A TRANSLATOR WEB SERVICE FOR DATA MEDIATION IN WEB SERVICE COMPOSITIONS

by

NITHYA VEMBU

(Under the direction of Prashant Doshi)

ABSTRACT

Atomic Web services may not always provide solutions for business requests. In such cases, several services are integrated to create new composite services with added value. Establishing message exchange between related but independently developed Web services is a key challenge faced during Web service composition. This difficulty is often due to the difference in the schema of the messages of the Web services involved. Data mediation is required to resolve these challenges. To achieve this, we define a formal model for data mediation that considers the names and types of the message elements. Based on this model, we propose methods for resolving different kinds of message-level heterogeneities.

INDEX WORDS: Web Service Composition, Message Level Heterogeneity, Data Mediation, Message Schema, SAWSDL

A Translator Web Service For Data Mediation in Web Service Compositions

by

Nithya Vembu

B.E., SSN College of Engineering, India, 2007

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Athens, Georgia

2011

A TRANSLATOR WEB SERVICE FOR DATA MEDIATION IN WEB SERVICE COMPOSITIONS

by

NITHYA VEMBU

Approved:

Major Professor: Prashant Doshi

Committee: Khaled Rasheed
John Miller

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
July 2011

DEDICATION

To my parents and sister.

ACKNOWLEDGMENTS

I thank my advisor Dr.Prashant Doshi for providing guidance throughout my reseach . I thank Dr.John Miller and Dr.Khaled Rasheed for serving on my committee. I also thank the Institute for AI at UGA for their support and resources.

I would like to thank Dr. John Harney for his valuable advice in programming and help with understanding the concepts related to my research. I would like to thank Muthukumaran Chandrasekaran for helping me format the thesis and proof reading it.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The research behind this thesis concentrates on facilitating composition of diverse but related Web services through data mediation. This chapter presents the need for data mediation in the context of Web services and introduces the process through which we aim to achieve inter-operation of the Web services.

1.1 MOTIVATION

A successful business is one that is able to provide more by utilizing the resources already available to it instead of creating new resources with every new consumer requirement. One approach to achieve this is by using the Service Oriented Architecture [42] that efficiently delivers various functionalities by reusing and merging the existing Web services. The SOA triangle which is formed by the service provider, registry and consumer is shown in Fig. 1.1.

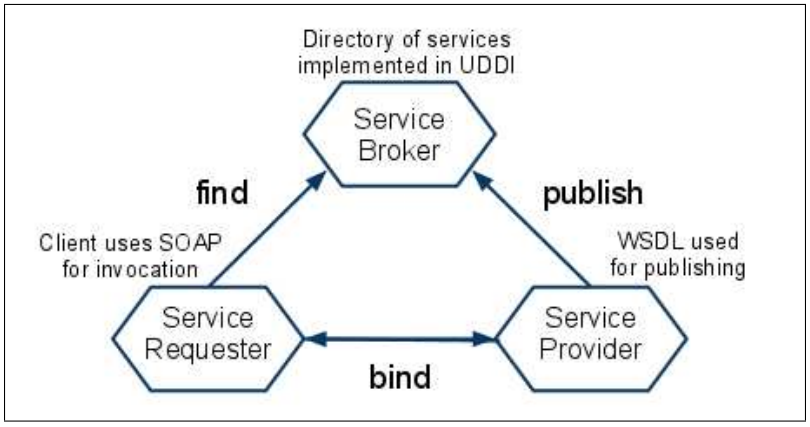

Figure 1.1: The SOA model composed of Web service provider, Web service consumer and the service broker

Web services are hosted independently across distributed systems. They allow interaction between machines over a network by letting the applications exchange data. In cases where a business goal cannot be achieved by a single Web service, multiple Web services could be used to achieve the same. *This process of combining several Web services to produce a service of greater value is called Web service composition*. Web service composition has been a research interest for several years now, with recent focus being on automatic Web service composition techniques.

While it is safe to assume that Web services are interoperable at the syntactic level (in terms of the languages used for representing the message schema), more often than not, message-level heterogeneities exist between them since different Web services are created and published by different vendors. The message level heterogeneities could be of the semantic or structural type. Semantic heterogeneities occur when similar message elements of two different Web services are represented with different names, and structural heterogeneities occur when there is a difference in the data types or structures of the message elements. Many more Web service compositions can successfully be created by resolving these heterogeneities.

The process of making the Web services interoperable by transforming the output message of the predecessor Web service into the input message of the successor Web service, is called data mediation. An approach for performing the data mediation to resolve these heterogeneities is proposed in this paper and suitable test cases are also presented to evaluate the approach.

1.2 CONTRIBUTIONS

The primary goal of this research is to enable message exchange between any Web services that are otherwise unable to do so because of the heterogeneous nature of their schemas. The goal is achieved through a sequence of steps that lead to the following contributions:

- We define the mediation problem as a novel, formal mathematical model. This model accommodates all the sufficient and necessary information that is required from the Web services to perform data mediation.

- We use a middleware-based service based on the data mediation model to resolve the message-level heterogeneities. We call this service the Translator service.
- We show how each of the different types of message-level conflicts that are identified by Nagarajan et al [36] may be resolved using our data mediation model.
- We validate the translator Web service and hence the data mediation model with the support of various test cases that we constructed or obtained externally.

1.3 Structure of this Work

This document is organized as follows. Chapter 2 outlines the Web service composition process and the known message-level heterogeneities that hinder the Web services from being composed. It also summarizes the previous research on data mediation. Chapter 3 explains the proposed data mediation model and its components. Chapter 4 describes the various types of message-level heterogeneities that can exist between two Web services and illustrates how these are solved. Chapter 5 demonstrates the implementation and evaluation of the translator service. Chapter 6 concludes the document with some discussion and possible future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

We have learned that data mediation is required to make related but dissimilar Web services participating in a Web service composition inter-operable. To understand the context of this research better, in this chapter, we present an overview of Web service composition and the message-level heterogeneities in it. We also discuss the past attempts in resolving heterogeneities between Web services.

2.1 WEB SERVICE COMPOSITION AND DATA MEDIATION

Long gone are the days when applications were localized software programs. With the rise of the Internet, applications became Web applications using Web services that are published and available over the World Wide Web. These are not only machine independent but are also platform independent and support many programming language. This reduces redundancy to a great extent because a service providing some functionality can be created just once and reused by all applications rather than manually incorporating those functions in every application. Web services have gained wide popularity because of their reusability and their ability to allow exchange of data between applications across a network. A Web service is associated with three entities: the service provider, the service registry and the service consumer. The service provider describes the operations of a Web service and its address with the help of the Web Service Description Language (WSDL) [20]. Web services exchange message using the Simple Object Access Protocol (SOAP) [17] and they are listed in the registries implemented using Universal Description, Discovery and Integration or UDDI [12].

Some Web services provide functionality by combining the functionalities of several other Web services. This is the process of Web service composition. Two approaches to composing Web services are orchestration and choreography. In an orchestration, there is a central orchestrator that controls the order in which the Web services are invoked and the operations in the composition in order to accomplish a goal. The individual Web services act as independent services and are not aware that they are a part of a composition. Fig. 2.1(a) represents an orchestration.

Business Process Execution Language for Web Services or BPEL4WS [5] is an orchestration language for describing how Web services should be composed into processes. These processes can later be executed with the help of orchestration engines such as ActiveBPEL [1], Oracle BPEL Process Manager [9] and others. A typical BPEL process includes information on how the Web services in the composition interact, how the messages are exchanged between them and how to handle exceptions if and when they occur. BPEL4WS contains activities such as 'invoke', 'receive', 'reply', 'assign' and 'wait' that will be used by the central controller to coordinate the process.

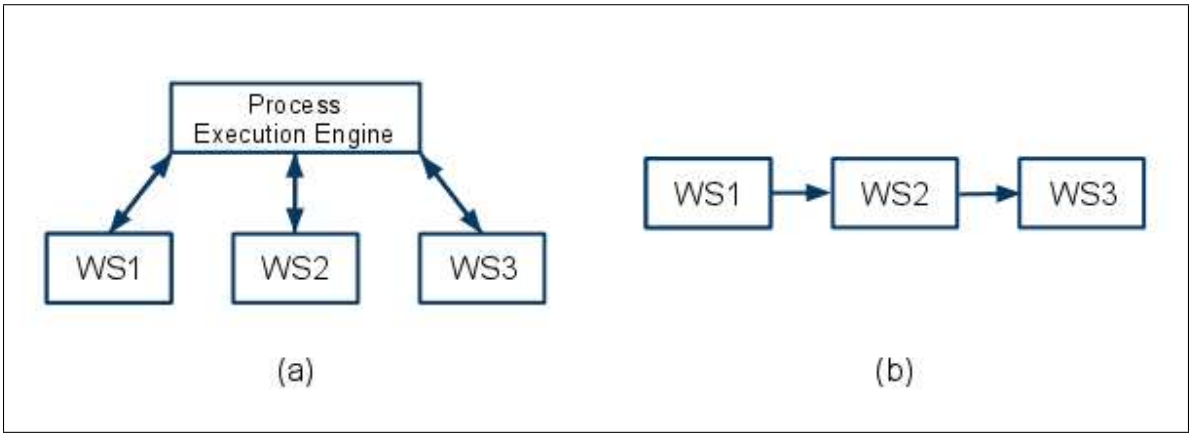

Figure 2.1: Service Composition a) Orchestration, where a central coordinator provides a controlled environment b) Choreography, where the functionality is delivered through a collaborative effort of individual Web services. Both types of service composition might require data mediation

In a choreography, the central coordinator is absent. The Web services know their roles in the composition. For the process to be successful, the individual Web services should be made aware of the operations to execute, what messages to exchange and more importantly when to exchange the messages. This makes fault handling in orchestration easier than in choreography. Fig. 2.1(b)

depicts choreography. Web Service Choreography Interface [14] and Web Service Choreography Description Language [13] are examples of choreography languages.

Over the years, many approaches have been developed for Web service composition. Microsoft Biztalk [8] and BEA WebLogic [4] are composition engines used when the Web services and the way in which they are going to interact with each other are pre-determined. This kind of composition technique will work only when the components of the composition are static. Other composition techniques are not this restrictive and work in dynamic environments. E-Flow [19], StarWSCoP (Star Web Services Composition Platform) [51], METEOR-S [16], WebTransact [44], DynamiCoS [29] [49] and SeGSeC [23] are dynamic Web service composition approaches. SELF-SERV [48] is a framework for composing Web services in a declarative manner. SWORD [46] creates rule-based Web service compositions where a Web service is treated as a rule. There are many Web service composition techniques based on AI planning methods. RET-SINA [16] and SHOP2 [52] are planning systems for Web service composition based on Hierarchical Task Network. Doshi [21] and Gao [24] have presented Web service composition techniques by applying MDPs(Markov Decision Processes).

Although there are minimal problems of incompatibility at the software environment level and Web service composition seems like an easy process, issues at the message-level often exist. By message-level heterogeneity, we mean the difference in terms of semantics, structure and syntax, of the messages being exchanged. The terms that are repeatedly used in a definition of a message-level conflict would be concepts, attributes or data types. In the example of a Web service message schema given below in Fig. 2.2, “Institution” is the concept, “Name” and “Population” are the attributes, and “String” and “Integer” are their corresponding data types.

