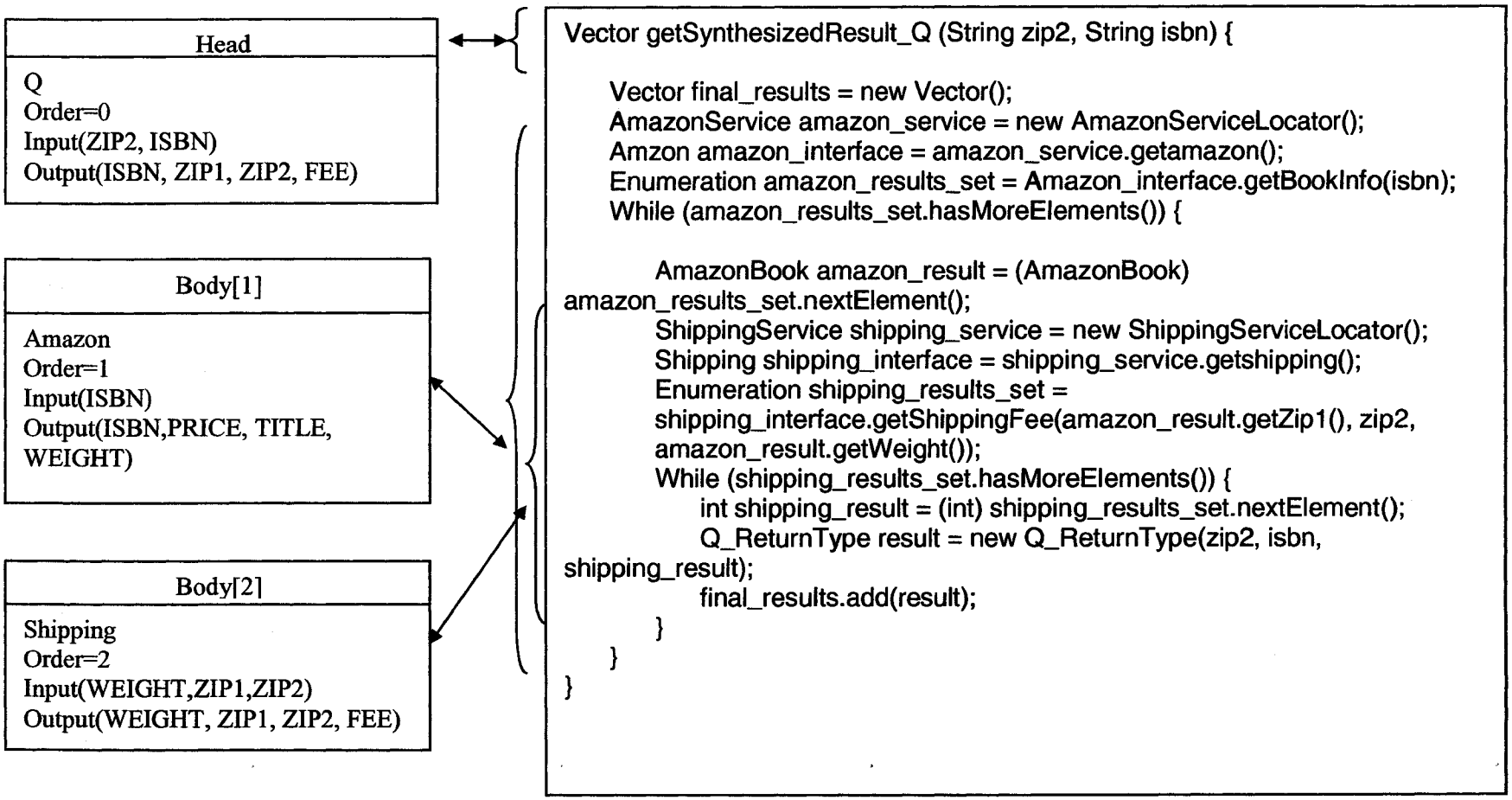


Figure 26: Code Generation Mapping with Service Plan

3.4.2.2 Component Service Invocation

For each node in the service plan, we call it as a *component service*. Each component service is a fully deployed web service that is accessible from the Internet. The component service is described by a specification in WSS, and also published by a WSDL file for service consumer on the Internet.

There are different approaches to invoke a web service. For example, we can communicate directly with the web service through SOAP message. In this approach, WSDL is not required, but the client-side codes are very complex. In mainstream SOAP frameworks, such as Java and .NET, WSDL is required, and the client-side codes are generated automatically from WSDL by supported tools.

WSS adopts the Java implementation of SOAP [6] – Apache Axis [35] – to invoke web service. The invocation of the component service has two steps: (1) WSS generates several Java classes from the component service's WSDL; (2) WSS executes these Java programs with input values to instantiate the component web service and get returned values from the service.

For example, the first node in service plan corresponds to "*Amazon*" service. It has an operation "*getBookInfo*". WSS can easily locate this web service and its corresponding WSDL by its service specification. The complete WSDL of "*Amazon*" service in this thesis can be found in Appendix 2 or at http://137.207.234.209:8080/axis/services/amazon?wsdl. We will start our introduction from this WSDL file. Figure 27 gives the abstracted tree structure of the WSDL:

