Detecting Semantic Matching In Service Oriented System Integration

Dario Saveliovsky

A Thesis in

The Department of Computer Science

and Software Engineering

Presented in Partial Fulfillment of the Requirements for the Degree of Master of Science (Computer Science)

at Concordia University

Montreal, Quebec, Canada

September 2012

©Dario Saveliovsky, 2012

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Dario Saveliovsky

Entitled: Detecting Semantic Matching In Service Oriented System Integration

and submitted in partial fulfilment of the requirements for the degree of

Master in Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final Examining Committee:

_____ Chair

Dr. D. Goswami

_____ Examiner

Dr. R. Witte

_____ Examiner

Dr. V. Haarslev

_____ Supervisor

Dr. Y. Yan

Approved by _____

Chair of Department or Graduate Program Director

_____

Dr. Robin A. L. Drew, Dean

Faculty of Engineering and Computer Science

Date _____

Abstract

Detecting Semantic Matching In Service Oriented System Integration

Dario Saveliovsky

Service Oriented Architecture is being adopted by an increasing number of businesses in order to make their software available through the network, resulting in a considerable growth in the number of available Web services. The need for an automated way for service integration becomes then a pressing issue. This thesis looks at the major obstacles faced when trying to achieve this goal, including the problem of finding element mappings across service interfaces. For this, a method is proposed which employs schema matching techniques extended for the specifics of Web service definitions. A proof of concept service integration assistant is presented, and tested on several case studies. These include the collaboration between mainstream Enterprise Resource Planning and Product Lifecycle Management software systems —a real-world scenario that illustrates the relevance of this work.

Acknowledgements

I take this opportunity to thank my supervisor, Dr. Yuhong Yan, who has provided me with the necessary guidance to complete this work.

My special thanks to Dr. Volker Haarslev, Dr. Rene Witte, and Dr. Dhrubajyoti Goswami for taking the time to review my thesis.

I want to also thank the people of Mecanica Solutions for providing the necessary information to define a real-world case study for this study.

I would also like to thank the Department of Computer Science and Software Engineering for everything I have learnt during my time in this program.

Finally, my deepest gratitude goes to my family and friends for always being there to give me their huge support and valuable advice while I worked on this thesis.

Contents

List of Figures **vii**

List of Tables **x**

Introduction **1**

1 Introduction **1**

2 Background **5**

2.1 WSDL and RESTful Web Services 6

2.1.1 The WSDL Format 6

2.2 The System Integration Problem 11

2.2.1 Approaches to Service Integration 12

2.2.2 Web Service Composition 15

2.2.3 Development of a Service Adaptor 16

2.3 Schema Trees 18

2.4 Stop Words 20

2.5 String Similarity Metrics 21

2.6 Schema Matching 22

2.6.1 Challenges in Schema Matching 22

2.6.2 Approaches to Schema Matching 24

2.6.3 The XPrüM System 25

2.7 The Case of Mecanica Solutions 360 Enterprise 28

3 Automatic Service Integration 32

3.1 Presentation of the Problem 32

3.2 Challenges 37

3.3 Techniques for Service Parameter Matching 42

3.3.1 Preparation of the Data 42

3.3.2 Computing Linguistic Similarities 45

3.3.3 Comparing Schema Elements 46

3.3.4 Obtaining the Matching Pairs 47

3.4 Evaluation Criteria 49

3.5 Data Collection 50

3.6 Support Tool 52

4 Tool Design and Experiments 53

4.1 The Service Integration Assistant 53

4.1.1 Building of Schema Trees 53

4.1.2 Removal of Stop Words 54

4.1.3 Node Filtering and Addition of Synonyms 55

4.1.4 Computing Similarity and Obtaining Matching Pairs 59

4.2 Experiments 60

4.2.1 Data Sets 61

4.2.2 Effect of Combining Parent Node Names 62

4.2.3 Effect of Two Threshold Selection Method 63

4.2.4 Effect of Auxiliary Data 64

4.2.5 Test Environment and Efficiency 67

4.2.6 Comparison with COMA++ 68

4.3 Concluding Remarks 70

5 Conclusions **71**

List of Figures

2.1 Abstract portion of a WSDL document 8

2.2 Concrete portion of a WSDL document 8

2.3 Abstract definitions from the GeoIPService WSDL 10

2.4 Java class generated from the GeoIP XML type 10

2.5 Java interface generated from the GeoIPServiceSoap portType 11

2.6 Invocation of GeoIPService using the code generated by JAX-WS . . 11

2.7 Sample XML schema 19

2.8 Sample schema tree 20

2.9 Semantically equivalent XML structures 23

2.10 Calculating similarity between node names 26

2.11 Schema tree with post order numbers 27

3.1 Excerpt from SAP's time sheet WSDL 34

3.2 Excerpt from 360e's time sheet WSDL 35

3.3 Services p and c are integrated into process ip 37

3.4 Sections of the SAP and 360e schema trees 40

3.5 Sample schema tree 45

3.6 Building linguistic similarity matrix 47

3.7 Algorithm to compute matchings 49

4.1 Schema trees on the service integration assistant 54

4.2 Suggested stop words on the service integration assistant 56

4.3 City services schema tree 56
4.4 A university course schema tree 58
4.5 A different university course schema tree 58
4.6 Lists of exclusive terms sorted by the frequency in which they appear 59
4.7 Service integration assistant prompts for threshold values 60