Nagarajan et al. [36] have further classified semantic and structural heterogeneities as attribute level incompatibilities, entity or concept level incompatibilities and abstraction level incompatibilities as shown in Fig. 2.3. Methods for resolving each of these conflicts have been implemented in this paper. A brief description of the types of message-level incompatibilities as defined in [36] is provided below.

```
<xsd:schema targetNamespace="http://tempuri.org/WS3/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="Institution" type="tns:InstiInfo"/>
    <xsd:complexType name="InstiInfo">
      <xsd:sequence>
              <xsd:element name="Name" type="xsd:string"/>
              <xsd:element name="Population" type="xsd:integer"/>
              <xsd:element name="Address" type="xsd:string" sawsdl:modelReference=
              "http://daml.umbc.edu/ontologies/ittalks/address#Address"/>
          </xsd:sequence>
      </xsd:complexType>
</xsd:schema>
```

Figure 2.2: A sample Web service message schema from the WSDL of a test Web service

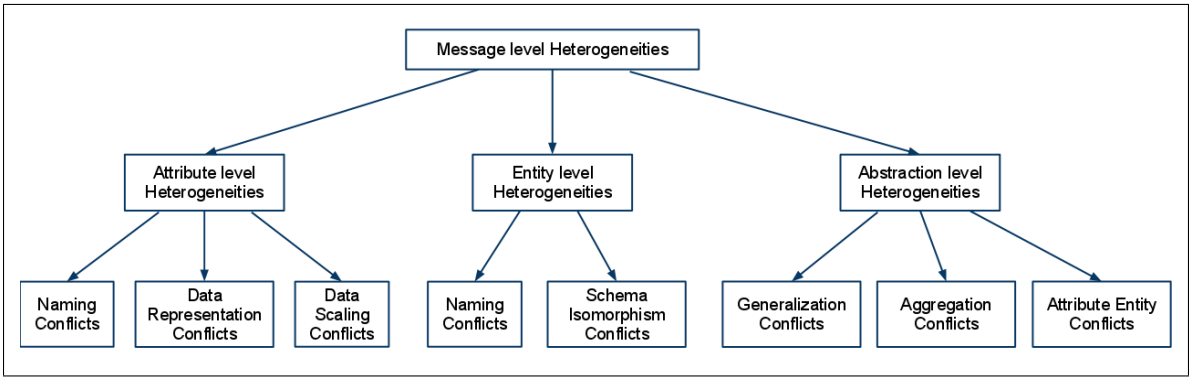

Figure 2.3: Classification of message-level heterogeneities by Nagarajan et al. [36]

2.1.1 Attribute Level Incompatibilities

Attribute level incompatibilities are those that occur in attributes that are semantically similar but are different in terms of their description and structure. Some of the attribute level incompatibilities are naming conflicts, data representation conflicts and data scaling conflicts.

Naming conflicts among attributes occur when two semantically similar attributes are called by different names. An example for attribute level naming conflicts is the following:

WS_1: Student (Name, Grade)

WS_2: Student (Name, Score)

Note that in the above example, WS_1 represents its output message and WS_2, its input message. This would be the format used in all future examples. Even though the names are different, 'Grade' and 'Score' are semantically similar or related, and it would be correct to pass on the value of 'Grade' to 'Score' with a simple transformation if needed, if WS_1 and WS_2 are in a composition.

Data representation conflicts exist between attributes when semantically similar attributes are conflicted by their datatypes or how they are represented. For example

WS_1: Student (Name, Grade), (String, Float)

WS_2: Student (Name, Score), (String, Int)

In the above case, even though 'Grade' and 'Score' are semantically similar, their datatypes are different.

Data scaling conflicts are said to occur when the difference in representation or format of data can be resolved by processes like scaling or shifting of the data based on some context information. For example,

WS_1: Object (Length, Width), (Float, Float)

WS_2: Object (Length, Width), (Float, Float)

Consider that the SAWSDL (Semantic Annotations for WSDL) [27] model references of both attributes in WS_1 points to an URI "http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#meter" and the model references of both attributes in WS_2 are "http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#centimeter". This shows that WS_1 is expecting the values of its attributes in 'meters' and WS_2 is expecting it in 'centimeters'. The heterogeneity between the two messages can be resolved by converting the attributes of WS_1 to WS_2 using the appropriate scaling rule.

2.1.2 CONCEPT LEVEL INCOMPATIBILITIES

These incompatibilities occur when semantically similar entities are represented with different names or different structures. Concept level incompatibilities could be classified as Naming conflicts or Schema Isomorphism conflicts.

Similar to Naming conflicts in attributes, concept level Naming conflicts occur when two semantically similar entities are called by different names.

WS_1: Institution (Name, Population)

WS_2: University (Name, Population)

In the given example, though 'Institution' and 'University' are semantically similar, the two Web services will be rendered as not composable since the entities have different names.

Schema isomorphism is a type of structural heterogeneity that occurs when semantically similar concepts have dissimilar number of attributes. An example of Web services having a schema isomorphism conflict is given below:

WS_1: Institution (Name, AptNo, StreetAddr, City, Zip)

WS_2: University (Name, Address)

Even though the concepts 'Institution' and 'University' are semantically similar, they have unequal number of attributes. Direct mapping of attributes based on their names and datatypes will not be possible in this case. Additional context information will be necessary for mediation in this case.

2.1.3 ABSTRACTION LEVEL CONFLICTS

Sometimes, semantically similar concepts from a domain are placed at different levels when represented in different schemas. When they are used as concept names in messages, it prevents the Web services involved from participating in a composition even though they are typically representing the same kind of information through their attributes. Also, the kind of incompatibilities we saw so far were between the same kinds of schema elements, i.e., they were at the concept level

or at the attribute level. Abstraction level incompatibilities can occur between an concept and an attribute when an concept in one message schema is modeled as an attribute in another. Generalization conflicts, aggregation conflicts and attribute-entity conflicts are types of abstraction level incompatibilities.

Generalization conflicts occur when semantically similar concepts are represented with different names, one more general than the other. An example of generalization conflict is given below:

WS_1: PhDAlumnus (Name, Dept, Email, Website)

WS_2: Alumnus (Name, Dept, Email, Website)

'Alumnus' and 'PhDAlumnus' are different labels but 'Alumnus' is a generalization of the concept, 'PhD Alumnus'. Therefore, it is possible that both the entities could be holding information about the same candidate but are assumed to be otherwise due to the difference in their representation.

Aggregation conflicts occur between messages when a collection of the concept in one represents the concept in another or when an concept in one message can be considered as a part/division of the concept in another. An example of aggregation conflict is presented below:

WS_1: UndergraduateStudent (Name, ID, Dept, Email)

WS_2: StudentBody (Name, ID, Dept, Email)

A student body is made of graduate students, undergraduate students, etc. Therefore, the concept "UndergraduateStudent" is a division of the concept "StudentBody".

Attribute-entity conflicts happen when a concept in one message schema is an attribute in another. Consider the following example,

WS_1: Resident (Name, Address, Phone)

WS_2: AreaInfo (Resident, Address, Phone)

The concept "Resident" in WS_1 is present as an attribute in WS_2. Unlike the other message-level conflicts, this conflict cannot be mediated in a straight forward manner. The resolution of the Attribute-Entity conflict is described in detail later.

The given message-level conflicts describe situations where semantic correspondences can be found between heterogeneous attributes/concepts. Once, these attributes/concepts have been matched, a mapping is required to actually transform one attribute/concept to the format of another. Some languages used for representing these mappings are XQuery [40] and XSLT (Extensible Stylesheet Language Transformations) [15]. The next section presents the past endeavors in data mediation.

2.2 Previous Work

Much of the previous works in mediation have concentrated on schema mapping techniques [45] and there is a lack of well-rounded research on Web services in the context of data mediation. Web Services Modeling Ontology or WSMO [22] is a conceptual framework for semantic Web services. Web Service Modeling Language (WSML) [28] is an ontology language that supports the WSMO architecture. The main constituents of the framework are Ontologies, Web Services, Goals and Mediators. Mediators are used when heterogeneous situations arise. The different types of mediators defined by WSMO are OO Mediators, GG Mediators, WG Mediators and WW mediators where O stands for ontology, G for goal and W for Web service. WSMX (Web Services Modeling Execution Environment) [25] [39] is an implementation for the WSMO framework that lets service providers register their services with it and requesters to discover and invoke Web services. The data mediator component of WSMX implements only the OO mediator from the WSMO specification. Cabral and Domingue [18] propose a broker-based mediation approach following the WSMX framework, to compose semantic Web services. Contrary to the WSMX implementation, this research focuses on mediation between WSDL Web services.

Mrissa et al. [35] present an approach based on their context model [34] to extend WSDL specifications to allow inclusion of context information for resolving semantic Web services. Another

context based mediation approach [32] for BPEL processes uses the COIN (Context Interchange) lightweight ontology that describes generic concepts. XPath functions are used to perform transformations based on the context differences between the service descriptions. The idea of using a common ontology to support mediation between Web services has been partially used in our paper to resolve the data scaling conflicts.

Li et al. [31] describe an algorithm for classification, detection and resolution of semantic conflicts in OWL ontologies. The conflicts defined in this paper are somewhat parallel to the message-level conflicts defined by Nagarajan et al. [36] who have proposed an approach for data mediation that utilizes the SAWSDL *schemaMapping* attributes. Using the mapping defined in the *liftingSchemaMapping* attribute, the output message of the first Web service is mapped to an ontology instance and then the ontology instance is transformed to the input message of the second Web service, using the *loweringSchemaMapping* mapping.

BPEL for Semantic Web Service or BPEL4SWS [38], an extension of BPEL 2.0 aims at message exchange independent of WSDL. Its *mediate* element is an extension of the BPEL *Assign* operator and is helpful for ontology based mediation [37]. BPEL4SWS and ontology based mediation are implemented in the SUPER project [11]. WSIRD [50] is a rule based engine that analyses OWL-S descriptions to transform a message into the format of another. Leitner et al. [30] present several mediation strategies that have been fused into their Web service invocation framework DAIOS [41], to resolve interface level conflicts. The authors claim that their approach is more flexible than other middleware-based mediation approaches because unlike those approaches, the communication between the client and the provider is not disconnected.

Since, the mediation technique proposed in this paper is independent of the schemas of the messages exchanged; any two related Web services whose WSDL definitions are accessible may be mediated. A mathematical model for data mediation is defined and transformation rules are created dynamically using the model.

CHAPTER 3

A FORMAL MODEL FOR DATA MEDIATION

In a typical Web service composition, the output message of the first Web service is fed as the input for the second one along with any other data in hand. As it has been mentioned before, mediation will be required if there is a mismatch between the output message of the first Web service, the message on hand from previous invocations and the input message that the second service is expecting.

The choice of Web services taking part in a composition is variable. It is not practical to have individual Web services tailor made to perform particular functions nor is it possible to dynamically modify a Web service to handle data mediation implicitly. Based on these facts, it is preferable to have an external mediator service map the two messages involved. This thought lead to the derivation of a general data mediation model that will allow many of the previously incompatible Web services to be more inter-operable. The derived definition for the data mediation problem serves as the interface for a translator Web service, which will identify the type of message heterogeneity that exists between the two Web services under consideration, and then provide a set of semantic and structural rules to transform the output message of the predecessor Web service into the format of the input message of the successor Web service.