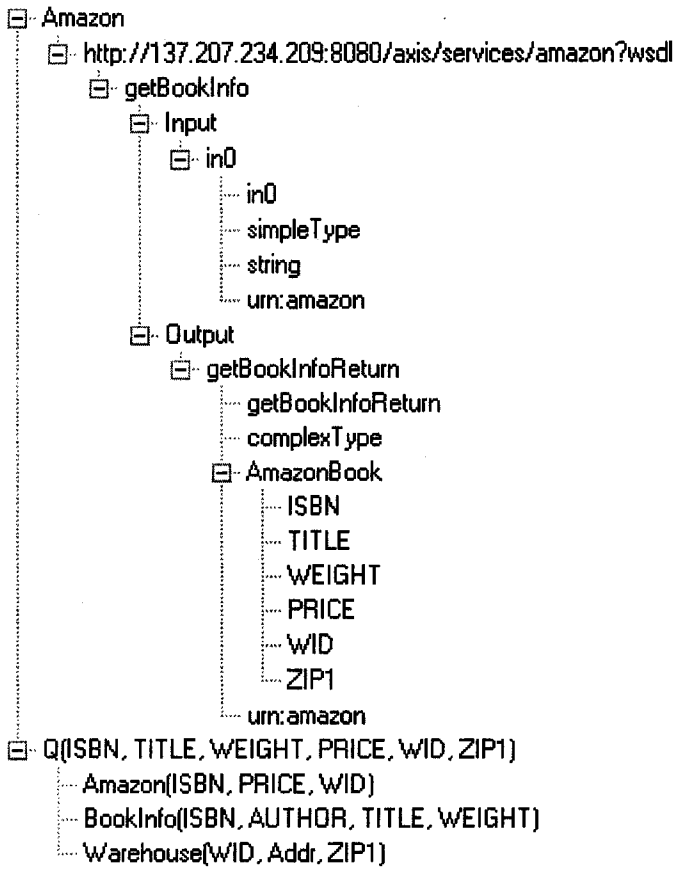


Figure 27: Abstracted Amazon Service Operation

Firstly, to simplify the execution of each component service, WSS uses WSDL2Java in Axis [35] to generate several Java classes. The invocation form is as below:

```
% java org.apache.axis.wsdl.WSDL2Java   the-WSDL-file
```

To "Amazon" web service, the WSDL file can be converted into Java programs by the following command:

```
% java org.apache.axis.wsdl.WSDL2Java
http://137.207.234.209:8080/axis/services/amazon?wsdl
```

By WSDL2Java, WSS generates the following Java classes from the WSDL:

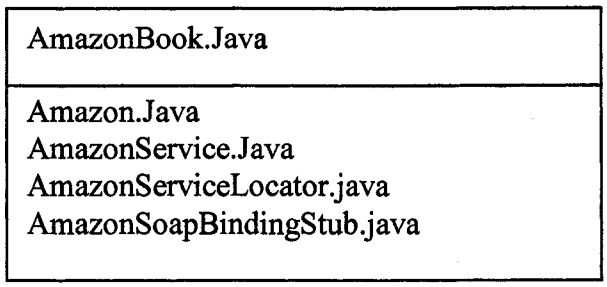


Figure 28: Classes Generated from Amazon's WSDL

AmazonBook.java is a data type class that has the same name as the complex type "*AmazonBook*" in the WSDL. Four classes are generated, and they are a Java interface for portType, a service interface and a service locator for service, and a stub class for soap binding. They are named by the web service name "*Amazon*" plus a fixed name. Amazon.Java corresponds to the portType "*Amazon*" in WSDL, and it defines the interface of the operation "*getBookInfo*" of web service "*Amazon*". For more details on WSDL2Java and these generated Java classes, please see Axis user manual in [35].

To different web service, WSS will generate different classes from its WSDL. For example, to the second node "Shipping", WSDL2Java generates the following Java classes:

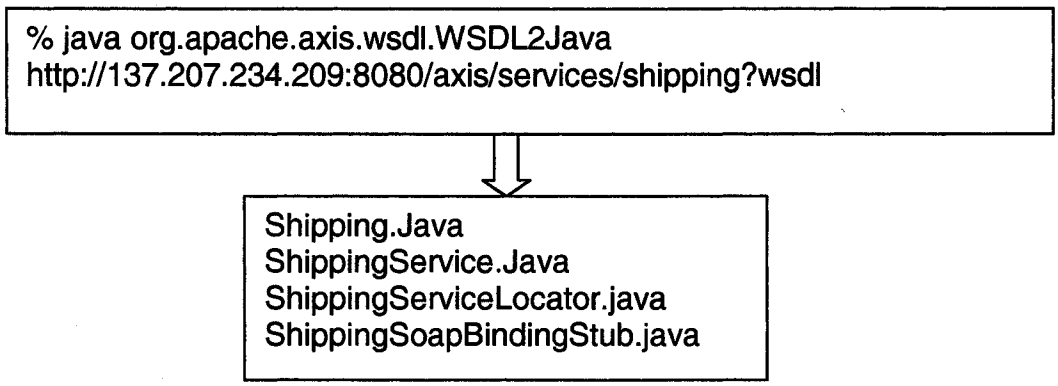


Figure 29: Classes Generated from Shipping's WSDL

Since the operation "*getShippingFee*" of "*Shipping*" service returns a simple type, therefore WSS doesn't create a data type class. All these classes' names start with the service name "*Shipping*".

In step two, WSS will execute these Java programs to invoke services. Amazon.java has a method "getBookInfo" that corresponds to the operation "*getBookInfo*" of service "*Amazon*". With correct input ("ISBN" = isbn) value, these Java programs can be executed as below to invoke the "getBookInfo" method of "Amazon" web service:

```
AmazonService as = new AmazonServiceLocator();
Amazon amaz = as.getamazon();
AmazonBook ab = amaz.getBookInfo(isbn);
```

AmazonService.java is a service interface which is implemented by the locator AmazonServiceLocator.java. The locator obtains the binding information from the stub class AmazonSoapBindingStub.java. Amazon.java is returned by the "get" method "getamazon()" of AmazonService, and is implemented in AmazonSoapBindingStub.java.

When user specifies a "ISBN" value as input, Amazon.getBookInfo(isbn) method instantiates the "*getBookInfo*" operation of web service "*Amazon*" with input "isbn", and returns the results as a serialized object *AmazonBook*.

For example, when isbn = '184628581X', the following object is returned:

AmazonBook
ISBN = "184628581X" TITLE="Semantic Web" PRICE=79.95 WEIGHT=6 WID=1005 ZIP1=98101

It is handy to build web service applications on a SOAP framework such as Axis, and a lot of work has been standardized for ease of use. There is complete information about

Axis SOAP framework in [35]. This thesis will not cover how Axis generates these Java classes. This thesis uses Axis to simplify the invocation of web services.