List of Tables

4.1 Data sets 62
4.2 Results from testing the effect of combining parent node names . . . 63
4.3 Results from comparing two matching pairs selection methods 64
4.4 Experiment results for parts suppliers services 65
4.5 Experiment results for academic bibliographies services 66
4.6 Experiment results for parts orders services 66
4.7 Experiment results for SAP and 360e timesheet services 67
4.8 Execution time by data set 68
4.9 Comparing results from our technique and COMA++ 69

Chapter 1

Introduction

Service Oriented Architecture (SOA) is a software development paradigm that is based on the design of services which are published and provided over a network through a set of common interfaces [34]. These services can be described in machine-readable formats based on open standards, which allows for the possibility of services to be automatically integrated. This is a major breakthrough from previous technologies, where integration of multiple information systems was a necessarily labour-intensive task with developers having to write adaptors manually. Therefore, as the SOA paradigm is adopted by more and more software systems, it becomes more relevant to find an efficient way for services to be integrated seamlessly.

Two major technologies have emerged for the development of Web services — Representational State Transfer (REST) [39] services and services based on the Web Service Description Language (WSDL) [11]. The former is a light-weight architecture, with no standard support for the specification of service interfaces. On the other hand, WSDL is an open standard for describing Web services in an XML format. This allows for different services to be automatically integrated, using a computer program to inspect their interfaces and adapt parameters to the specific requirements of a service. However, services that have been developed by different parties are likely to

use dissimilar names and data structures to describe the same concepts, complicating the automation of the task. This is called an interface level incompatibility. Moreover, the number of operations exposed by each service and the order in which they should be invoked also depend on arbitrary design decisions taken by the developers — causing control flow level incompatibilities [34].

Some of the proposed solutions for the automatic integration of services (such as [46, 44, 4]) suggest the inclusion of metadata in service definitions in order to provide a semantic description of the services. This kind of information about a service's data types, their relationships, and semantic constraints is added by extending current Web service standards to include links to predefined ontologies. However, for this technique to work, a standard would need to be adopted by the industry. As this has not happened yet, the majority of Web services in existence today do not include semantic information in their definitions [46].

Other approaches try to work around the lack of semantic information in service definitions by analyzing their contents and finding recurring patterns across services. An important issue that arises in this situation is the identification of semantically equivalent concepts across services —i.e., a mapping between elements representing the same concepts over multiple service definitions. Among the research literature, we find some works that try tackling differences in the control flow and interfaces of services, such as [27] and [42]. The former proposes a method for identifying the equivalent elements across services by relying on generic schema matching techniques. The latter paper, on the other hand, expects that the matching of these equivalent elements be previously made as a prerequisite to using their method. In this thesis, we will develop a method for the automated identification of semantically equivalent elements across services. This method distinguishes itself from the ones in other works in the fact that it targets the specific particularities of the WSDL format, making it a more appropriate strategy for Web service integration.

As a real-world example of service integration, we will introduce the case of our industry partner Mecanica Solutions [23]. Their 360 Enterprise (360e) Product Lifecycle Management (PLM) software integrates a suite of solutions to optimize product development throughout an enterprise. A typical customer of Mecanica's product will be a company that also uses an Enterprise Resource Planning (ERP) system for managing other aspects of their business such as accounting or human resources. In such cases, some of the information handled by both pieces of software will coincide. Therefore, the ability to share data between these applications becomes important. A traditional way of accomplishing this would be for 360e developers to write specific code to integrate each type of common data for every known ERP system. However, since 360e and the major ERP systems provide Web service APIs, there is a possibility to integrate the two systems automatically. In this thesis, we will discuss this as a case study of the automatic service integration problem.

The main goal of this thesis is then to present a strategy for the semi automated integration of Web services that can aid developers by easing this time-consuming task. Specifically, this strategy contains methods that focus on the features of Web service definitions to allow the automatic identification of semantically equivalent elements across the services being integrated. In order to achieve this objective, we provide the following contributions to the research domain:

- A detailed analysis of the service integration problem along with its challenges and a review of previous work on the subject.
- An examination of the problem with real-world data —the case of Mecanica Solutions 360e. This way, we override many previous academic assumptions.
- The introduction of new techniques for automated service integration with real-world data.
- The development of an interactive prototype tool that implements these tech-

niques to aid developers in the task of mapping semantically equivalent elements across services.

- A series of experiments using these techniques on multiple data sets to test the effectiveness of our methods.