The different types of message-level heterogeneity include Naming Conflicts, Date Representation Conflicts, Data Scaling Conflicts, Schema Isomorphism conflicts, Generalization, Aggregation Conflicts, and Attribute-Entity conflicts. The different conflicts are further categorized on the basis of the contextual information needed for solving them. XSLT is used for writing the rules for transformation. The various conflicts and their resolution have been explained later in this paper. Our mediation model is described in detail in the next section.

3.1 Data Mediation Problem Definition

The data mediation problem has been formally defined as follows:

$$\textbf{\textit{MD}} = <WS_1,\ WS_2,\ M_1,\ M_2,\ M_h,\ \textbf{\textit{T}}>$$

WS_1 and WS_2 are the Web services that need mediation i.e. the output message M_1, of WS_1 needs to be transformed to input message M_2, of WS_2. M_h is the message available if M_1 is insufficient for mediation. There may be cases where WS_2 needs data from the mediated WS_1 as well as some external source. M_h is assumed to be this external source.

A message ***M*** is a tuple, ***M*** =<***MS, MR***>, where ***MR*** is the format of the message representation and ***MS*** is the message schema which is composed of concepts, attributes and their datatypes. Hence, message schema can be represented as ***MS*** =<***C, A, D***>. Any concept ***C*** can be further broken down into ***C*** =<N_C, MR_C>, where N_C is the name of the concept and MR_C is a reference to a semantic model or another Web service itself. Similarly, any attribute can be represented as ***A*** =<N_A, MR_A>. ***D*** =<***N, R***> contains information on the corresponding datatypes of all the attributes. Here, ***N*** is the set of all datatypes' names. For example: The values in ***N*** could be *Int*, *String*, etc. ***R*** is the set of corresponding restrictions on the values that a particular attribute can hold. When XSD (XML Schema Definitions) is used to describe the Web services' schema, there is a provision to define acceptable values for the attributes present. These are called XSD restrictions/facets. They are particularly helpful when resolving data representation conflicts.

$T : M_1$ x M_2 x $M_h \rightarrow \rho$ is the translation function that provides the set of semantic and structural rules, ρ, for transforming M_1 and M_h into M_2.

3.2 Translation Function

Often, the output of one Web service is translated to the input of another by using custom made service-specific or domain-specific mappings. As the name suggests, these mappings have very limited functionality; in the case of service-specific mappings, new mappings have to be created every time new services are created while domain-specific mappings may be inapplicable beyond

that particular domain. This is what our translation function aims to overcome. Other mappings are based on the underlying structure of the schema but our translation function is independent of the it and uses only the schema components to arrive at the mapping dynamically. Any two Web services whose messages are resolvable can be mediated with the help of this function.

In the previous section, the translation function was defined as $T : M_1$ x M_2 x $M_h \rightarrow \rho$. Programmatically, this means that the messages M_1, M_h and M_2 will be the input to the translator Web service which will produce the rule set as its output. The rule set is an XSLT style sheet providing the template to transform one message into another. This template will be applied to the output SOAP message of WS_1. The SOAP message and the message *M* in the data mediation problem definition are not the same and cannot be used interchangeably.

From the definition of the data mediation problem, a message is *M* =<*MS, MR*> and its schema is *MS* =<*C, A, D*>. The values for the components in the message schema *MS* is obtained from parsing the WSDL documents of the two Web services involved in the data mediation process. The values of the components of a message schema *MS* are used to check if one or more of the earlier mentioned message-level conflicts exist and whether or not they can be resolved. If the conflicts that exist are all resolvable, then the XSLT template to do the actual transformation is created. To understand how we arrive at the rule set for performing the translation, we need to look into the translator Web service and the Web service composition that acts as the translator Web service' client in detail.

3.3 TRANSLATOR SERVICE

In the process of fashioning a more versatile method for resolving the message-level heterogeneities in Web services and to avoid having to create and store custom mappings for individual Web services or Web services in similar domains, we have arrived at an independent middleware-based Web service which can mediate two Web services based on certain conditions. Suppose, there are two Web services WS_1 and WS_2 that are going to participate in a Web service composition. Then the translation function *T* theoretically represents the functionality of the translator Web

service i.e. it uses the messages M_1 (from WS_1), M_2 (from WS_2) and occasionally M_h that holds outside information, to make a rule set ρ that will mediate WS_1 and WS_2. Fig. 3.1 is an abstract representation of the translator Web service depicting the inputs it is expecting and the output it gives out.

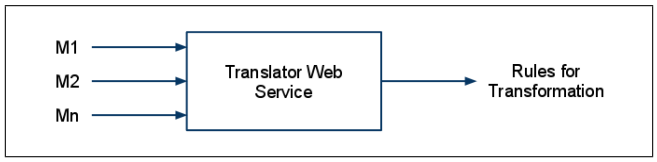

Figure 3.1: Model of the translator Web service showing its inputs and output. Here, *M* =<*MS, MR*> according to our data mediation model. The message schema *MS* =<*C, A, D*> comprises of concepts, attributes and datatypes, and MR is the message representation format

A situation where the translator Web service is going to be used to mediate two Web services can be seen as a sequence where the invocation of Web services is in the order: WS_1, Translator Web service and WS_2. This sequence is shown in Fig. 3.2.

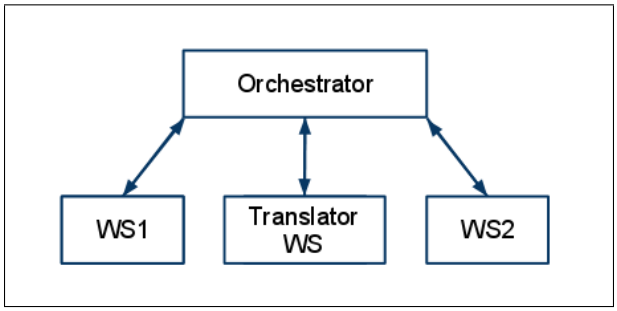

Figure 3.2: The sequence of invocations controlled by the composition acting as both the orchestrator and the translator WS's client

Modern practices would use a BPEL process to define this sequence of invocation but we have used Java. BPEL is portable and can be easily monitored but despite its general advantages, the Java process proved to be the right fit than BPEL for this implementation because it was easier to use the simplicity and expressiveness of Java to program the mediation function rather than extending BPEL to incorporate data mediation. This composition serves a dual purpose which is being the orchestrator that mimics the behavior of a BPEL process and acting as the client for the translator Web service.

After WS_1 returns its output message, the preparations necessary for invoking the translator service are done. This primarily involves parsing the WSDL file of WS_1 and WS_2. From the WSDL definition of the translator Web service in Fig. 3.3, we can see that it adheres to the defined data mediation model and it also gives us an idea as to what the translator Web service is expecting as its input. This chapter is divided into two sections based on what happens before invocation of the translator Web service and what happens after. The following section describes the processes necessary and prior to invoking the translator Web services.

3.3.1 BEFORE INVOKING THE TRANSLATOR WEB SERVICE

The API of Apache AXIOM [3] is used to parse the WSDL definitions of the two Web services that require mediation. This API lets us conveniently exploit the tree structure of the WSDL file through the method of pull parsing. The advantage of this method of parsing is in how it allows iteration through the XML structure repeatedly and in how the tree and its various sub-trees can be used as variables in any method that is handling them.

Suppose Fig. 3.4, is the WSDL definition of one WS_1. Here is the summary of how the WSDL document is pull-parsed to obtain instantiations for the components of messages M_1 and M_2.

1. Create two AXIOM document elements out of the two WSDL definition documents and pass them one by one to the local method that will parse them.

2. Next, the parser will be asked to find the WSDL element *portType*. If the document element was tagged as "WS_1", the name of output message is returned and if it was tagged as "WS_2" then the name of the input message is returned.

3. Now, the parser will locate the WSDL element "message" corresponding to the names of the messages returned in the previous step. This is necessary to obtain the XSD element that is abstractly defining the data that will be transmitted or received.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://tempuri.org/Translator/" xmlns:wsdl="http://schemas.xmlsoap.org/
wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="Translator"
targetNamespace="http://tempuri.org/Translator/">
  <wsdl:types>
    <xsd:schema targetNamespace="http://tempuri.org/Translator/" xmlns:xsd="http://
www.w3.org/2001/XMLSchema">
       <xsd:element name="OutputElem" type="xsd:string"/>
       <xsd:element name="InputElem" type="tns:Message"/>

       <xsd:complexType name="Attribute">
       <xsd:sequence>
          <xsd:element name="Name_Attribute" type="xsd:string"/>
          <xsd:element name="ModelRef_Attribute" type="xsd:anyURI" nillable="true"/>
       </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Concept">
       <xsd:sequence>
          <xsd:element name="Name_Concept" type="xsd:string"/>
          <xsd:element name="ModelRef_Concept" type="xsd:anyURI" nillable="true"/>
       </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="DataTypes">
       <xsd:sequence>
          <xsd:element name="Type" type="xsd:string"  maxOccurs="unbounded"/>
          <xsd:element name="Restriction" type="xsd:string"  maxOccurs="unbounded"/>
       </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="MessageSchema">
       <xsd:sequence>
          <xsd:element name="Concept" type="tns:Concept" maxOccurs="unbounded"/>
          <xsd:element name="Attribute" type="tns:Attribute" maxOccurs="unbounded"/>
          <xsd:element name="DataTypes" type="tns:DataTypes"/>
       </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Message">
       <xsd:sequence>
          <xsd:element name="MessageSchema" type="tns:MessageSchema"/>
          <xsd:element name="MessageRepresentation" nillable="true" type="xsd:string"/>
       </xsd:sequence>
    </xsd:complexType>
    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="TranslatorResponse">
    <wsdl:part name="part1" type ="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="TranslatorRequest">
    <wsdl:part name="message1" type="tns:Message"/>
    <wsdl:part name="message2" type="tns:Message"/>
  </wsdl:message>
  <wsdl:portType name="Translator">
    <wsdl:operation name="TranslatorOperation">
      <wsdl:input message="tns:TranslatorRequest"/>
      <wsdl:output message="tns:TranslatorResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="TranslatorSOAP" type="tns:Translator">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="TranslatorOperation">
      <soap:operation soapAction="http://tempuri.org/Translator/NewOperation"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="Translator">
    <wsdl:port binding="tns:TranslatorSOAP" name="TranslatorSOAP">
      <soap:address location="http://tempuri.org"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Figure 3.3: The WSDL of the Translator Web service. The input message schema is based on the defined formal mediation model. Output is the XSLT transformation rules returned as a string.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http:
//tempuri.org/WS2/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http:
//www.w3.org/2001/XMLSchema" name="WS2" targetNamespace="http://tempuri.
org/WS2/" xmlns:sawsdl="http://www.w3.org/ns/sawsdl">
  <wsdl:types>
    <xsd:schema targetNamespace="http://tempuri.org/WS2/" xmlns:xsd="http://www.w3.
org/2001/XMLSchema">
      <xsd:element name="resp" type="xsd:string"/>
      <xsd:element name="Student" type="tns:StudentInfo"/>
      <xsd:complexType name="StudentInfo">
        <xsd:sequence>
          <xsd:element name="Name" type="xsd:string" sawsdl:modelReference="http:
//daml.umbc.edu/ontologies/ittalks/person#Person"/>
          <xsd:element name="Score" type="xsd:double"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="WS2Response">
    <wsdl:part element="tns:resp" name="part1"/>
  </wsdl:message>
  <wsdl:message name="WS2Request">
    <wsdl:part element="tns:Student" name="part1"/>
  </wsdl:message>
  <wsdl:portType name="WS2">
    <wsdl:operation name="WS2Operation">
      <wsdl:input message="tns:WS2Request"/>
      <wsdl:output message="tns:WS2Response"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="WS2SOAP" type="tns:WS2">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="WS2Operation">
      <soap:operation soapAction="http://tempuri.org/WS2/NewOperation"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="WS2">
    <wsdl:port binding="tns:WS2SOAP" name="WS2SOAP">
      <soap:address location="http://tempuri.org/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Figure 3.4: The WSDL of a test Web service

4. The XSD element's name is the N_C in the data mediation problem definition. Had there been reference to any semantic model using SAWSDL, then that would serve as the MR_C in the same definition.