From the bode nodes, WSS generates the cascading codes for service invocation and result composition. For each body nodes, WSS needs to generate a small segment of codes to such as below:

```
AmazonService amazon_service = new AmazonServiceLocator();
Amzon amazon_interface = amazon_service.getamazon();
Enumeration amazon_results_set = amazon_interface.getBookInfo(isbn);
While (amazon_results_set.hasMoreElements()) {
    AmazonBook amazon_result = amazon_results_set.nextElement();
    //Insert Nested Service Implementation Here
}
```

Figure 30: Codes Segment of Amazon service

This segment of codes have fixed format and naming rules. WSS has a template to save it, which is "code_template" file. Please see the following template codes:

```
<?>Service   <#>_service = new <?>ServiceLocator();
<?>   <#>_interface = <#>_service.get<#>();
Enumeration <#>_results_set = <#>_interface.<method>(<inputs>);
while (<#>_results_set.hasMoreElements()) {
    <returnType> <#>_result = <#>_results_set.nextElement();
    //Insert Nested Service Implementation Here
}
```

Figure 31: Codes Template

<?> can be replaced with "Amazon" in data type names and <#> can be replaced with lower case "amazon" in variable names. Method "getBookInfo" and its return type "AmazonBook" come from the method name and the returned data type name of the service node's specification file. The input of the method <input> can be replaced by isbn which is located in the input list from user specification.

WSS will replace these variables with correct values from the service node. <?> is the service name (same as query name); <#> is the service name in lower case; <method> is

the method name of this service node; <inputs> is the bound parameter of the specification, and its location is recorded in the service plan; <returnType> is the returned data type of this method. The return type can be simple type or complex type. All these information can be dynamically extracted from the specification of this service node.

The generator codes are as below:

```
public void GenerateServiceCodesByServiceNode(String classname, ServicePlanNode
node) {
    String serviceName = node.getQueryName().ToString();
    String serviceNameLCase = serviceName.toLowerCase();
    String method = node.getWSDL().getOperationName();
    String inputlist = node.locateInputs().toParameterList();
    String returntype =
node.getWSDL().getOperation().getReturnMessageType().getName();
    DupTemplate(serviceName+"_template", "code_template");
    KeywordsReplace(serviceName+"_template", "<?>", serviceName);
    KeywordsReplace(serviceName+"_template", "<#>", serviceNameLCase);
    KeywordsReplace(serviceName+"_template", "<method>", method);
    KeywordsReplace(serviceName+"_template", "<inputs>", inputlist);
    KeywordsReplace(serviceName+"_template", "<returnType>", returntype);
    FileHandler file = OpenFile(classname+".java");
    file.InsertMethodImplementation(serviceName+"_template");
}
```

Figure 32: Generation of Implementation Codes

3.4.2.3 Result Extraction and Join operation

The overall service composition is simple when component service returns single result. When component service returns multiple results, "JOIN" operation must be computed. There are "while loop" on each result, and the component services that are nested inside will be invoked multiple times. In the most inner component service, WSS uses the output data type to store the results of each invocation in a collection.

```
Initial Inputs, expected Outputs = Spec.OutputList
Invoke S1 with Inputs → Result(1);
{ Invoke S2 with each row in {Inputs, Result(1) } → Result(2);
{ Invoke S3 with each row in {Inputs, Result(1), Result(2)} → Result(3);
... ...
{Invoke Sn with each row in {Inputs, Result(1), Result(2), Result(3), ...,Result(n-1)} →
Result(n);
Outputs are extracted from {Inputs, Result(1), Result(2), ... , Result(n)};
} //End of Invoke Sn
... ...
} //End of Invoke S3
} //End of Invoke S2
} //End of Invoke S1
```

Figure 33: Result Extraction while Executing Services

The web service in the inner loop will be invoked on each returned results of the current service. These nested loops carry out the "JOIN" operation on outputs of all component services.

In the inner most loop, where the last component service node is reached, the whole service plan is executed completely, and we have all the results from all component services. The output list in the user specification defines all required output parameters, and these results are extracted and returned as one successful result.

3.4.3 Summary of the concrete implementation

In this thesis we use Java as the implementation language. The concrete implementation includes the service invocation codes generated by WSDL2Java, the generated serialized object, and the core class with nested loops to invoke component services. The core class can be executed by user with different input values, and different results are returned from invoking all component services.

Currently WSS doesn't provide an implementation in the form of web service. Actually the web service implementation codes are already generated. And it is not difficult for WSS to generate WSDL by Java2WSDL.

If user needs to publish the Java program as a web service, user needs:

(1) A web server to process http requests, such as Tomcat [34];

(2) A SOAP engine to wrap Java objects, such as Axis [35];

(3) A valid domain

And these are not hard tasks.

4. Experiments

Due to the lack of agreements with large web service providers like "Amazon" or "Fedex", the global schema can only be constructed locally in my experiments. In this thesis, I set up an experimental web services to test Web Service Synthesis.

4.1 Experimental Web Service Repository

I use Apache Axis [35] to build web services in this thesis. Axis must be installed on an application server. I use Jakarta Tomcat 5.5.9 [33] as application server. Axis can serialize/deserialize Java classes which follow the standard JavaBean pattern of get/set accessors. All you need to do is to tell Axis which Java classes map to which XML Schema types. Configuring a bean mapping looks like this:

```
<beanMapping qname="myNS:AmazonBook" xmlns:myNS="urn:Amazon"
          languageSpecificType="java:AmazonBook"/>
```

This mapping tells class "AmazonBook" in Java is mapping to the qname "AmazonBook" in web service "Amazon". Once we finish the WSDD file, we can deploy and publish the web service using a special utility from Axis: org.apache.axis.client.AdminClient. The web service becomes public to any Internet user program. We can even register this web service in UDDI, so that web service consumer can find it easily.

I set up 10 test web services in the web service repository, which can be found at: http://137.207.234.209:8080/axis/servlet/AxisServlet. The following table introduces the functions of these services.

Service	Functions
Shipping	Computes shipping fee by weight and zip codes
PurchaseOrders	Track purchase orders and details
Customers	Lookup customer's detail information by customer ID
Discount	Provides a book's discount information
Authors	Lookup an author's detail information by author name
Exchange	Provides currency exchange information
Vendors	Provides book vendor information by vendor ID
Chapters	Provides book price from a faked vendor "Chapters"
Amazon	Provides book price from a faked vendor "Amazon"

I build these web services using Java and Axis. There are 3 steps:

1. Write Java programs to implement the functions of these web services. The backbone database schema is defined as well.
2. Install these Java programs on Axis
3. Use WSDD to publish these programs as web services

These web services become public after they are published through Axis. Anyone can use these web services freely if he knows the WSDL. Once we have a public domain name, and register these services in UDDI, these services can be searched and consumed by arbitrary client programs on the Internet. In current implementation, our experiments are all local.