This study is organized as follows. Chapter 2 offers an initial introduction of the problem of service integration along with a survey of the literature on the subject. It also provides a review of the different concepts that will be used throughout subsequent chapters for the development of our techniques. Finally, it introduces the case of Mecanica Solutions 360e. Chapter 3 starts with a more in-depth presentation of the problem, applying it to the case study of 360e. It continues by showing the details of our proposed methods. Chapter 4 is divided into two main sections. The first one delineates the features and implementation details of the service integration assistant tool. The second section describes the experiments performed and the results obtained from them. To conclude, chapter 5 offers a review of the contributions of this thesis as well as an overview of topic ideas to be explored in future work.

Chapter 2

Background

Service Oriented Architecture (SOA) is a software development paradigm that is based on the design of components referred to as services. These services are published and provided over a network through a set of common interfaces, and described using a set of formats based on open standards. These standards allow services to be defined and communicate in a common way, regardless of the platform in which they are implemented. In practice, this results in developers leveraging from this technology to create APIs through which individual software systems communicate [34]. On top of this, the standards in which services are described are based on machine-readable formats such as XML, opening up the possibility of automatic service integration —a major breakthrough from previous technologies where integrating multiple information systems was a necessarily labour-intensive task with developers having to write adaptors manually. This chapter presents the main topics used in this thesis. It starts with an introduction to Web services, followed by a section describing the service integration problem and its major challenges, and further sections presenting the main concepts that will be applied in the approach presented in the next chapter.

2.1 WSDL and RESTful Web Services

Two major technologies have emerged for the development of software as a service — Representational State Transfer (REST) [39] services and services based on the Web Service Description Language (WSDL) [11]. The former is a light-weight architecture under which services provide operations to be performed on resources. Each resource is identified by a distinct URI, while the operations are given by the standard HTTP methods (the main ones being GET, POST, PUT and DELETE) [13]. To describe a RESTful service's interface, Sun Microsystems has defined the Web Application Description Language [15] —an XML-based format readable by both humans and machines. However, this protocol has not seen a wide adoption rate in the industry, and most RESTful services remain described in documents aimed to human readers [39].

On the other hand, WSDL is an open standard which is widely used for describing Simple Object Access Protocol (SOAP) Web services [36]. Service interfaces are specified using a common XML format, detailing the names and types of the parameters a service expects. This is what allows for different services to be automatically integrated, using a computer program to inspect their interfaces and adapt parameters to the specific requirements of a service. We will focus on WSDL-based services since they provide a standard that is already widely used in the industry. However, if in the future WADL gets accepted as the common format for describing RESTful Web services, this work can easily be applied to it as well.

2.1.1 The WSDL Format

WSDL documents include several elements that contribute to the definition of services. Multiple actions can be defined in one file —each of them contained in an *operation* element, which are grouped into a *portType* element. The operation el-

ement lists the action's inputs and outputs as well as its exceptions, all of which reference *message* elements. The latter specify the types of data exchanged in the message by referencing elements in the *types* section of the document. All these mentioned elements define what the actions are and what kind of data they handle but they do not specify how to invoke these actions. That is why they are called the abstract portion of the service definition. WSDL also includes a concrete part, which provides the combination of an operation and a protocol used to call it —described in a *binding* element. Bindings are referenced by *port* elements, which tie them to the network address used to invoke the service, and are grouped into *service* elements [11].

Example 2.1.1 *Figure 2.1 shows the abstract portion of a WSDL document. Here we can see the service provides an action called* getProjectDetails*, which has an input and an output message. By looking at the types referenced by the message elements, we can see the input expects an* int *value called* projectID *and the output contains three fields:* projectID*,* projectName *and* startDate*.*

Example 2.1.2 *The concrete part of the service defined on Figure 2.1 can be seen on Figure 2.2. The* binding *element specifies the operation is available via SOAP over HTTP, while the* service *element includes a* port *tying the binding to the URL used for accessing the service.*

Since WSDL files follow this open standard, it is possible to programmatically inspect one to obtain the list of operations defined in its abstract part. With further parsing of the document, the names and types of expected parameters —as well as the ones of return values— can be extracted. This leads to the possibility of automatically creating the client code for a service. The Java API for XML Web Services (JAX-WS) is an API that facilitates the development of Web services using the Java language. It includes a tool called *wsimport* that reads a WSDL document

```
<types>
   <schema targetNamespace="http://wsdl_example" xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="getProjectDetailsRequest">
         <complexType>
         <sequence>
            <element name="projectID" type="int"/>
         </sequence>
         </complexType>
      </element>
      <element name="getProjectDetailsResponse">
         <complexType>
            <sequence>
               <element name="projectID" type="int"/>
               <element name="projectName" type="string"/>
               <element name="startDate" type="date"/>
            </sequence>
         </complexType>
      </element>
   </schema>
</types>

<message name="getProjectDetailsIn">
   <part name="body" element="getProjectDetailsRequest"/>
</message>

<message name="getProjectDetailsOut">
   <part name="body" element="getProjectDetailsResponse"/>
</message>

<portType name="projectPortType">
   <operation name="getProjectDetails">
      <input message="getProjectDetailsIn"/>
      <output message="getProjectDetailsOut"/>
   </operation>
</portType>
```

Figure 2.1: Abstract portion of a WSDL document

```
<binding name="projectBinding" type="tns:projectPortType">
   <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="getProjectDetails">
      <soap:operation soapAction=""/>
      <input>
         <soap:body use="literal"/>
      </input>
      <output>
         <soap:body use="literal"/>
      </output>
   </operation>
</binding>

<service name="projectService">
   <port name="getProjectDetailsPort" binding="tns:StockQuoteBinding">
      <soap:address location="http://somedomain/projectService"/>
   </port>
</service>
```

Figure 2.2: Concrete portion of a WSDL document

and automatically generates code for invoking the Web service [16]. The WSDL is parsed, and a Java class representing the service is created, along with an interface providing the operations defined in the *portType*, and the necessary class structure for the input and output parameters.