5. The WSDL element *"types"* provides the message schema definitions and by iterating through the "types" element, we are able to obtain the attributes $A = <N_A, MR_A>$ and their corresponding datatypes $D = <N, R>$ for the messages M_1 and M_2.

The XSD datatype restrictions corresponding to the restrictions *R* in $D = <N, R>$ are a predefined finite number. Based on what kinds of restrictions are mentioned for each attribute *A* in the WSDL definition, the values of its *R* are encoded with a set of characters in order to facilitate their manipulation inside the translator Web service. The way in which restrictions are encoded here is by using the first three characters of the type of restriction(s). Then, this encoding is used as the value for *R*. For example, if there is a restriction on length for a particular attribute, then its *R* will be set with "len[min-max]" where 'len' represents the type of restriction and the 'min' and 'max' values give the range of the length. The other restrictions are handled in a similar manner. The values for *MR*, which would be URIs, are manually set and it is assumed that they will be the same for WS_1 and WS_2. Now, the process of parsing values for messages M_1 and M_2 is complete and the translator Web service is ready to be invoked.

3.3.2 Post invocation of the translator Web service

After the translator Web service is invoked, the input messages M_1 and M_2 are used to create two AXIOM document elements that will hold the messages as XML tree structures. Out of the two components *MS* and *MR* of a message *M*, *MS* is largely used in the data mediation process inside the translator Web service. The next step in the process is checking if one or more of the various message-level conflicts exist between the two messages. It is necessary that every conflict that is present is resolved. So, if even one of them cannot be resolved, then the heterogeneity problem between the two Web services is declared irreconcilable.

The search for conflicts always starts with checking if there is a naming conflict between the two concepts C_1 and C_2. The mediation process is abandoned here if the concept names could not be matched either syntactically or semantically, and the next conflict is addressed if it is otherwise. Intuitively, the next conflict to be checked for and resolved if possible would be the naming

conflicts for attributes. This is the next step taken when the number of attributes is same for both the messages while certain exceptions in matching are made when the number of attributes is different. After the Naming conflicts are verified, the rest of the conflicts such as Date Representation Conflicts, Data Scaling Conflicts, Schema Isomorphism conflicts, Generalization, Aggregation Conflicts, and Attribute-Entity conflicts are also handled.

The key to resolving these conflicts lies in intelligently constructing the transformation rules as and when a type of conflict is identified. Since the translator Web service is operating independently, and because the message schema of two services requiring mediation are broken down to and accessed at the attribute level, we are able to efficiently achieve the automation of the production of the rule set for transforming one message into the format of another. The description of each conflict along with when it is resolvable and how it is resolved is given in *Chapter* 4. The order in which the conflicts are handled and how the set of rules for transformation is built is also shown in the following chapter.

The transformation rules are built as an XML structure using the StAX-based [7] builder in the Apache AXIOM API. At the end of the process of identifying the conflicts, if it was inferred that they can be resolved, then the translator Web service returns the transformation rule set as its output. It should be noted that the actual transformation of one message into another format has not occurred yet but only the rules for transformation have been created.

The output of the translator service is stored as an XSLT file that will be applied on the output SOAP message of WS_1. The output of this transformation is an XML structure that can be used as the input SOAP message for WS_2 after slight modifications such as adding the envelope, namespaces, etc. However, in the current implementation where WSDL2Java Web services are used, the values for the message elements are first pulled from the XML structure and then individually assigned. This individual assignment of values for the elements is similar to the 'Assign' function in BPEL. If WS_2 is successfully invoked using the message obtained through the mediation process, we know that the data mediation problem between the two involved Web services was successfully resolved by the translator Web service.

CHAPTER 4

CONFLICTS AND RESOLUTION

In chapter 2, we summarized the different message-level heterogeneities identified by Nagarajan et al [36]. In this chapter, we provide the techniques that our translator Web service explained earlier, uses, for resolving each of these message-level heterogeneities. We assume that Web services are specified using SAWSDL thereby allowing model references, if needed.

4.1 ATTRIBUTE LEVEL INCOMPATIBILITIES

The methods for resolving various attribute level conflicts are presented below.

4.1.1 NAMING CONFLICTS

We use the example provided earlier in Chapter 2 for naming conflicts in order to describe its resolution,

WS_1: Student {(Name, Grade), (String, Int)}

WS_2: Student {(Name, Score), (String, Int)}

Both Web services are about the same concept 'Student'. They also have the same number of attributes. However, one of the attributes does not have the same name as another. So, there is a naming conflict present. Even though the names are different, 'Grade' and 'Score' are semantically similar, and it maybe likely correct to use the value of 'Grade' for 'Score' since the datatypes are identical. *This means that the naming conflict is resolvable if the translator WS can identify the two attributes as being synonyms of one another with the help of a lexicon, thesaurus or a semantic reference.* We use WordNet [33] for this purpose. Hence, if we are able to match every attribute

from WS_1 directly or semantically to one or more attributes in WS_2, then we have resolved the naming conflict for attributes. It should be duly noted that, it is not sufficient if just the attribute names are matched i.e. we must ensure that the attributes are referring to the same concept. To understand this, consider the following example:

WS_1: Student {(Name, Grade), (String, Int)}

WS_2: Game {(Name, Score), (String, Int)}

The attributes could be seen as semantically similar in the case given above but they do not pertain to the same concept and it would not make sense in using the values of 'Name' and 'Grade' from WS_1 to instantiate 'Name' and 'Score' in the input message of WS_2. Therefore, this type of naming conflict cannot be resolved.

4.1.2 Data Representation Conflicts

Consider the example,

WS_1: Student {(Name, Grade), (String, Float)}

WS_2: Student {(Name, Score), (String, Int)}

In the above example, even though 'Grade' and 'Score' can be matched to be semantically similar, their datatypes are different. WSDL definitions use XSD datatypes. So, the datatype of any attribute comes from a set of finite number of datatypes and we have complete knowledge of each datatype in terms of what kind of values it represents, the range of values it can take, the restrictions on it, etc. The following paragraphs explain how data representation can be resolved.

The first step in this process would be checking if the semantically matched attributes are representing the same type of data. A datatype falls under one of these categories: numeric, string, date/time, URI, byte or object type. If the two attributes under consideration have datatypes in the same category/type, then there is a chance that the conflict could be resolved. If they don't fall under the same category, then the process is exited and the data mediation problem between the two

Web services is declared irreconcilable. However, an exception is made when the two datatypes being matched are 'String' and 'URI'/'anyURI'.

Sometimes, data that are not of string type is represented as a string. If an attribute of a string datatype and that of another datatype have semantically similar labels, then the value of one maybe assigned to another if their names are semantically similar and both their model references are referring to the same entity. Since the data itself is not available, it is risky to make assumptions on the kind of data (int, float, date, etc) that the string attribute is holding. Therefore, we do not assign values from an attribute of a string datatype to that of numeric or date type, but we do so the other way round.

Even if the datatypes are representing the same type of data, the actual values that an attribute of that datatype is allowed to hold differ widely. For example, even though 'int' and 'long' represent numeric data, 'int' holds 32-bit signed integers whereas 'long' can hold 64-bit signed integers. Furthermore, even when the length of data it can hold is same, the range of values it is allowed to have can be different. For example, 'int' and 'unsignedInt' are 32-bit long but 'unsignedInt' cannot have negative numbers as its value. Similarly, there is a difference in datatypes in terms of datatypes that allow decimals and those that do not. Keeping in mind that the actual data from the output message of WS_1 is not available to the translator WS, all these factors become crucial in resolving the data representation conflict. The translator WS first checks if the datatypes are of the same category; then it checks the range/pattern of values both the datatypes are allowed to hold. Suppose, the two attributes a_1 and a_2 being currently matched from M_1 and M_2 have datatypes d_1 and d_2 correspondingly. For numeric types, if Range-of-values (d_2) $\geq$ Range-of-values (d_1), then the conflict can be resolved. In the case where d_2 does not allow decimals in its values but d_1 does, the value from a_1 can still be passed on to a_2 through the use of rounding, or floor or ceiling functions. Based on this explanation, the list of acceptable d_1 when d_2 is of a certain type is given below:

When category is numeric:

if d_2=decimal or double or float or integer, then accept values from d_1=any numeric type

if d_2=int, then accept values from d_1=float, unsignedShort, byte, unsignedByte

if d_2=long, then accept values from d_1=decimal, float, int, short, unsignedInt, unsignedShort, byte, unsignedByte

if d_2=nonNegativeInteger, then accept values from d_1=positiveInteger

if d_2=nonPositiveInteger, then accept values from d_1=negativeInteger

if d_2=short, then accept values from d_1=byte, unsignedByte

if d_2=unsignedInt, then accept values from d_1=int, short, unsignedShort, byte, unsignedByte

if d_2=unsignedShort, then accept values from d_1=byte, unsignedByte

When category is string:

if d_2=string or normalizedString, then accept values from d_1=any string type

if d_1= not string type, then accept values if the attribute names are semantically similar and model references refer to the same entity

When category is URI:

if d_2=anyURI, then accept values from d_1=string

When category is date/time:

if d_2=gDay, then accept values from d_1=date, dateTime by separating the 'Day' part from them

if d_2=gMonth, then accept values from d_1=date, dateTime by separating the 'Month' part from them

if d_2=gMonthDay, then accept values from d_1=date, dateTime by separating the 'Month-Day' part from them

if d_2=gYear, then accept values from d_1=date, dateTime by separating the 'Year' part from them

if d_2=gYearMonth, then accept values from d_1=date, dateTime by separating the 'Year-Month' part from them

if d_2=date, then accept values from d_1=dateTime by separating the 'Date' part from it

if d_2=time, then accept values from d_1=dateTime by separating the 'Time' part from it

The second part of resolving data representation conflict involves matching the XSD restrictions on the datatypes. Checking is done to ensure that they have the same kind of restrictions and

that Range/Length-of-values (d_2) $\geq$ Range/Length-of-values (d_1). To reiterate, since we do not have access to the data itself, we avoid any situation that leads to mediating the two Web services when it is not possible offline. The data mediation problem is considered irreconcilable if there is a restriction on d_2 while there is none on d_1.