4.2 Constructing Web Service Synthesis Environment

The web service synthesis environment includes the Global Schema and Web Service Specification List.

The global schema is an assumption in this thesis. The global schema is constructed manually by the author in the experiments of this thesis. An experimental global schema is defined in Appendix 1.

The web service specification list is constructed when I implemented the Java programs of each web service. When I design the functionality of a web service, the query of this service is extracted as semantics, and the input/output parameter lists are extracted as signature. The web service specification list is based on the global schema. It is manually constructed by the author in this thesis.

The construction of global schema and the specification list is served as the registration process in WSS. Whenever a new web service is registered in WSS, it refers to the global schema and defines its specifications, therefore this new web service is understandable to both service consumers and service integrators.

4.3 Test Web Service Synthesizer

I use several different queries to test web service synthesizer.

I use the query Q(ZIP2, ISBN, FEE) as an example to explain WSS implementations. WSS first uses query rewriting module to find out the equivalent rewriting of Q using views in the view list (Appendix 2).

```
Q( ZIP2, ISBN, FEE) :-
        AmazonSvc(ISBN, TITLE, PRICE, WEIGHT, WID, ZIP1),
        ShippingSvc(ZIP1, ZIP2, WEIGHT, FEE)
```

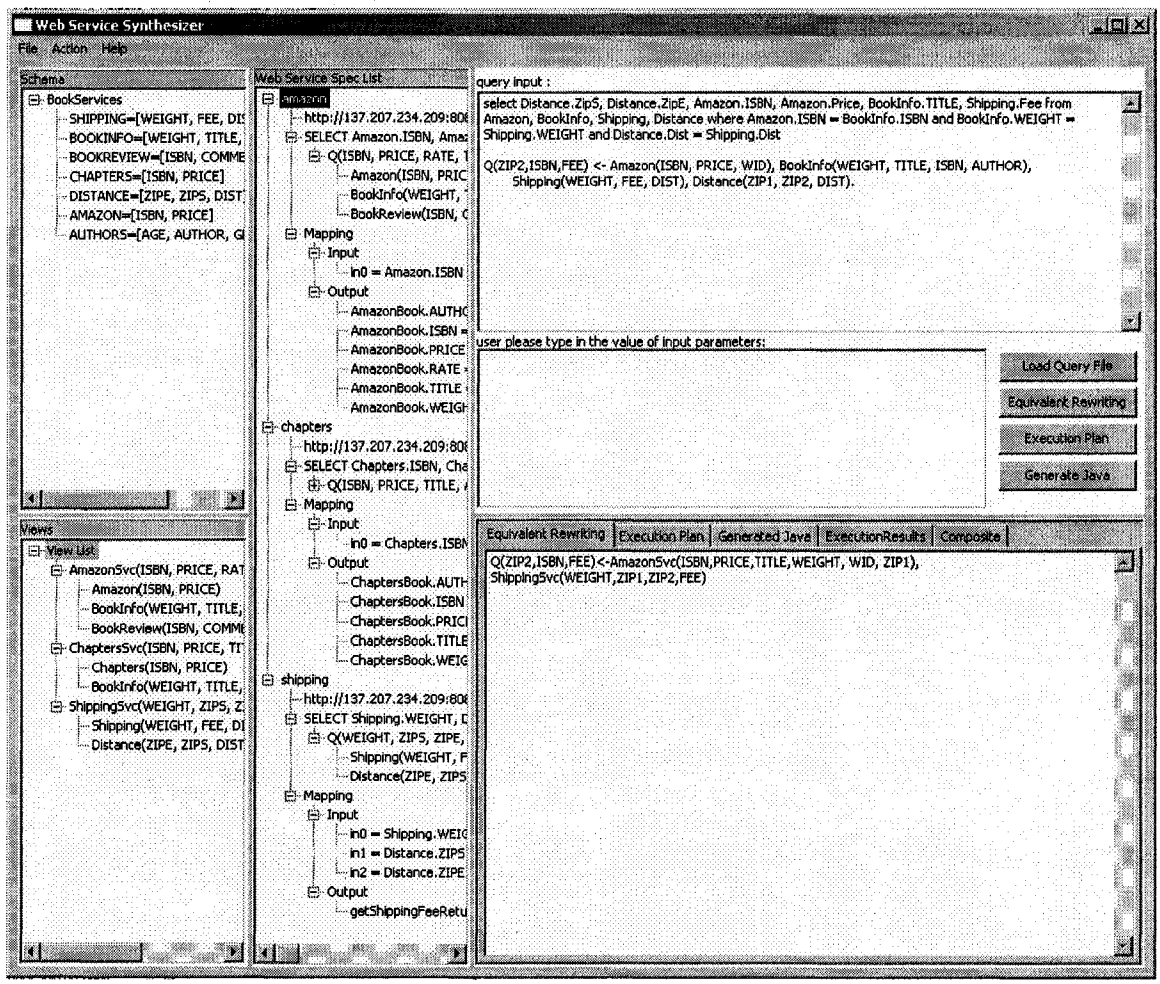


Figure 34: Web Service Synthesis system

The above is the interface of the Web Service Synthesis system. It loads the global schema and specification list as environments, and computes the equivalent rewriting of the user query Q(ISBN, ZIP2, FEE). Then we can generate an execution plan as below:

<Head>
<Query>Q(ISBN, ZIP2, FEE)</Query>
<Input>ISBN, ZIP2</Input>
<Output>ISBN, ZIP2, FEE</Output>
</Head>
<Nodes>
<Node>
<Query>AmazonSvc(ISBN, TITLE, PRICE, WEIGHT, WID, ZIP1)</Query>
<Input>ISBN</Input>
<Output>ISBN, TITLE, PRICE, WEIGHT, WID, ZIP1</Output>
</Node>
<Node>
<Query> ShippingSvc(ZIP1, ZIP2, WEIGHT, FEE)</Query>
<Input>ZIP1, ZIP2, WEIGHT</Input>
<Output>ZIP1, ZIP2, WEIGHT, FEE</Output>

```
</Node>
</Nodes>
```