Example 2.1.3 *GeoIPService is a service for looking up the country where an IP address is located [47]. Using JAX-WS's* wsimport *tool, we obtain a group of classes that can be used in the development of a client for this service. Figure 2.3 shows part of the WSDL file for this service. As defined in the* types *section, the output of the GetGeoIP operation contains an element of the complex type* GeoIP*. This includes a series of fields such as the country name, country code and IP address. Accordingly, a GeoIP Java class is generated, with instance variables corresponding to those fields. Figure 2.4 shows this Java class. A class representing the service (*GeoIPService*) and an interface providing the service's operations are also created. Figure 2.6 illustrates how the service can be invoked by a client program, using an instance of the GeoIPService class to obtain the* GeoIPServiceSoap *port.*

```
<wsdl:types>
   <s:schema elementFormDefault="qualified" targetNamespace="http://www.webservicex.net/">
      <s:element name="GetGeoIP">
         <s:complexType>
            <s:sequence>
               <s:element minOccurs="0" maxOccurs="1" name="IPAddress" type="s:string"/>
            </s:sequence>
         </s:complexType>
      </s:element>
      <s:element name="GetGeoIPResponse">
         <s:complexType>
            <s:sequence>
               <s:element minOccurs="0" maxOccurs="1" name="GetGeoIPResult" type="tns:GeoIP"/>
            </s:sequence>
         </s:complexType>
      </s:element>
      <s:complexType name="GeoIP">
         <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="ReturnCode" type="s:int"/>
            <s:element minOccurs="0" maxOccurs="1" name="IP" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="ReturnCodeDetails" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="CountryName" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="CountryCode" type="s:string"/>
         </s:sequence>
      </s:complexType>
      <s:element name="GeoIP" nillable="true" type="tns:GeoIP"/>
   </s:schema>
</wsdl:types>

<wsdl:message name="GetGeoIPSoapIn">
   <wsdl:part name="parameters" element="tns:GetGeoIP"/>
</wsdl:message>
<wsdl:message name="GetGeoIPSoapOut">
   <wsdl:part name="parameters" element="tns:GetGeoIPResponse"/>
</wsdl:message>

<wsdl:portType name="GeoIPServiceSoap">
   <wsdl:operation name="GetGeoIP">
      <wsdl:input message="tns:GetGeoIPSoapIn"/>
      <wsdl:output message="tns:GetGeoIPSoapOut"/>
   </wsdl:operation>
</wsdl:portType>
```

Figure 2.3: Abstract definitions from the GeoIPService WSDL

```
public class GeoIP
{
   protected int returnCode;
   protected String ip;
   protected String returnCodeDetails;
   protected String countryName;
   protected String countryCode;

   // getters and setters for all fields
   // ...
}
```

Figure 2.4: Java class generated from the GeoIP XML type

```
public interface GeoIPServiceSoap
{
   public GeoIP getGeoIP(String ipAddress);
}
```

Figure 2.5: Java interface generated from the GeoIPServiceSoap portType

```
GeoIPService service = new GeoIPService();
GeoIPServiceSoap port = service.getGeoIPServiceSoap();
GeoIP geoip = port.getGeoIP(ipAddress);
System.out.println("Country code: " + geoip.getCountryCode());
System.out.println("Country name: " + geoip.getCountryName());
```

Figure 2.6: Invocation of GeoIPService using the code generated by JAX-WS

2.2 The System Integration Problem

System integration refers to the interaction of multiple software components, in order to collaborate in the performance of a task. In the SOA paradigm, these components are services, so we refer to the problem as service integration. As more and more software projects are developed following SOA, the need for service integration becomes more important. However, there is still no automated way to achieve seamless service integration. The main difficulty is that even though services are defined using common standards, these standards only apply to the syntactic structure of service definitions. So while services comply to a standard format, their semantics —which include interfaces and business protocols— are left to be designed by the developers. A service's interface includes the names and types of its operations and parameters, while its business protocol defines the order in which messages should be exchanged when using these operations [18]. This means that two services that have been designed to perform the same task can each be available through a distinct interface and business protocol. The semantic interoperability problem originates when two services that were not specifically designed to interact with each other are required to do so. In such cases, the semantic details of their communication must be agreed on in order for the interaction to succeed [27]. Once this has been achieved, either one

of the interacting services can be modified to match the requirements of the other, or an adaptor can be developed to mediate between them [18].