4.1.3 Data Scaling Conflicts

Despite the fact that two attributes under consideration for mediation are semantically similar and have the same datatypes or datatypes that can be mediated, there can be conflicts in the scale of the data. There is more than one way this can occur. A simple example would be when there is one attribute in the output message of WS_1 representing currency in one form (Eg:-USD) and the attribute representing currency in the input message of WS_2 is expecting it in another form (Eg:-Euro). It is the same case when data needs to be represented in different metric systems in different Web services. This type of conflict where the difference in representation/format of data can be resolved by processes like scaling/shifting the data with the help of some context information is called data scaling conflict.

Sometimes, when data representation conflict cannot be resolved between the two services, the reason could be that there exists data scaling conflict that is actually resolvable. Therefore, it should be ensured that the mediation process is discontinued only after both data representation and data scaling conflicts cannot be resolved on the attributes.

The most important part of resolving data scaling conflicts is being able to identify that the conflict exists, from the data available in the inputs to the translator WS. The values of model references for attributes, MR_A, are solely used for this purpose. Here is an example of two services that have data scaling conflicts and how they are resolved.

WS_1: Object {(Length, Width), (Float, Float)}
WS_2: Object {(Length, Width), (Float, Float)}

Suppose the model reference of both attributes in WS_1 points to an URI

"http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#meter"

and the model references of both attributes in WS_2 are

"http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#centimeter"

The *sciUnits* ontology is part of *SWEET* (Semantic Web for Earth and Environmental Terminology) [6] ontologies. The *sciUnits* in particular is useful for resolving data scaling conflicts when programmatically parsed and reasoned efficiently.

Using an ontology reasoner like Pellet [43] or HermiT [47], it can be inferred that 'meter' and 'centimeter' are representing the same kind of data and therefore there is a data scaling conflict present. Furthermore, one value can be converted to another using a definite scaling factor. To summarize, once the data scaling conflict is identified with the help of model references, then the messages can be mapped by incorporating the scaling factors (if available from the model reference) in the transformation rule set.

4.2 Concept Level Conflicts

The resolution of concept level incompatibilities is explained in this section.

4.2.1 Naming Conflicts

The identification and resolution of naming conflict for concepts is done in the same manner as done in naming conflicts for attributes discussed previously. An example for Web services having naming conflicts in concepts is given below.

WS_1: Institution {(Name, Population), (String, Int)}

WS_2: University {(Name, Population), (String, Int)}

Though 'Institution' and 'University' have the same meaning, since they have different labels the two Web services will be rendered as not composable. To avoid this, the concept names are checked to see if they are synonyms of one another with the help of a lexicon, dictionary etc.

4.2.2 SCHEMA ISOMORPHISM CONFLICTS

In the example for Web services having a schema isomorphism conflict as given below, the concepts 'Institution' and 'University' are identified to be semantically similar.
WS_1: Institution {(Name, AptNo, StreetAddr, City, Zip), (String, String, String, String, String)}
WS_2: University {(Name, Address), (String, String)}

However, they have an unequal number of attributes. By common knowledge, we know that the WS_1 attributes 'AptNo', 'StreetAddr', 'City' and 'Zip' together has to be matched to the attribute 'Address'. This could be automated if there was a machine understandable model representing that 'Address' is composed of 'AptNo', 'StreetAddr', 'City', 'State' and 'Zip'. Ontologies are used to represent concepts and can be used for this purpose. Consider the following attributes in WS_2 and their corresponding SAWSDL model references:

(AptNo, http://daml.umbc.edu/ontologies/ittalks/address#aptnumber),
(StreetAddr, http://daml.umbc.edu/ontologies/ittalks/address#street),
(City, http://daml.umbc.edu/ontologies/ittalks/address#city),
(Zip, http://daml.umbc.edu/ontologies/ittalks/address#zip).

For the attribute 'Address' in WS_2, the SAWSDL model reference is

http://daml.umbc.edu/ontologies/ittalks/address#Address

Using a reasoner on the $Address$ ontology [2], it can be found that 'apt_number', 'street', 'city', 'zip' are datatype properties for the class 'Address'. In Fig. 4.1, we show an OWL individual of the 'Address' class. Based on this, it is safe to conclude that the values of attributes 'AptNo', 'StreetAddr', 'City', 'Zip' from WS_1 can be concatenated and assigned to the attribute 'Address' in WS_2. The values are concatenated with comma separations in between them.

The SAWSDL lifting and lowering schema mappings can be used for mapping the attributes too but they have not been used in the work for this thesis. The method is the same if the attributes were associated as object properties instead of datatype properties.

```
<address:Address rdf:ID="my_address">
    <address:city rdf:datatype="&xsd;string">athens</address:city>
    <address:state rdf:datatype="&xsd;string">georgia</address:state>
    <address:zip rdf:datatype="&xsd;string">30602</address:zip>
</address:Address>
```

Figure 4.1: Instance of the 'Address' class from the $Address$ ontology

Suppose WS_1 and WS_2 were swapped with the attributes having the same SAWSDL model references as before:

WS_1: University {(Name, Address), (String, String)}

WS_2: Institution {(Name, AptNo, StreetAddr, City, Zip), (String, String, String, String, String)}

In this case, the value of 'Address' needs to be split based on a delimiter and the substrings have to be assigned to the attributes 'AptNo', 'StreetAddr', 'City', 'Zip'.

Consider the following example for another type of schema isomorphism between two Web services that involve information about people:

WS_1: Person {(Name, ID, Phone, WorkEmail, PersonalEmail), (String, String, String, String, String)}

WS_2: Person {(Name, ID, Phone, Email), (String, String, String, String)}

The attributes, 'Name', 'ID', 'Phone' from WS_1 can be matched and mapped to their namesake attributes in WS_2. We know that the attributes 'WorkEmail' and 'PersonalEmail' (in WS_1) are types of 'Email' (in WS_2). This can be seen as a class-superclass relationship. Again, we need a machine understandable model that represents this relationship and ontologies can be used for this purpose.

The model references for the attributes involved are:

(WorkEmail, http://daml.umbc.edu/ontologies/ittalks/person#workEmail),

(PersonalEmail, http://daml.umbc.edu/ontologies/ittalks/person#personalEmail),

(Email, http://daml.umbc.edu/ontologies/ittalks/person#Email).

Using a reasoner on the $Person$ ontology [10], we would be able to infer that 'workEmail' and 'personalEmail' are subclasses of the 'Email' class. In this case, either one of the values from 'workEmail' or 'personalEmail' in WS_1 can be assigned to 'Email' in WS_2.

In some cases, it is not sufficient for the mediation to be based on the model references alone. Suppose, there was no attribute called 'Email' in WS_2 in the above example and there was another attribute called 'Title' instead. Therefore, WS_2: Person (Name, ID, Phone, Title). In a situation where a wrong model reference has been provided to 'Title',

For e.g., http://daml.umbc.edu/ontologies/ittalks/person#Email,

the mediation will still happen and the value of either 'workEmail' or 'personalEmail' will wrongly be assigned to 'Title' after mediation. Therefore, the attribute names are compared to ensure that they are semantically similar with the help of WordNet [33] or with the help of a similarity measure [26].

If WS_1 and WS_2 are swapped, then the mediation process becomes a little more complicated.
WS_1: Information {(Name, ID, Phone, Email), (String, String, String, String)}
WS_2: Information {(Name, ID, Phone, WorkEmail, PersonalEmail), (String, String, String, String, String)}

'WorkEmail' and 'PersonalEmail are subclasses of 'Email' i.e. they are types of 'Email'. This means that 'Email' can take values of both 'WorkEmail' and 'PersonalEmail'. However, 'WorkEmail' or 'PersonalEmail' cannot simply be assigned with values of 'Email' without confirming the definite type of subclass it belongs to.

4.3 Abstraction Level Incompatibilities

This conflict occurs when semantically similar entities are placed at different levels of abstraction in different schemas. The different types of abstraction level conflicts and how we may resolve them are given below.

4.3.1 Generalization Conflicts

In the example given below, 'Alumnus' is a way of representing 'PhD Alumnus' more generally.
WS_1: PhDAlumnus {(Name, Dept, Email, Website), (String, String, String, anyURI)}
WS_2: Alumnus {(Name, Dept, Email, Website, Type), (String, String, String, anyURI, String)}

Suppose, the SAWSDL model reference of "PhDAlumnus" is

"http://ebiquity.umbc.edu/ontology/person.owl#PhDAlumnus"

and that of "Alumnus" is

"http://ebiquity.umbc.edu/ontology/person.owl#Alumnus".

In the *person.owl* ontology [52], "PhDAlumnus" is a subclass of the class "Alumnus". Using the subclass relationship between the two concept names, the conflict between the two concepts can be resolved. However, this isn't enough to resolve the generalization conflict because we should also try to incorporate in WS_2 that the "type" of "Alumnus" is "PhD". Therefore, generalization conflicts in Web services are resolved if,

1. The concept C_1 can be linked to the concept C_2 through a subclass relationship. (To confirm that the concept names NC_1 and NC_2 are indeed semantically similar, homographic similarity can be measured with the help of a standard similarity measure such as n-gram similarity measure [26]).

2. After mediating all the attributes in WS_1 and WS_2, there is an attribute left in WS_2 called 'Type' that can be assigned with the value of NC_1

If the schema of WS_1 and WS_2 were swapped, the two Web services can still be mediated if the new C_1 is found to be a super-class of the new C_2 and the 'Type' attribute (now in the new WS_1) contains a value which is equal to the value of the new NC_2. Unfortunately, this cannot be implemented with the mediation model defined in this paper because we do not have access to the actual instantiations (in this case the value stored in 'Type') of the attributes inside the translator WS.

4.3.2 AGGREGATION CONFLICTS

An example of aggregation conflict is:
WS_1: UndergraduateStudent {(Name, ID, Dept, Email), (String, Int, String, String)}
WS_2: StudentBody {(Name, ID, Dept, Email), (String, Int, String, String)}

A student body is made of undergraduate students, graduate students, etc. Thus, in the above example, the concept C_1 "UndergraduateStudent" is a division of concept C_2 "StudentBody". To resolve this type of conflict, with the help of a lexicon like WordNet [33] it is first checked if NC_2 is a holonym of NC_1 or if NC_1 is a meronym of NC_2. For additional confirmation, it is checked if the (SAWSDL) model reference MRC_1 is pointing to a datatype or object property of the class, the model reference MRC_2. This is done because datatype properties and object properties are often used to express the 'part-of' relationship in ontologies. If WS_1 and WS_2 were swapped, the mediation will not be possible for aggregation conflicts.