From the feasible service execution plan, WSS generates the following Java codes:

```
Vector getSynthesizedResult_Q (String zip2, String isbn) {
      Vector final_results = new Vector();
      AmazonService amazon_service = new AmazonServiceLocator();
      Amzon amazon_interface = amazon_service.getamazon();
      Enumeration Amazon_Results = Amazon_interface.getBookInfo(isbn);
      While (Amazon_Results.hasMoreElements()) {
        AmazonBook amazon_result =
      (AmazonBook) Amazon_Results.nextElement();
        ShippingService shipping_service = new ShippingServiceLocator();
        Shipping shipping_interface = shipping_service.getshipping();
        Enumeration Shipping_Results =
              shipping_interface.getShippingFee( shipping_interface.getZip1(),
                                                 zip2,
                                                 shipping_interface.getWeight());
        While (Shipping_Results.hasMoreElements()) {
          Int shipping_result = (int) Shipping_Results.nextElement();
          Q_ReturnType result = new Q_ReturnType(zip1, zip2, isbn, shipping_result);
          final_results.add(result);
        }
      }
}
```

Figure 35: Generated Java Codes

We can verify the correctness of this Java program by computing the results with different input values.

4.4 Publish the web service

WSS is responsible to generate a group of Java programs, including a core Java program and several accessorial programs. To make a real web service, we have to manually deploy the composite service using Axis. This step cannot be automated in WSS. It is up to the user's choice on how to publish it.

The author uses WSDD file to deploy the new composite web service using Axis.

4.5 Experimental Results

The experiment proves WSS is a feasible approach for web service composition, especially for database-based web service.

Query rewriting technique reveals the semantic relation of web services. I our tests, the global schema and the service repository do not include many web services and semantic relations, therefore the rewriting results are limited. Theoretically a user query can not be answered if all of the existing services do not include enough data to answer the user query. That means WSS needs as many services as possible so that a composite service can be constructed.

With the proliferation of web services, more and more web services are published, and WSS are expected to be more capable of answering user queries.

5. Conclusion and Future work

5.1 Conclusion

With the proliferation of web services, more and more consumers are using resources in the form of web service. A lot of time the consumer doesn't know what web services are available, and how to identify if a web service is exactly the correct service he needs. Consumer only knows what information he needs to query. We need an easy approach for web service users to consume and integrate existing services. WSS can solve these problems effectively.

WSS use *Web Service Specification* to describe a web service. Web service specification has a signature to define its syntax and a query to define its semantics; and it supplements the missing semantics in WSDL. There is a global schema which abstracts all the relational relations behind all existing web services. The query in a specification is defined on top of the global schema. User's request can be simply described by a query Q, and WSS uses *Query Rewriting* techniques to discover existing services so that the user query Q can be answered. WSS verifies the query rewriting results using the corresponding signatures so that a feasible service execution plan is generated. Finally WSS will generate the *Service Composition* Java codes, which invoke every component service according to the service plan and return the final results as user requested.

Not like current existing frameworks which are mostly at research level and very had to implement, WSS is the most practical approach for web service consumers. WSS uses Datalog notation to define semantics, which is easy to be understood by most of users with basic Database knowledge. Query rewriting module automatically discovers services, and executable Java codes are finally generated. User doesn't need to read the service plan and invoke the services one by one. WSS reduces web service consumers' work dramatically. The generated Java codes can also be reused by future users when the same query is submitted by a different user.

WSS approach is based on query rewriting. The feasibility of WSS is significantly relied on query rewriting techniques. Fortunately query rewriting techniques have been thoroughly studied in database area, especially specialized in data integration, which is similar to web service composition scenario. When there are enough predefined web services, we can always find a complete rewriting if theoretically there is an answer. Many existing techniques and tools in the area of data integration can be applied in our approach. Also, a large number of web services are migrated from existing database-based applications, and it is natual to apply existing techniques in database area, such as query rewriting, to solve web service synthesis problem.

In current implementation, we have the following assumptions:
(1) There is a global schema
(2) A web service can be described by database query semantically
(3) Each web service in all examples only have 1 operation, therefore a query can describe this web service. (only to reduce the complexity)

WSS approach has two major limitations. First the searching speed is limited by the efficiency of query rewriting module. When the number of queries and schemas are large, it might take very long time to compute the rewriting result. And the searching time increases with the number of queries and the size of the schema. Secondly, the semantic expressiveness in WSS is limited by Datalog notation.

5.2 Contribution

This thesis introduces Web Service Synthesis as a new approach to dynamically discover and synthesize web services. In the research, I design the architecture of the Synthesizer, and I develop a prototype Web Service Synthesis system, which can synthesize the implementation of a service from a service specification. I follow modular design to implement this prototype system, and core module can be replaced by external packages if available. I also develop two core modules: the service planning module and the service composition module.

5.3 Future Work

In the future, how to increase the capability of the global schema is a very important issue. When new web service is added, or existing web service is modified or deleted, the schema transformation is another important research issue.

Also, we need to enhance the expressiveness of query in the specification. The accuracy of Web Service Synthesizer mostly depends on the accuracy of query rewriting techniques. From view to global schema, the semantic tie is very strong. Future research may focus on finding the query relations in a loosen condition, especially when global schema definition might be limited.

Efficiency of Web Service Synthesizer can be improved as well. Current implementation doesn't have a parallel execution engine. In service planning phase, component services are planned into sequential order, and therefore the generated Java codes are simply having cascading while loops to invoke component services in the same sequential order. This sequential order doesn't fully represent the dependencies of component services. Some component services can be executed in parallel, and the Synthesizer needs to dynamically detect it and record it. The ideal structure for a service plan is a tree structure, the root is the head predicate, and the leaves are component services.

In more complex case, the planning algorithm can be optimized to generate a tree service plan. The tree starts from the head predicates, and all body nodes are leaves on the tree. If the bound parameter of a node *A* comes from output of another node *B*, the *A* is *B*'s leaf. Figure 36 is an example of such a service plan tree.