2.2.1 Approaches to Service Integration

The absence of semantic information in service descriptions is identified as a major item contributing to the service integration problem. To solve this, a practise that has been proposed consists of including semantic descriptions in service definitions via the addition of metadata. Metadata refers to data that describes other data —in this case meta data describes the operations and parameters handled by a service. This can include descriptions of data types, their relationships, and semantic constraints that affect them. To add semantics to a service's data types, metadata references predefined ontologies, which are vocabularies that describe a business domain in a standard way agreed upon by the community. Ontologies allow services to communicate through a set of common terminology in their particular domain [34]. The Resource Description Framework (RDF) [5] and Web Ontology Language (OWL) [14] are XML-based languages endorsed by the World Wide Web Consortium (W3C) used in the definition of ontologies. These ontologies can be combined with existing service description standards. [44] proposes the addition of references to ontologies in order to add semantics to WSDL. Other examples of work in this direction are OWL-S [12] and USDL [4], which share the goals of facilitating the automation of service discovery, invocation and interoperation. Both of these languages are based on the OWL standard, but while the former requires the use of an ontology defined by the appropriate industry, the latter proposes the use of Wordnet [24] as its ontology source. The development of these technologies can help solve the semantic interoperability problem. However, this will depend on a standard being widely adopted by the industry, so that developers can start including semantic information in service definitions.

There also exist approaches to service integration that try to circumvent the lack of semantic information in current service definition standards. These approaches focus on the analysis of the services involved, in search of mismatches in their interfaces and protocols. Several frameworks for the creation of service adaptors have been proposed [18, 45, 42]. The concept of *mismatch patterns* (or *mismatch classes*) is used in [18], [42] and [27], to define abstractions of the differences that occur between service interfaces. Identifying these recurring dissimilarities and providing templates for their resolution are suggested as aids in adaptor development. Mismatch patterns can be at the message or the control flow level, with the former referring to cases where the services involved handle messages of incompatible data types, and the latter describing two processes whose operations have different activity structures.

[42] proposes a framework for service integration, addressing both the message and control flow aspects. It defines mismatch patterns which include *message data generation* to produce default values for fields that are not provided by one of the services, *message data casting* to convert between data formats, *message fragmentation and aggregation* for reformatting data from lists or tuples to single elements, and *message reconstruction* to rearrange tuples or lists. The paper also provides an algorithm for the creation of an adaptor between two services. This algorithm is based in the categorization of mismatches into three cases. First, if a send and a receive action share all the same parameters, they are called *full-duals* and do not need adaptation to work together. Secondly, when all of a send action's parameters are included in a subset of the parameters of a receive action (or vice versa), this is called a partial match. The last case occurs when the union of parameters of a group of send actions matches the parameters of a single receive action, and is called a *full match*. The algorithm iterates through the services' operations looking for instances of these cases, and using this to build a sequence in which messages are sent and their results transformed to create other messages. However, in this paper the au-

thors make two significant assumptions. The first one is that a service's messages and their fields have been previously mapped to their equivalent counterparts in the other service. And associated to this there is a second assumption that a set of functions to perform the necessary transformations of messages between the formats used by the different services have been predefined. This means that an important portion of the integration process is left to be done by the developers —either manually or by other means not contemplated by the authors.

The work in [27] identifies the following message mismatch patterns: a) *message signature pattern* for differences in the message data types; b) *split/merge pattern*, where one service's message corresponds to multiple messages in another; c) *missing/extra message pattern* where a service's message has no correspondence in the other service; and d) *message ordering pattern*, where the order in which a certain message is expected differs from one service to the other. For the latter pattern, a special case is observed when both services in question are waiting for the other to send a message —a *deadlock* situation. The method for service integration proposed by the authors uses finite state machines to navigate through the possible states in the interaction between a pair of services. Each state in the machine consists of a pair, with each component of this pair representing the current state of one of the services. Transitions in the machine indicate a message received by one of the services. The algorithm keeps a queue of the possible states —starting with the initial state— and iterates through the values in the queue. At every step, if a transition is available, the state that results from it is added to the queue. If no transition is available, on the other hand, it means that the state results in a deadlock. All the deadlock states detected by the algorithm are compiled into a *mismatch tree* where a node represents a deadlock state, and its children show future deadlocks that would occur after resolving the original one. To resolve a deadlock situation, the user is prompted for input on how to produce one of the missing messages. The mismatch

tree is presented at this point to aid the user in the decision —depending on the user's choice, subsequent deadlocks can occur. The inputs to the algorithm are the protocol definitions of the pair of services, as well as a mapping between the equivalent messages and parameters of these services. Although their method focuses mostly on the control flow issues of service integration, the authors also propose a means to avoid computing the mapping of interfaces manually, leveraging schema matching techniques. However, they use generic schema matching tool COMA++ [8] which does not take advantage of specific features of WSDL. As we will show in the next chapter, our work proposes methods directed toward the WSDL format, which can produce better results.