4.3.3 ATTRIBUTE-ENTITY CONFLICTS

Consider the following example again,
WS_1: Resident {(Name, Address, Phone), (String, String, String)}
WS_2: AreaInfo {(Resident, Address, Phone), (String, String, String)}

The concept in WS_1, "Resident" is present as an attribute in WS_2. Suppose, the attributes "Address" and "Phone" can be mediated even if they have a data representation conflict, then the

attributes left to be matched are "Name" in MS_1 and "Resident" in MS_2. It is safe to assume that the attribute "Resident" is looking for a value of an identity or name of sorts. Thus, when an attribute-entity conflict occurs, it can be resolved by:

1. First, mediating the attributes other than the attribute (say A_i) in MS_2 whose name is similar to the concept name in MS_1

2. Now, if there is just one attribute remaining in MS_1 whose name is semantically similar to "Name" or "Identity" or "ID", and if this attribute and A_i can be resolved of other attribute level conflicts as well, then it's instantiation is assigned to A_i

If A_i cannot be matched to an attribute in MS_1, that is semantically similar to "Name" or "Identity" or "ID", or if there are one more attributes still remaining in MS_2 after the match has been made, then the two Web services cannot be mediated unless information from M_h can be used. If WS_1 and WS_2 are interchanged, then the attribute-entity conflict is resolved by,

1. First, mediating the attributes other than the attribute (say A_i) in MS_1 whose name is equal the concept name in MS_2

2. Now, if there is just one attribute remaining in MS_2 whose name is semantically similar to "Name" or "Identity" or "ID", and if this attribute and A_i can be resolved of other attribute level conflicts as well, then it is assigned with the instantiation of A_i

If A_i cannot be matched to an attribute in MS_2, that is semantically similar to "Name" or "Identity" or "ID", or if there are one more attributes still remaining in MS_2 after the match has been made, then the two Web services cannot be mediated. Although, mediation might still be possible by using data from M_h if it proves to be appropriate. The following section describes how these conflicts are programmatically identified and resolved.

4.4 RULE SET

The previous section described in detail, the different types of message-level heterogeneities and how they maybe resolved. The examples given were simple cases portraying situations when one particular conflict occurs between the Web services involved. In practical real world Web services, there may be a combination of conflicts occurring. Any two Web services that require the assistance of the mediator Web service will be examined for the occurrence of all the mentioned conflicts and will be mediated if possible. The order in which the conflicts are looked for and are handled is important in order to guarantee that one conflict does not undermine another leading to partial and incorrect mediation, or concluding that mediation cannot be done in a case where mediation can actually be performed. These pointers have been taken into consideration in this implementation.

We mentioned that the output of the translator Web service in case of successful mediation between two Web services, is an XSLT file. The following passages describe how any basic XSLT file transforms an XML file, thus providing an insight into how the translator WS constructs its output (the XSLT file) and why it constructs it the way it does. Fig. 4.2 gives an example of a simple XSLT file. Let the XML document on which, the above transformation is going to be applied on, be:

<Student>
<Name>Adam</Name>
<Score>65.5</Score>
</Student>

When the execution engine reaches line 15 in the transformation, it creates an element called “Student”. At line 16, an element called “Name” and at line 19, an element called “Grade” is created inside the “Student” element. The value in the “select” attribute of the “apply-template” function gives us the element from the input XML document to be processed. The transformation rule to process this particular element/node can be found if it matches the value assigned to the “match” attribute of an “xsl:template” element.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="xml" indent="yes" omit-xml-declaration="yes" />

  <xsl:template match="/">
            <xsl:element name="Student">
                    <xsl:element name="Name" >
                      <xsl:apply-templates select="//Name"/>
                    </xsl:element>

      <xsl:element name="Grade">
                      <xsl:apply-templates select="//Score"/>
                    </xsl:element>
            </xsl:element>
        </xsl:template>

        <xsl:template match="Name">
             <xsl:value-of select="current()"/>
      </xsl:template>

          <xsl:template match="Score">
        <xsl:value-of select="current() div 2"/>
          </xsl:template>
</xsl:stylesheet>
```

Figure 4.2: Sample XSLT transforming an XML file containing information of a student

In the above example, to assign a value for the element “Name” once it is created for the output document, the XSLT processor tries to process the element “Name” in the input document. At line 25, it finds a template that matches and the transformation rule in provided at line 26. In this particular example, the rule is pretty simple and the processor will just assign the current value of the “Name” element to the output “Name” element. For the “Grade” element, the value will be assigned by transforming the “Score” element of the input document based on the rules provided in the template it matches with. The output of this transformation document on the given input is:

<Student>

<Name>Adam</Name>

<Grade>32.75</Grade>

</Student>

If it is assumed that the given input XML document is the output SOAP message of a WS_1: Student (Name, Score) and, if WS_2: Student (Name, Grade), then the XSLT document in Fig 4 could easily be conceived as the output of the mediator Web service trying to mediate WS_1 and WS_2. For any other two Web services that can be mediated, the transformation rules can be automatically built using the same structure as the XSLT file above. The names of elements and specific rules for transformation change depending on the Web services being used. The values for names of elements come from the concepts and attributes names in the input to the translator WS and the transformation rules are built dynamically by the translator WS based on what conflicts exists between the two Web services in question. To understand this better, consider two Web services,

WS_1: Parameters {(Width_cm, Height_cm), (Float, Float)}

WS_2: Area {(Width_m, Height_m), (Float, Float)}

Suppose name of WS_1 is *ParametersService* and name of WS_2 is *AreaService*, the transformation rules for this set of Web services returned by the translator WS is given in the Fig. 4.3 below:

Comparing the XSLT documents in Fig. 4.2 and Fig. 4.3, we can see that the underlying structure is the same and only the element names and their transformation rules are different and are dependent on the Web services being mediated.

All types of concept level conflicts are handled first, starting with Naming conflicts. If a concept level conflict exists and cannot be mediated, then there is no need for investigating the attribute level conflicts at all. Given below is the sequence in which the various conflicts are checked for.

1. Check for concept level Naming conflicts
2. Check for Generalization conflicts
3. Check for Aggregation conflicts
4. Check for Attribute-Concept conflicts (within that check for 1-3)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="xml" indent="yes" omit-xml-declaration="yes" />

    <xsl:template match="/">
        <xsl:element name="Parameters">
            <xsl:element name="Width_m">
                <xsl:apply-templates select="//Width_cm" />
            </xsl:element>

            <xsl:element name="Height_m">
                <xsl:apply-templates select="//Height_cm" />
            </xsl:element>
        </xsl:element>
    </xsl:template>

    <xsl:template match="Width_cm">
        <xsl:value-of select="current()*0.01" />
    </xsl:template>

    <xsl:template match="Height_cm">
        <xsl:value-of select="current()*0.01" />
    </xsl:template>
</xsl:stylesheet>
```

Figure 4.3: Transformation rules for mediating the *ParametersService* and the *AreaService*

5. If any of the conflicts in 1-4 exists and can be resolved, and if the number of attributes in MS_1 and MS_2 are unequal, then check for Schema Isomorphism conflicts.

6. If the number of attributes in MS_1 and MS_2 are equal after steps 1-4 or 1-5 (after subtracting the number of attributes that were mapped in step 5 through Schema Isomorphism conflict check), then examine all the remaining attributes for attribute level Naming conflicts. Align the semantically similar attributes of both Web services in the same order of sequence in the variable arrays holding them.

7. If any of the conflicts in 1-4 exists and cannot be resolved or if any naming conflict among the attributes cannot be resolved, then we quit the mediation process and conclude that the two Web services cannot be mediated.

8. If step 4 was completed successfully, then inspect the attributes for Data Representation conflicts

9. Whether Data representation conflicts in the attributes is present or not, check if Data Scaling conflicts are present

10. If the attribute level conflicts are also resolved, then the two Web services have been mediated successfully.

If mediation was successful, then the transformation rules in XSLT are returned as the output of the translator Web service.

CHAPTER 5

EVALUATION

We have described the implementation of the translator Web services in the previous chapters of this thesis. To test the working of the translator Web service, a repository of test Web services was created. The Web services in the repository were paired with one another. Between these pairs of Web services exist one or more types of message-level heterogeneities. The translator Web service mediates these conflicted Web services. Apart from this, we have also tested our translator Web service on few real world Web services. The various case studies and how the mediator Web service operates in each of them, is given below.

5.1 SYNTHETIC TEST CASES

In the case studies discussed below, the Web services are specified in the following format:

$$WS_n\text{: } [N_C n(MR_C n)\ \{N_{Ani}(N_{Dni},\ R_{Dni},\ MR_{Ani})\}]$$

Where,

n = 1 or 2 indicating it is either WS_1 or WS_2 NC = Concept Name MRC = Model reference for Concept NA = Attribute Name ND = Attribute Datatype RD = Restriction on Attribute's Datatype MRA = Model reference for Attribute, and *i* = 1 to *m*, where *m* is number of attributes present

The values for model references and restrictions exist or not depending on their presence in the Web service's specification. The other details are present in the case of all the Web services.

Case 1:

WS_1: [Institution {Name (String), People (Integer), Address (String)}]

WS_2: [University {Name (String), Population (Integer), Address (String)}]

From observing the Web services WS_1 and WS_2, we know that only concept level naming conflicts and attribute level naming conflicts (on the attributes 'People' and 'Population') are present. These conflicts are resolved with the help of WordNet [33] and now the Web services can successfully participate in a Web service composition.

Case 2:

Along with naming conflicts on the concept and an attribute (between attributes 'People' and 'Population'), data representation conflict (on datatypes of attributes 'People' and 'Population') and schema isomorphism (between attributes 'Address' in WS_1 and attributes 'City' and 'State' in WS_2) conflict also occur in this case. The naming conflicts are resolved with the help of WordNet [33].

WS_1: [Institution {Name (String), People (Int),
Address (String,, http://daml.umbc.edu/ontologies/ittalks/address#Address)}]
WS_2: [University {Name (String), Population (Long),
City (String,, http://daml.umbc.edu/ontologies/ittalks/address#city),
State (String,, http://daml.umbc.edu/ontologies/ittalks/address#state)}]

The datatype of 'People' is 'Int' and that of 'Population' is 'Long'. Since the range of an 'Int' datatype falls within the range of 'Long' datatype, there won't be a problem in assigning the value of the 'People' attribute to the 'Population' attribute. From Chapter 4.2.2, we know that a schema isomorphism conflict may be resolved using the model references of the attributes involved.

Case 3:

WS_1: [Dimensions {
Height_Object(Float,, http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#centimeter),
Width_Object (Float,, http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#centimeter)}]
WS_2: [Area {
Height_Object(Float,, http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#meter),
Width_Object (Float,, http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#meter)}]

Data scaling conflict exists between these two Web services. In the sciUnits.owl ontology, as shown in Fig. 5.1 'centimeter' is defined as an instance of the class 'UnitDerivedByScaling'.

```
<units:UnitDerivedByScaling rdf:ID="centimeter">
    <units:derivedFromUnit rdf:resource="#meter"/>
    <units:hasPrefix rdf:resource="#centi"/>
</units:UnitDerivedByScaling>
```

Figure 5.1: Excerpt from the sciUnits.owl showing the 'centimeter' class

Using a reasoner, we obtain the value of the "derivedFromUnit" property which is "meter". Since the base unit for all measurement units are "meter" in this ontology, we can convert any measurement to "meter" once we know the scaling factor. This can be obtained by traversing to the instance "centi" shown in Fig. 5.2, and getting the value of the "hasPrefix" property for "centimeter".

```
<units:Prefix rdf:ID="centi">
    <units:hasSymbol rdf:datatype="&xsd;string">c</units:hasSymbol>
    <units:hasValue rdf:datatype="&xsd;double">0.01</units:hasValue>
</units:Prefix>
```

Figure 5.2: Excerpt from the sciUnits.owl showing the instance, 'centi'

The property "hasValue" holds the scaling value and this is factored into the transformation rules for converting the values of the attributes in WS_1 from centimeters to meters for those in WS_2. Since "meter" is the base unit, if the value of the scaling factor$>$1, then a division function is used as the operator during conversion and multiplication is used otherwise.