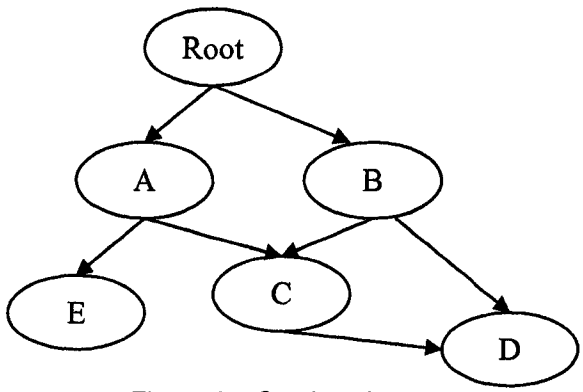


Figure 36: Service plan tree

When such a more complex service plan is generated, how to generate the corresponding Java codes will become another interesting topic. This process is similar to compiling process; the problem would become how to compile a service plan into actual service invocation codes.

In current version, WSS does not support such complex cases. But this is a very interesting topic for future research.

Multiple implementations for one user specification are possible. This happens when there are multiple feasible service plans. From each service plan we can generate an implementation. How to evaluate the efficiency of different implementation codes of the same user specification is another research direction.

Appendix 1: A Experimental Global Schema

We assume there is a global schema that describes the database schema behind all web services.

As experimental environment, we abstract several relations from a typical scenario of purchasing and shipping a book. The following tables are implemented in our Web Service Synthesis Environment.

Amazon

Field Name	Data Type
ISBN	Text
Price	Float
WID	Integer

Distance

Field Name	Data Type
Zip1	Text
Zip2	Text
Dist	Float

BookInfo

Field Name	Data Type
ISBN	Text
Title	Text
Author	Text
Weight	Integer

Warehouse

Field Name	Data Type
WID	Integer
Addr	Text
Zip1	Text

BookReview

Field Name	Data Type
ISBN	Text
Comment	Text
Rate	Integer

Shippping

Field Name	Data Type
Weight	Integer
Dist	Float
Fee	Float

Chapters

Field Name	Data Type
ISBN	Text
Price	Float
WID	Integer

Tables "Amazon" and "Chapters" describe a book's ISBN, its price and the corresponding stocking warehouse ID from two different sources.

Table "BookInfo" records the related information of a book, such as its ISBN, book title, author, weight.

Table "BookReview" records a book's ISBN, reader's comments and its rating.

Tables "Distance" records two zip codes zip1 and zip2, and the corresponding distance between these two zip codes.

Table "Shipping" records the distance, weight and the shipping fee calculation of a ship item.

Table "Warehouse" records the warehouse's universal ID, its detailed address and zip code.

The seven tables above are the abstractions of an application scenario. The real application, such as the web service of Amazon.com, is much more complicated and contains more relations and variables. In our research work, we abstract the core parameters and relations to simplify the illustration of the web service synthesis process. The table definitions are in XML format. These tables are the fundamental relation schema for higher level definitions of web service semantics. The designer of such a global schema should be the expert in this application domain.

The following is the definitions of all seven tables in XML format.

```
<?xml version='1.0'?>
<Schema DatabaseName="BookServices">
<TableName Name="Amazon">
    <ColumnName Name="ISBN" TypeName="VARCHAR"/>
    <ColumnName Name="PRICE" TypeName="FLOAT"/>
    <ColumnName Name="WID" TypeName="INTEGER"/>
    <PrimaryKey>
    </PrimaryKey>
    <ForeignKey>
    </ForeignKey>
</TableName>
<TableName Name="Chapters">
    <ColumnName Name="ISBN" TypeName="VARCHAR"/>
    <ColumnName Name="PRICE" TypeName="FLOAT"/>
    <ColumnName Name="WID" TypeName="INTEGER"/>
    <PrimaryKey>
    </PrimaryKey>
    <ForeignKey>
    </ForeignKey>
</TableName>
<TableName Name="BookInfo">
    <ColumnName Name="ISBN" TypeName="VARCHAR"/>
    <ColumnName Name="TITLE" TypeName="VARCHAR"/>
    <ColumnName Name="AUTHOR" TypeName="VARCHAR"/>
    <ColumnName Name="WEIGHT" TypeName="INTEGER"/>
    <PrimaryKey>
    </PrimaryKey>
    <ForeignKey>
    </ForeignKey>
</TableName>
<TableName Name="BookReview">
    <ColumnName Name="ISBN" TypeName="VARCHAR"/>
    <ColumnName Name="COMMENT" TypeName="VARCHAR"/>
    <ColumnName Name="RATE" TypeName="INTEGER"/>
    <PrimaryKey>
    </PrimaryKey>
    <ForeignKey>
    </ForeignKey>
</TableName>
<TableName Name="Shipping">
    <ColumnName Name="WEIGHT" TypeName="INTEGER"/>
    <ColumnName Name="DIST" TypeName="FLOAT"/>
    <ColumnName Name="Fee" TypeName="FLOAT"/>
    <PrimaryKey>
    </PrimaryKey>
    <ForeignKey>
    </ForeignKey>
</TableName>
<TableName Name="Distance">
    <ColumnName Name="ZIP1" TypeName="VARCHAR"/>
    <ColumnName Name="ZIP2" TypeName="VARCHAR"/>
    <ColumnName Name="DIST" TypeName="FLOAT"/>
    <PrimaryKey>
    </PrimaryKey>
    <ForeignKey>
    </ForeignKey>
</TableName>
<TableName Name="Warehouse">
    <ColumnName Name="WID" TypeName="INTEGER"/>
    <ColumnName Name="Addr" TypeName="VARCHAR"/>
    <ColumnName Name="ZIP1" TypeName="VARCHAR"/>
    <PrimaryKey>
    </PrimaryKey>
    <ForeignKey>
    </ForeignKey>
</TableName>

</Schema>
```

Appendix 2. Two views: "v_amazon" and "v_shipping"

```
<?xml version="1.0"?>
<views>
<view name="v_amazon"> <Query>
   SELECT Amazon.ISBN, Amazon.PRICE, BookInfo.TITLE, BookInfo.AUTHOR, Warehouse.WID,
Warehouse.ZIP1 FROM Amazon, BookInfo, Warehouse WHERE Amazon.ISBN=BookInfo.ISBN and
Amazon.WID=Warehouse.WID
   </Query>
<Input>Amazon.ISBN</Input>
<Output> Amazon.ISBN, Amazon.PRICE, BookInfo.TITLE, BookInfo.AUTHOR, Warehouse.WID,
Warehouse.ZIP1</Output>
</view>

<view name="v_shipping"> <Query>
   SELECT Distance.ZIP1, Distance.ZIP2, Shipping.WEIGHT, Shipping.FEE FROM Distance, Shipping
WHERE Distance.DIST = Shipping.DIST
   </Query>
<Input> Distance.ZIP1, Distance.ZIP2, Shipping.WEIGHT </Input>
<Output> Distance.ZIP1, Distance.ZIP2, Shipping.WEIGHT, Shipping.FEE </Output>
</view>
```