2.2.2 Web Service Composition

Web service composition refers to the combination of multiple services —each of which can carry out a specific task— in order to build a composite service to perform a more complex task [34]. This concept is similar to service integration, and both terms are sometimes used interchangeably. However, among the research literature, there are certain issues that are usually associated with the problem of Web service composition. The following aspects have received considerable attention from the research community:

- Finding a way to efficiently compose services to achieve a required result. The problem gets complex when there is a large number of available services to choose from, resulting in a search space too big to be exhaustively searched [29]. As a solution for this, some papers propose the use of Artificial Intelligence (AI) methods, such as A* [30], planning [37, 22, 50], and genetic algorithms [1].

- Adding constraints to the problem described by the previous item, such as quality of service values —for example, throughput, response time and reliability

of the services involved— [51, 1]. This adds complexity to the problem, since instead of finding the smallest number of services to fulfil a task, the algorithms need to take extra considerations to satisfy the constraints.

- Recovering from change among the composed services. When a service that is part of a previously defined composition experiences a failure, it causes the whole composition to malfunction. In such cases, rebuilding the composition from scratch is an option that can be expensive. Research has therefore focused on how to replace the failing service in such cases [53, 52].

Our main concern in this thesis is to discover automated ways of integrating two services so that they can work together. This means that the services that need integration are known by us —as opposed to the case of Web service composition, where an important part of the problem consists in finding the right services which composed can achieve a predetermined task. Moreover, finding the mapping between service interface elements is an aspect not usually studied as part of the Web service composition problem, whereas in this thesis we explore that issue in detail.

2.2.3 Development of a Service Adaptor

As a solution to service integration, a service adaptor can be built. This adaptor is responsible for the mediation between the two services not originally designed to work together [18]. The following main steps can be identified in the development of a service adaptor:

1. Find the matching operations between service S_1 and service S_2. For example, a service provides an operation *retrieveOrder* that matches another service's *createOrder* operation. In some cases, relationships between operations can be more complex than one-to-one matches, such as an operation *getCustomerDetails* matching both *addCustomer* and *addCustomerAddress*.

2. Identify the protocol differences between the services, such as the specific ordering in which operations must be invoked. For instance, on the previous example the first service expects one call to its *getCustomerDetails* operation, which returns one set of return values. The second service however, requires two calls to be made —first to *addCustomer* and then to *addCustomerAddress*.

3. Find the matching parameters between the matching operations, for instance *customerID* to *clientID*, or a one-to-many case such as *employeeName* to *employeeFirstName* and *employeeLastName*.

4. Write functions to transform instances of S_1's output messages to S_2's input messages.

5. Write the mediator process that coordinates the communication between the two processes, by transforming and forwarding the messages from one service to the other.

In this thesis, we will focus on the third step of the list above, assuming the matching operations from each service have already been identified, and leaving out the analysis of the business protocols. The focus on the matching of parameters allows us to concentrate on a complex part of the problem —arguably the most important one. While some cases of integration involve services with single operations, for which steps 1 and 2 become trivial, step 3 always remains essential to the problem. Therefore, our analysis and development of techniques for the matching of semantically equivalent service parameters constitutes our main contribution in this thesis. Furthermore, we intend to leave steps 4 and 5 as the focus of future work.

2.3 Schema Trees

It is common to model an XML schema as a tree structure, using nodes to represent elements and attributes in the schema, and edges to represent the relationship between them. A formal definition of this structure can be seen in Definition 2.3.1 [2].

Definition 2.3.1 *A schema tree T is a 4-tuple $T = (N_T, E_T, Lab_{NT}, l)$ where:*

- $N_T = \{n_1, n2, ..., n_n\}$ *is a set of uniquely identified nodes, which represent either an element or an attribute definition in the XML schema.*
- $E_T = \{(n_i, n_j) | n_i, n_j \in N_T\}$ *is a set of edges, which represent a parent-child relationship between two nodes.*
- Lab_{NT} *is a set of labels, which represent node properties, including name and data type.*
- $l : N_{NT} \mapsto Lab_{NT}$ *which maps every node to its labels.*

Example 2.3.1 *Figures 2.7 and 2.8 show the correspondence between an XML schema and a schema tree.*

```
<?xml version=1.0 ?>

<xsd:schema xmlns:xsd=http://www.w3.org/2000/10/XMLSchema>

   <xsd:simpleType name="nameType">
      <xsd:restriction base="xsd:string">
         <xsd:maxLength value="100"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="telType">
      <xsd:restriction base="xsd:string">
         <xsd:pattern value="(d{3})-d{3}-d{4}"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="dateType">
      <xsd:restriction base="xsd:date"/>
   </xsd:simpleType>

   <xsd:simpleType name="stateType">
      <xsd:restriction base="xsd:string">
         <xsd:maxLength value="2"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="zipType">
      <xsd:restriction base="xsd:string">
         <xsd:pattern value="[0-9]{5}"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="order_numType">
      <xsd:restriction base="xsd:string">
         <xsd:pattern value="[A-Z]{1}[0-9]{6}"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:complexType name="addressType">
      <xsd:sequence>
         <xsd:element name="street" type="nameType"/>
         <xsd:element name="city" type="nameType"/>
         <xsd:element name="state" type="stateType" minOccurs="1" maxOccurs="1"/>
         <xsd:element name="zip_code" type="zipType" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="customerType">
      <xsd:sequence>
         <xsd:element name="name" type="nameType"/>
         <xsd:element name="address" type="addressType"/>
         <xsd:element name="tel" type="telType"/>
         <xsd:element name="ship_date" type="dateType"/>
      </xsd:sequence>
      <xsd:attribute name="order_num" type="order_numType" use="required"/>
   </xsd:complexType>

   <xsd:element name="customer" type="customerType"/>

</xsd:schema>
```