Case 4:

WS_1: [Dimensions {

Height_Object (Float,, http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#kilometer),

Width_Object (Float,, http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#kilometer)}]

WS_2: [Area {

Height_Object (Float,, http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#nanometer),

Width_Object (Float,, http://sweet.jpl.nasa.gov/2.0/sciUnits.owl#nanometer)}]

The two Web services in this example have a data scaling conflict similar to that in the previous case. However, a difference arises when it comes to the transformation rules built for converting the data in one unit of measurement to another. Since “meter” is the base unit, it is easy to convert a value from any other unit of measurement to “meter” and vice-versa. Suppose neither of the two attributes concerned have data represented in units that is a base unit, then the mediation is performed by first converting the value of the attribute in WS_1 to “meter”, and then converting that value to the actual unit required by the attribute in WS_2. For this example, the sequence of conversions would be “kilometer” to “meter” and then “meter” to “nanometer”.

Case 5:

WS_1: [PhDAlumnus (http://ebiquity.umbc.edu/ontology/person.owl#PhDAlumnus) {
Name (String), Dept (String),
WorkEmail (String,, http://daml.umbc.edu/ontologies/ittalks/person#workEmail),
PersonalEmail (String,, http://daml.umbc.edu/ontologies/ittalks/person#personalEmail)}]

WS_2: [Alumnus (http://ebiquity.umbc.edu/ontology/person.owl#Alumnus){
Name (String), Dept (String), Type (String),
Email (String,, http://daml.umbc.edu/ontologies/ittalks/person#Email)}]

When the mediator Web service is checking these two web services for potential conflicts, it would discover that generalization conflicts (between PhDAlumnus and Alumnus) and schema isomorphism conflict (between attributes ’WorkEmail’ and ’PersonalEmail’ in WS_1 and ’Email’ in WS_2) are existing. Based on the resolution techniques described in Chapter 4.3.1, the mediator Web service will know that this generalization conflict is resolvable because of the subclass-class relationship between the model references of the concept C_1 and the concept C_2. The value of NC_1 = “PhDAlumnus” will be assigned to the ’Type’ attribute in WS_2. Using a reasoner on the *’person’* ontology, the translator Web service will be able to infer that ’workEmail’ and ’personalEmail’ are subclasses of the ’Email’ class, and will assign either one of the values from ’workEmail’ or ’personalEmail’ in WS_1 to ’Email’ in WS_2.

Case 6:

WS_1: [UndergraduateStudent (http://ebiquity.umbc.edu/ontology/person.owl#BSStudent)
{FirstName (String,, "http://ebiquity.umbc.edu/ontology/person.owl#firstName"),
LastName (String,, "http://ebiquity.umbc.edu/ontology/person.owl#lastName"),
Dept (String), Email (String), Website (anyURI)}]

WS_2: [StudentBody (http://ebiquity.umbc.edu/ontology/person.owl#Student) {
Name (String,, "http://ebiquity.umbc.edu/ontology/person.owl#Person"),
Dept (String), Email (String), Website (String)}]

Aggregation conflict exists between the concepts 'UndergraduateStudent' and 'StudentBody' and this is resolvable. Schema isomorphism conflict occurs between attributes 'FirstName' and 'LastName' in WS_1 and the attribute 'Name' in WS_2. Using the model references of these attributes, this conflict can be resolved using the techniques provided in Chapter 4.2.2. Data representation conflict (between the datatype of attributes 'Website' and 'Website')is present and based on Chapter 4.1.2, the mediator Web service will derive that the value of an attribute whose datatype is 'anyURI' can be assigned to an attribute whose datatype is 'String'.

Case 7:

WS_1: [UndergraduateStudent (http://ebiquity.umbc.edu/ontology/person.owl#BSStudent)
{Name (String,, "http://ebiquity.umbc.edu/ontology/person.owl#Person"),
Dept (String), Email (String), Website (String)}]

WS_2: [StudentBody (http://ebiquity.umbc.edu/ontology/person.owl#Student) {
Name (String,, "http://ebiquity.umbc.edu/ontology/person.owl#Person"),
Dept (String), Email (String), Website (String)}]

The WS_1 and WS_2 from case 6 are swapped for this case. A student body is comprised of undergraduate students, graduate students, etc. Therefore, all undergraduate students are students but not all students are undergraduate students. Based on this, the translator Web service declares that these two Web services cannot be mediated.

Case 8:

WS_1: [Resident {Name (String), Address (String), Phone (String)}]

WS_2: [AreaInfo {Resident (String), Address (String), Phone (String)}]

The concept 'Resident' in WS_1 is an attribute in WS_2. The translator Web service will identify this as a case of an attribute-entity conflict and will try to mediate the two Web services. This particular example of attribute-entity conflict is resolvable. After matching all the other attributes, we will be left with 'Name' in WS_1 and 'Resident' in WS_2. Using the techniques described in Chapter 4.3.3, the value of the 'Name' attribute is assigned to 'Resident'

Case 9:

WS_1: [Institution {Name (String), Population (Integer), Address (String)}]

WS_2: [Community {Name (String), Population (Integer), Address (String)}]

The concept names of the two Web services are different and their attributes have no conflicts between each other. This makes it seem like there exists a concept level naming conflicts between WS_1 and WS_2 which can be resolved using a lexicon like WordNet [33]. However, with the help of WordNet [33], it is discovered that these two concept names are not synonyms of each other. Hence, these two Web services cannot be mediated.

5.2 Real world test cases

The translator Web service mediates the created pairs of synthetic Web services successfully. As an additional evaluation step and for impartial evaluation, we have tested our translator Web services on four of the real world Web services mentioned in the evaluation section in [36]. These real world Web services along with the in-house *'investment assistant'* Web service built by the Nagarajan et al [36] was recreated with some modifications to the WSDLs, and stored locally in our repository. The concept names N_C of all these Web services are assigned with the same value to prevent the declaration of the Web services as irreconcilable based on just the concept-level naming conflicts. The *'investment assistant'* Web service is used as WS_2 for all the test

cases discussed in this section. The input message schema for the *'investment assistant'* [36] is given in Fig. 5.3. The attributes are annotated using the *LSDIS Finance* ontology whose URI is "http://lsdis.cs.uga.edu/projects/meteor-s/wsdl-s/ontologies/LSDIS_Finance.owl".

```
<xsd:element name="Investment" type="tns:InvestmentInfo"/>

<xsd:complexType name="InvestmentInfo">
      <xsd:sequence>
            <xsd:element name="current_price" type="xsd:double"
                  sawsdl:modelReference="Ontology#price"></xsd:element>
            <xsd:element name="change" type="xsd:double"
                  sawsdl:modelReference= "Ontology#changeInValue">
            </xsd:element>

            <xsd:element name="volume" type="xsd:int"
                  sawsdl:modelReference="Ontology#volume"></xsd:element>
            <xsd:element name="bid_quantity" type="xsd:int"
                  sawsdl:modelReference="Ontology#quantity"></xsd:element>
            <xsd:element name="investAmount" type="xsd:double"
                  sawsdl:modelReference="Ontology#transactionAmount"></xsd:element>
            </xsd:sequence>
</xsd:complexType>

<wsdl:message name="investmentHelperRequest">
      <wsdl:part element="tns:Investment" name="part1"/>
</wsdl:message>
```

Figure 5.3: Excerpt from the WSDL of $investmenthelper$ showing its input message schema. The $Ontology$ refers to the LSDIS Finance ontology

We will now discuss the different case studies based on the real-world Web services and how the translator operates in those cases. In all the examples, $Ontology$ refers to the LSDIS Finance ontology.

Case 10:

WS_1: [StockQuote {

CompanyName (string,, Ontology#shareOf),

StockTicker(string,, Ontology#stockSymbol),

StockQuote (string,, Ontology#StockQuote),

CurrentPrice (int,, Ontology#price),

Change (string,, Ontology#changeInValue),

OpenPrice (int,, Ontology#open),

DayHighPrice (string,, Ontology#high),

DayLowPrice (int,, Ontology#low),

Volume (string), MarketCap (string),

YearRange (double,,Ontology#_52WeekChange)}]

There is a data representation conflict present since all the attributes of WS_1 are of 'String' type whereas the datatypes of the attributes in WS_2 are numeric. Based on the resolution techniques in Chapter 4.1 along with comparison of the model references of the attributes, this conflict is resolved. By observing the model references of the attributes 'YearRange' in WS_1 and 'year_high' in WS_2, we know that schema isomorphism conflict occurs. The translator Web service resolves this using the technique presented in Chapter 4.2. We assume that the data for the attributes *bid_quantity* and $investAmount$ are obtained from M_h. The URI of the original Web service is *"http://www.webservicex.net/stockquote.asmx?wsdl"*.

Case 11:

WS_1: [StockQuote {GetStocksXMLSchemaResult (string,,Ontology#StockQuote)}]

All the attributes in WS_2 cannot be matched to attributes in WS_1. Therefore, this case cannot be mediated. The URI of this Web service is *"http://gama-system.com/webservices/stockquotes.asmx?wsdl"*.

Case 12:

WS_1: [StockQuote {

StockSymbol(string,, Ontology#stockSymbol),

CurrentPrice (decimal,, Ontology#price),

LastTradeDateTime (dateTime),

StockChange (decimal,, Ontology#changeInValue),

OpenAmount (decimal,, Ontology#open),

DayHigh (decimal,, Ontology#high),

DayLow (decimal,, Ontology#low),

StockVolume (int), MarketCap (string),

PrevCls (int), ChangePercent(string), FiftyTwoWeekRange(string,, Ontology#_52WeekChange),

EarnPerShare (string,, Ontology#earning_per_share),

PE (int), CompanyName (string,, Ontology#shareOf),

QuoteError (boolean)}]

Data representation conflict occurs between the attributes of WS_1 and WS_2. However, they are all resolvable using the techniques described in Chapter 4. The naming conflicts are resolved using linguistic aids. A schema isomorphism conflict exists between the attribute 'FiftyTwoWeekRange' in WS_1 and the attribute 'year_high' in WS_2 which is also resolved using the methods explained in Chapter 4.2. We assume that the data for the attributes *bid_quantity* and $investAmount$ are obtained from M_h. The URI for this Web service is *"http://ws.cdyne.com/delayedstockquote/delayedstockquote.as*

Case 13:

WS_1: [StockQuote {

Symbol(string,, Ontology#stockSymbol),

CompanyName (string,, Ontology#shareOf),

Date (string), Time (string),

Open (double,, Ontology#open),

High (decimal,, Ontology#high),

Low (decimal,, Ontology#low),

Current_Price (decimal,, Ontology#price),

Volume (int), Change (double,, Ontology#changeInValue),

PercentChange (double), Previous_Close (string),

Bid(string), Bid_Size(string,, Ontology#quantity),

Ask(string), Ask_Size(string),

Low_52_Weeks (string,, Ontology#fifty_two_week_low),

High_52_Weeks (string,,Ontology#fifty_two_week_high)}]

There are data representation conflicts present between some of the attributes in WS_1 and WS_2. They are all resolvable based on the explanation provided in Chapter 4.1. Attribute-level naming

conflicts are resolved with the help of a similarity measure [26]. There exists schema isomorphism conflict between Low_52_Weeks and High_52_Weeks of WS_1 and $year\_high$ of WS_2 which is also resolved like in the previous cases, using the techniques in Chapter 4.2. The URI of this Web service is *"http://xignite.com/xquotes.asmx?wsdl"*.