Appendix 3: WSDL of "Amazon" web service

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions targetNamespace="http://137.207.234.209:8080/axis/services/amazon"
xmlns:impl="http://137.207.234.209:8080/axis/services/amazon"
xmlns:intf="http://137.207.234.209:8080/axis/services/amazon"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns1="urn:amazon"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
<wsdl:types>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="urn:amazon">
<import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
<complexType name="AmazonBook">
<sequence>
<element name="ISBN" nillable="true" type="xsd:string"/>
<element name="PRICE" type="xsd:double"/>
<element name="TITLE" nillable="true" type="xsd:string"/>
<element name="WEIGHT" type="xsd:int"/>
<element name="WID" type="xsd:int"/>
<element name="ZIP" type="xsd:string"/>
</sequence>
</complexType>
</schema>
</wsdl:types>
<wsdl:message name="getBookInfoResponse">
<wsdl:part name="getBookInfoReturn" type="tns1:AmazonBook"/>
</wsdl:message>
<wsdl:message name="getBookInfoRequest">
<wsdl:part name="in0" type="xsd:string"/>
</wsdl:message>
<wsdl:portType name="Amazon">
<wsdl:operation name="getBookInfo" parameterOrder="in0">
<wsdl:input name="getBookInfoRequest" message="impl:getBookInfoRequest"/>
<wsdl:output name="getBookInfoResponse" message="impl:getBookInfoResponse"/>
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="amazonSoapBinding" type="impl:Amazon">
<wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="getBookInfo">
<wsdlsoap:operation soapAction=""/>
<wsdl:input name="getBookInfoRequest">
<wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://amazon.services"/>
</wsdl:input>
<wsdl:output name="getBookInfoResponse">
<wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://137.207.234.209:8080/axis/services/amazon"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="AmazonService">
```

```
<wsdl:port name="amazon" binding="impl:amazonSoapBinding">
<wsdlsoap:address location="http://137.207.234.209:8080/axis/services/amazon"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Appendix 4: WSDL of "Shipping" web service