Figure 2.7: Sample XML schema

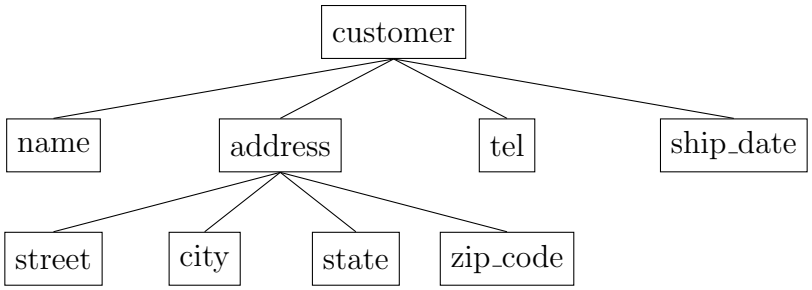

Figure 2.8: Sample schema tree

2.4 Stop Words

Stop words are a concept used in information retrieval to specify words that do not add semantic significance to the context in which they appear. On the other hand, words that do contribute with meaning are called content words. People have compiled generic lists of stop words [19] but unfortunately this type of list will not always be helpful. In fact, terms included in such a list can actually be content words depending on the context in which they are used. This is because the classification of terms into stop words and content words depends on the domain in which they are used [9]. For example, the list in [19] includes the words "group" and "shows". However, if working in the context of music, one will probably consider those to be important content words, used in phrases like "the most popular group of the year" and "several upcoming shows throughout the city".

In this thesis, we use stop words to overcome the problem of terms that do not contribute to the meaning of the service parameters in which they appear. Our approach combines the use of a list of standard stop words, along with human input to build a domain-specific set of stop words.

2.5 String Similarity Metrics

Several metrics have been developed to determine the similarity of two strings. Following is an explanation of three of the most popular ones.

- **Levenshtein or Edit Distance:** This is a measure of the number of character operations needed to transform one string into another. The allowed operations are insertion, deletion and substitution. For example, the distance between the words *manually* and *January* is 3, obtained by replacing m for j, removing the first l, and replacing the second one by an r. This metric has a lower bound of 0, when applied to identical words. Its upper bound is the length of the longer of the two strings being compared [28]. This metric can be normalized into a [0, 1] interval, as in [3]: $norm_e d_d ist(t_1, t_2) = \frac{max(|t_1|,|t_2|)-ed_d ist(t_1,t_1)}{max(|t_1|,|t_2|)}$.

- **Dice Coefficient or n-gram Distance:** n-grams are the sequences of n contiguous characters contained in a string. For example, the 3-gram sets for *manually* and *January* are {man, anu, nua, ual, all, lly} and {jan, anu, nua, uar, ary} respectively. To compare two strings, their sets of n-grams are extracted, and the ratio of common n-grams over the total number of n-grams is computed, following the following formula: $dist(s_1, s_2) = 2 * \frac{|ngrams(s_1) \cap ngrams(s_2)|}{|ngrams(s_1)|+|ngrams(s_2)|}$ [17]. The metric is bounded by 0 at the lower end, in the case of strings with no common n-grams, and 1 in the upper end, for strings sharing all their n-grams. The common 3-gram between *manually* and *January* are *anu* and *nua*, so the distance is $2 * 2/11 = 0.3636$.

- **Jaro Distance:** This metric is based on the count of matching characters between the two strings. A character a appearing in string s_1 at position i and character b appearing in string s_2 at position j are said to be matching if $a = b$ and $|i - j| \leq \delta$, where $\delta = \frac{max(|s_1|,|s_2|)}{2} - 1$. The Jaro distance is computed as follows: $dist = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right)$, where m is the number of matching

characters and t is the number of transpositions, i.e. the number of matching characters for which the order of appearance in n_1 is not the same as in n_2 [6]. For our example of *manually* and *January*, $\delta = \frac{max(8,7)}{2} - 1 = 3$ and the matching characters are $\{a, n, u, a, y\}$, which occur in the same order in both words. So the Jaro distance is $dist = \frac{1}{3}\left(\frac{5}{8} + \frac{5}{7} + \frac{5-0}{5}\right) = 1.7798$.