5.3 Discussion

Using all the information available to it, the conditions under which any message-level conflict can be resolved using the translator Web service was described in Chapter 4. So, if those conditions prevail when the translator Web service is trying to mediate a pair of Web services, then it should indeed mediate these two Web services. During mediation, the output message of WS_1 is converted to the format of the input message of WS_2. This means that the converted message can be used to invoke WS_2. Therefore, the receipt of the output message from WS_2 can be used to evaluate the performance of the translator Web service.

For the cases discussed in Chapter 5.1, the translator Web service successfully mediates the two Web services involved in every case where mediation is possible. The Web services in cases 1, 2, 5, 6 and 8 of Chapter 5.1 provide simple functionality and the input and output message schemas of a Web service are made to exactly match. The data in the output message of WS_1, input and output messages of WS_2 are compared and found to be the same. In cases 3 and 4 that exemplify data scaling conflicts, the WS_2 returns the area calculated as the product of the two attribute values. Based on the output message data from WS_1 and the scaling it needs, it is found that the WS_2 returns the correct area value in both cases.

If the two Web services that require mediation can successfully be mediated, then the translator Web service returns the transformation rules as a string which is later stored as an XSLT document. If the Web services cannot be mediated, then the rule set should be empty. Since a Web service cannot return a null value as the output, the translator Web service is set to return the string "Cannot be mediated". In cases 7 and 9 where the mediation cannot be performed, this value was returned.

CHAPTER 6

CONCLUSION AND FUTURE WORK

With the resolution of message-level heterogeneities between Web services, the number of functionalities delivered through Web service compositions will increase significantly. The data mediation problem model facilitates transforming at the element level instead of having to match the schemas through complicated schema matching processes. The implemented translator Web service is able to successfully mediate resolvable conflicts between any two Web services based on what information is available. The suggested improvements include modifying the data mediation model and the implementation to overcome the problems discussed below.

The mediator WS is unable to mediate Web services in cases where the mediation is dependent on the data rather than just the message schema. For example, the current resolution of data representation conflicts is solely based on the allowed sizes of the datatypes involved. Due to this reason, many situations that appear to be logically resolvable office, are declared otherwise by the translator Web service.

For every case study, individual compositions (of WS_1, translator WS and WS_2) had to be created because, in the implementation with Java, attention have to be paid to low level details such as explicit data conversion, preparing the messages for invocation of the individual Web services, error handling etc. Manually creating the composition for every pair of Web services requiring mediation weakens the purpose of a translator Web service that can create mappings dynamically. A vain attempt was made, to create just one composition for any pair of Web services, using the Java Reflections API. The XSLT transformation returns the transformed message as an XML structure and while preparing the message for invocation of WS_2, the XML is parsed to obtain the values of the attributes. All the data obtained after parsing are treated as 'String' by Java, and

they have to be manually converted to the appropriate datatype of the attribute that they are going to be assigned to. This prevented the creation of a universal composition using Java Reflections. With some improvements and when integrated into a Web service composition framework, this approach will prove to be a superior automatic mediator system.

BIBLIOGRAPHY

[1] Activebpel. http://www.activevos.com/community-open-source.php.

[2] Address ontology. http://daml.umbc.edu/ontologies/ittalks/address.

[3] Apache axiom. http://ws.apache.org/commons/axiom/.

[4] Bea weblogic workshop. http://www.bea.com/products/weblogic/workshop/ in-dex.shtml.

[5] Business process execution language for web services (bpel), version 1.1. http://www.ibm.com/developerworks/library/ws-bpel.

[6] Guide to sweet ontologies. http://sweet.jpl.nasa.gov/.

[7] Jsr 173. streaming api for xml. http://www.jcp.org/en/jsr/detail?id=173.

[8] Microsoft biztalk server 2004: Biztalk mapper. http://msdn.microsoft.com/library/en-us/introduction/htm/ebiz intro story jgtg.asp.

[9] Oracle bpel process manager. http://www.oracle.com/technology/products/ias/bpel/index.html.

[10] Person ontology. http://daml.umbc.edu/ontologies/ittalks/person.

[11] Super inegrated projects. http://www.ip-super.org.

[12] Uddi working group: Universal description, discovery, and integration of business for the web. http://www.uddi.org.

[13] Web service choreography description language (wscdl) 1.0. http://www.w3.org/TR/ws-cdl-10/.

[14] Web service choreography interface (wsci) 1.0. http://www.w3.org/TR/2002/NOTEwsci-20020808.

[15] Xsl transformations (xslt) version 1.0. http://www.w3.org/TR/xslt.

[16] R. Aggarwal, K. Verma, J. Miller, and J. Milnor. Dynamic web service composition in meteor-s. In *Technical report*, LSDIS Lab, Univeristy of Georgia, Athens.

[17] D. Box, D. Ehnebuska, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (soap) 1.1. In *Technical report*, W3C.

[18] L. Cabral and J. Domingue. Mediation of semantic web services in irs-iii. In *Proceedings of the Joint 1st International Workshop on Mediation in Semantic Web Services (MEDIATE) and 3rd International Conference on Service Oriented Computing (ICSOC)*, 2005.

[19] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M. Shan. Adaptive and dynamic service composition in eflow. In *Advanced Information Systems Engineering*, pages 13–31. Springer, 2000.

[20] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1, w3c note. In *Technical report*, W3C.

[21] P. Doshi, R. Goodwin, R. Akkiraju, and K. Verma. Dynamic workflow composition: Using markov decision processes. In *International Journal of Web Service Research*, pages 1–17, 2005.

[22] B. et al. Web service modeling ontology (wsmo). In *W3C Member Submission*, 2005.

[23] K. Fujii and T. Suda. Semantics-based dynamic web service composition. In *International Journal of Cooperative Information Systems*, pages 293–324, 2006.

[24] A. Gao, D. Yang, S. Tang, and M. Zhang. Web service composition using markov decision processes. In *Proceedings of the WAIM 2005*, pages 308–319, 2005.

[25] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler. Wsmx - a semantic service-oriented architecture. In *International Conference on Web Services (ICWS 2005)*, 2005.

[26] G. Kondrak. N-gram similarity and distance. In *In Proceedings of the Twelfth International Conference on String Processing and Information Retrieval*, pages 115–126, 2005.

[27] J. Kopeck, T. Vitvar, C. Bournez, and J. Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. In *IEEE Internet Computing*, pages 60–67, 2007.

[28] H. Lausen, J. Bruijn, A. Polleres, and D. Fensel. Wsml - a language framework for semantic web services. In *W3C Workshop on Rule Languages for Interoperability*, 2005.

[29] F. L'ecu'e, E. Silva, and L. F. Pires. A framework for dynamic web services composition. In *Emerging Web Services Technology II*, pages 59–75, 2007.

[30] P. Leitner, A. Michlmayr, and S. Dustdar. Towards flexible interface mediation for dynamic service invocations. In *Proceedings of the 3rdWorkshop on Emerging Web Services Technology (WEWST08)*, 2008.

[31] C. Li and T. W. Ling. Owl-based semantic conflicts detection and resolution for data interoperability. In *ER (Workshops)*, pages 266–277, 2004.

[32] X. Li, S. Madnick, H. Zhu, and Y. Fan. Reconciling semantic heterogeneity in web services composition. In *The 30th International Conference on Information Systems (ICIS 2009)*, 2009.

[33] G. A. Miller. Wordnet: A lexical database for english. In *Communications of the ACM*, pages 39–41, 1995.

[34] M. Mrissa, C. Ghedira, D. Benslimane, and Z. Maamar. A context model for semantic mediation in web services composition. In *Proceedings of the 25th International Conference on Conceptual Modeling*, Lecture Notes in Computer Science, pages 12–35. Springer, 2006.

[35] M. Mrissa, C. Ghedira, D. Benslimane, and Z. Maamar. A context-based mediation approach to compose semantic web services. In *ACM Transactions on Internet Technology*, 2007.

[36] M. Nagarajan, K. Verma, A. P. Sheth, and J. A. Miller. Ontology driven data mediation in web services. In *Journal of Web Services Research*, 2007.

[37] J. Nitzsche and B. Norton. Ontology-based data mediation in bpel (for semantic web services). In *D. Ardagna, M. Mecella, and J. Yang, editors, Business Process Management Workshops*, Lecture Notes in Business Information Processing, pages 523–534. Springer, 2008.

[38] J. Nitzsche, T. van Lessen, D. Karastoyanova, and F. Leymann. Bpel for semantic web services (bpel4sws). In *On the Move to Meaningful Internet Systems, OTM 2007 Workshops*, Lecture Notes in Business Information Processing. Springer Verlag, 2007.

[39] J. Nitzsche, T. van Lessen, D. Karastoyanova, and F. Leymann. Wsmo/x in the context of business processes: Improvement recommendations. In *International Journal of Web Information Systems*, 2007.

[40] N. Onose and J. Simeon. Xquery at your web service. In *World Wide Web Conference*, 2004.

[41] S. D. P. Leitner, F. Rosenberg. Daios: Efficient dynamic web service invocation. In *IEEE Internet Computing*, pages 72–80, 2009.

[42] M. P. Papazoglou and D. Georgakopoulos. Serive-oriented computing. In *Communcications of the ACM*, pages 25–65, 2003.

[43] B. Parsia and E. Sirin. Pellet: An owl dl reasoner. In *International Semantic Web Conference (ISWC)*, 2004.

[44] P. Pires, M. Benevides, and M. Mattoso. Building reliable web services compositions. In *Web, Web-Services, and Database Systems*, pages 59–72, 2002.

[45] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. In *VLDB Journal*, 2001.

[46] A. F. S. R. Ponnekanti. Sword: A developer toolkit for web service composition. In *11th World Wide Web Conference (Engineering Track)*, 2002.

[47] R. Shearer, B. Motik, and I. Horrocks. Hermit: a highly-efficient owl reasoner. In *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008)*, 2006.

[48] Q. Z. Sheng, B. Benatallah, M. Dumas, and E. Mak. Self-serv: a platform for rapid composition of web services in a peer-to-peer environment. In *Proceedings of the 28th VLDB Conference*, 2002.

[49] E. Silva, N. F. Pires, and M. van Sinderen. Supporting dynamic service composition at runtime based on end-user requirements. In *Workshop at the International Conference on Service Oriented Computing (ICSOC)*, pages 22–27, 2009.

[50] B. Spencer and S. Liu. Inferring data transformation rules to integrate semantic web services. In *Proceedings of the International Semantic Web Conference*, Lecture Notes in Computer Science, pages 456–470. Springer, 2004.

[51] H. Sun, X. Wang, B. Zhou, and P. Zou. Research and implementation of dynamic web services composition. In *Advanced Parellel Processing Technologies*, Lecture Notes in Computer Science. Springer.

[52] D. Wu, E. Sirin, J. Hendler, D. Nau, and B. Parsia. Automatic web services composition using shop2. In *Workshop on Planning for Web Services*, 2003.