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://137.207.234.209:8080/axis/services/shipping"
xmlns:impl="http://137.207.234.209:8080/axis/services/shipping"
xmlns:intf="http://137.207.234.209:8080/axis/services/shipping"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
<!--WSDL created by Apache Axis version: 1.2.1
Built on Jun 14, 2005 (09:15:57 EDT)-->
    <wsdl:message name="getShippingFeeRequest">
        <wsdl:part name="in0" type="xsd:string"/>
        <wsdl:part name="in1" type="xsd:string"/>
        <wsdl:part name="in2" type="xsd:string"/>
    </wsdl:message>
    <wsdl:message name="getShippingFeeResponse">
        <wsdl:part name="getShippingFeeReturn" type="xsd:double"/>
    </wsdl:message>
    <wsdl:portType name="Shipping">
        <wsdl:operation name="getShippingFee" parameterOrder="in0 in1 in2">
            <wsdl:input name="getShippingFeeRequest" message="impl:getShippingFeeRequest"/>
            <wsdl:output name="getShippingFeeResponse" message="impl:getShippingFeeResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="shippingSoapBinding" type="impl:Shipping">
        <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="getShippingFee">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="getShippingFeeRequest">
                <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://DefaultNamespace"/>
            </wsdl:input>
            <wsdl:output name="getShippingFeeResponse">
                <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://137.207.234.209:8080/axis/services/shipping"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="ShippingService">
        <wsdl:port name="shipping" binding="impl:shippingSoapBinding">
            <wsdlsoap:address location="http://137.207.234.209:8080/axis/services/shipping"/>
        </wsdl:port>
    </wsdl:service>
```

References

[1] W3C, World Wide Web, http://www.w3.org/WWW

[2] Ivan Herman. Semantic Web Activity, Semantic Web of W3C http://www.w3.org/2001/sw/

[3] W3C, Web Service, http://www.w3.org/2002/ws/

[4] W3C, Web Service Definition Language, http://www.w3.org/TR/wsdl

[5] UDDI White Papers, http://www.uddi.org/whitepapers.html

[6] SOAP, http://www.w3.org/TR/soap/

[7] XML 1.0 (third edition), W3C Recommendation, http://www.w3.org/TR/REC-xml/, February 2004

[8] David C. Fallside, Priscilla Walmsley. XML Schema Part 0: Primer 2nd edition, http://www.w3.org/TR/xmlschema-0/, October 2004

[9] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, XML Schema Part 1: Structures 2nd edition, http://www.w3.org/TR/xmlschema-1/, October 2004

[10] Alon Y. Halevy, Answering Queries Using Views: A Survey, The VLDB Journal, Volume 10, Issue 4, December 2001

[11] Jianguo Lu, Yijun Yu, John Mylopoulos, A lightweight approach to semantic web service synthesis, ICDE Workshop, International Workshop on Challenges in Web Information Retrieval and Integration, Tokyo, 2005

[12] John E. Funderburk, Susan Malaika, Berthold Reinwald, XML programming with SQL/XML and XQuery, IBM Systems Journal, Volume 41, No. 4, Pages 642-665, 2002

[13] Katia Sycara, Jianguo Lu, Matthias Klusch, Interoperability among Heterogeneous Software Agents on the Internet, Technical Report , CMU-RI-TR-98-22, CMU, Pittsburgh, USA, 1998

[14] Katia Sycara, Matthias Klusch, Seth Widoff, Jianguo Lu, Dynamic Service Matchmaking Among Agents in Open Information Environments, Journal of ACM SIGMOD Record, Special Issue on Semantic Interoperability in Global Information Systems, A. Ouksel, A. Sheth (Eds.), Volume 28, Issue 1, Pages 47-53, 1999

[15] The OWL Services Coalition, http://www.daml.org/services/owl-s/1.0/

[16] Snehal Thakkar, Craig A. Knoblock, Jose-Luis Ambite, A View Integration Approach to Dynamic Composition of Web Services, In Proceeding of 2003 ICAPS Workshop on Planning for Web Services, June 2003

[17] Minghao Li, Complete and Equivalent Query Rewriting Using Views, Master Thesis, University of Windsor, 2006

[18] Jeffrey D. Ullman, Principles of database and knowledge-base systems, Volume 1, Computer Science Press, 1988

[19] Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju, Web Services: concepts, architectures and applications, Springer, ISBN/ISSN: 3540440089, 2004

[20] Jianguo Lu, John Mylopoulos, XIB: eXtensible Information Broker, International Journal on Artificial Intelligence Tools, Volume 11, No. 1, Pages 95-115, March 2002

[21] Akhil Sahai, Sven Graupner, Web Services in the enterprise: Concepts, Standards, Solutions, and Management, Springer, ISBN/ISSN: 0387233741, 2005

[22] Frank Leymann, Web Services Flow Language, Version 1.0, Technical Report, International Business Machines Corporation (IBM), May 2001

[23] Rachel Pottinger, Alon Halevy, Minicon: A scalable algorithm for answering queries using views. The VLDB Journal, Volume 10, No. 2-3, Pages 182 - 198, 2001.

[24] Satish Thatte, XLANG, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[25] Frank Leymann, WSFL 1.0, IBM, http://xml.coverpages.org/wsfl.html, May 2001

[26] Francisco Curbera, Tony Andrews, Hitesh Dholakia, BPEL4WS, http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

[27] Jianguo Lu, Shengrui Wang, Ju Wang, An Experiment on the Matching and Reuse of XML Schemas, International Conference on Web Engineering, Sydney 2005

[28] David Burdett, Kickolas Kavantzas, Web Service Choreography Model, http://www.w3.org/TR/2004/WD-ws-chor-model-20040324/, March 2004

[29] Web Service Choreography Description Language http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/

[30] Chen Li, Edward Y. Chang, Query Planning with Limited Source Capabilities. International Conference on Data Engineering (ICDE), Pages 401-412, 2000

[31] Chen Li, Computing Complete Answers to Queries in the Presence of Limited Access Patterns, Technical Report, Computer Science Department, Stanford University, 1999

[32] Shigeru Chiba, Javassist (Java Programming Assistant, a sub project of JBoss) http://www.csg.is.titech.ac.jp/~chiba/javassist/

[33] Snehal Thakkar, José Luis Ambite, Craig A. Knoblock, A Data Integration Approach to Automatically Composing and Optimizing Web Services, In Proceeding of 2004 ICAPS Workshop on Planning and Scheduling for Web and Grid Services, June 2004

[34] The Apache Software Foundation, Jakarta Tomcat http://tomcat.apache.org/

[35] The Apache Software Foundation, Axis 1.4, Axis User's Guide http://ws.apache.org/axis/java/user-guide.html

[36] Liangzhao Zeng, Dynamic Web Services Composition, PhD thesis, Univ. of New South Wales, 2003

[37] Susan Malaika, Constance J. Nelin, Rong Qu, Berthold Reinwald, Daniel C. Wolfson, DB2 and Web Services, IBM System Journal, Volume 41, N0. 4, Pages 666-685, 2002

[38] W3C, Document Object Model, http://www.w3.org/DOM/.

[39] Paul A. Buhler, José M. Vidal, Toward the Synthesis of Web Services and Agent Behaviors, 2002

[40] Liming Chen, Nigel R. Shadbolt, Carole Goble, Feng Tao, Simon J. Cox, Colin Puleston, Paul R. Smart, Towards a Knowledge-based Approach to Semantic Service Composition, Springer. Page 319-334, 2003

[41] Nikola Milanovic, Miroslaw Malek. Current Solutions for Web Service Composition, IEEE Internet Computing, Pages 51-59, November 2004

[42] Marco Pistore, Parlo Traverso, Piergiorgio Bertoli, Annapaola Marconi, Automated Synthesis of Composite BPEL4WS Web Services, IEEE International Conference on Web Services (ICWS'05), Pages 293-301, 2005

[43] Jinghai Rao, Xiaomeng Su, A Survey of Automated Web Service Composition Methods, First International Workshop, SWSWPC 2004, Springer, Page 43-54, 2004

[44] Parlo Traverso, Marco Pistore, Automated Composition of Semantic Web Services into Executable Processes, Third International Semantic Web Conference, ISWC 2004, Springer, 2004

[45] Hongbing Wang, Joshua Zhexue Huang, Yuzhong Qu, Junyuan Xie, Web services: problems and future directions. Journal of Web Semantics, Volume 1 No.3, Pages 309-320, April 2004

[46] Jian Yang, Mike P. Papazoglou, Web Components: A Substrate for Web Service Reuse and Composition. Advanced Information Systems Engineering: 14th International Conference, CAiSE 2002 Toronto, Canada, Springer, May 2002,

[47] Snehal Thakkar, José Luis Ambite, Craig A. Knoblock, A Data Integration Approach to Automatically Composing and Optimizing Web Services , In Proceeding of 2004 ICAPS Workshop on Planning and Scheduling for Web and Grid Services , June 2004

[48] Fabio Casati, Ski Ilnicki, Lijie Jin, Adaptive and dynamic service composition in EFlow. In Proceedings of 12th International Conference on Advanced Information Systems Engineering(CAiSE), Stockholm, Sweden. Springer Verlag, June 2000

[49] Brahim Medjahed, Athman Bouguettaya, Ahmed K. Elmagarmid, Composing Web services on the Semantic Web. The VLDB Journal, Volume 12, No. 4, November 2003

[50] Sheila A. McIlraith, Tran Cao Son, Honglei Zeng, Semantic Web services. IEEE Intelligent Systems, Volume 16, No. 2, Pages 46–53, March/April 2001

[51] Alfonso Gerevini, Derek Long, PDDL 3.0, http://zeus.ing.unibs.it/ipc-5/pddl.html

Vita Auctoris

NAME: Chang Zhou

PLACE OF BIRTH: Hanchuan county, Hubei province, China

YEAR OF BIRTH: 1977

EDUCATION: University of Windsor
Windsor, Ontario, Canada
2003-2007 M.Sc in Computer Science

Huazhong University of Science and Technology
Wuhan, Hubei, China
1995-1999 B.Eng in Computer Science and Technology