2.6 Schema Matching

As we have previously discussed, a WSDL document provides the list of parameter names and data types used by the service it defines. We will then need to identify the common concepts represented in a pair of service definitions. Since SOAP parameters are XML structures, they are detailed in their corresponding WSDL files using the XML Schema Definition language (XSD) [35]. Therefore we present the concept of schema matching which we will use in the next chapter.

Schema matching is the action of identifying semantically equivalent elements among two schemata. This task is commonly performed in a variety of applications, such as the merging of different data sources into a data warehouse, the migration of data from an XML-based source into a relational database, or the integration of two databases after a company is bought by another one [8]. Following are some of the challenges encountered in schema matching.

2.6.1 Challenges in Schema Matching

- **Each schema may use different terms to refer to the same concept.** When designing an API, developers must decide on how to name each field in the set of input and output parameters. Since each system is developed by a different group of people, differences in these names are bound to happen. For example, one software may use the term *entry* while another one uses the term

```
<header id=''1''>
   <items>
      <item id=''1'' name=''A'' />
      <item id=''2'' name=''B'' />
   </items>
</header>
```

```
<header>
   <id>1</id>
   <item id=''1''>A</item>
   <item id=''2''>B</item>
</header>
```

Figure 2.9: Semantically equivalent XML structures

item to refer to the same thing.

- **The structures of two schemata may vary.** An XML schema follows a tree-like structure. As is the case with field naming, schema structure greatly depends on the developers' design choices. Figure 2.9 shows an example of two XML instances that represent the same information using different structures.

- **Field names may contain terms that do not directly contribute to the fields semantics.** In some cases, the name of a field can contain one or more terms that do not add meaning to the field itself. Instead, they could refer more generally to the problem domain, or be too generic to be considered in the matching process. For example, in the case of employee time sheets, we could find fields named *timeSheetDetail*, *timeSheetEntry* and *timeSheetProjectID*. The terms *time* and *sheet* are present in all of them but refer to the domain itself, instead of the specific meaning of each field.

- **Fields in one schema may not have a corresponding match on the other.** Often a subset of the concepts defined in a schema will not exist in the schema to which is being matched. In such cases, the group of fields which define these concepts will not be mapped to any field in the other schema.

2.6.2 Approaches to Schema Matching

Approaches to schema matching can be divided into two categories, a schema-level one, and an instance-level one. The former relies on schema definition information —such as elements, tables, relationships and constraints— to discover the matches, while the latter approach inspects the values found in instances of a schema [38].

- **Instance-level matching.** In situations where data samples are available, instance-level approaches can effectively help in the matching process by inspecting the contents of schema elements. These techniques rely on the values inside schema elements to identify patterns among instances of the different schemata. One method consists in linguistically analyzing the data instances to find frequently occurring terms, which are then used to suggest matches between the fields that contain those common terms. Instance data can also be inspected to find the specific type of information contained in a field. For example, a field in schema X can be identified as being an email address by inspecting the format of its data samples. Therefore, its counterpart in schema Y would be expected to also hold email addresses [10].

- **Schema-level matching.** Multiple approaches have been developed to discover element mappings based only on the information included in the data schema definitions [43]. Element names, data types and the structures in which they are arranged constitute the input to this kind of technique. [8] and [20] use string similarity measures to find schema elements with similar names. Initially, the strings are normalized and tokenized to extract the terms inside them. Then a variety of measures, such as Edit Distance, n-gram, and soundex are applied to the pairs of terms being compared. These measures are combined and standardized into a value in the [0, 1] range, a potential similarity value between elements in the different schemata. XML provides a set of built-in data

types to be used when defining a schema in XSD [35]. This information can also contribute to the process of matching elements. [8] includes a data type matcher which checks the schema elements' data types against a lookup table. This table assigns a compatibility value to each combination of data types. For example, the decimal-float pair will have a higher value than the decimal-string pair. Another way of analyzing the element labels is by using a thesaurus, which can be either a general purpose one or one containing terms specific to the domain of the schemata in question. For a general purpose list of synonyms, Wordnet is often used. Wordnet is a lexical database that groups words into sets of synonyms, like in a thesaurus. However, it also provides extra features, such as linking of synonym sets into a hierarchical structure of hyperonyms and hyponyms [24]. A word x is a hyperonym of a word y if x's meaning includes y but x is a less specific term, i.e. y is a type of x. Conversely, y is a hyponym of x. For example, *appliance* is a hyperonym of *dishwasher*, and dishwasher is a *hyponym* of *appliance*.

In our study of the integration of service-oriented systems, we will leverage schema matching techniques to identify the common concepts shared by the services being integrated. Specifically, we choose a schema-level approach, since data instances for the inputs and outputs of the services in question may not be available.

2.6.3 The XPrüM System

In [2] the authors propose a schema-level matching method that combines structure and element measures. Each schema is parsed and a tree is built from it. Then, terms are extracted from node names by splitting these names at common boundaries such as underscores, spaces and uppercase letters. With the tokens extracted from the schema trees' nodes, the process continues by comparing the token sets of each pair of nodes (n_S, n_T) where n_S is a node in the source schema tree and n_T is a node in the

```
input: n_S, n_T
procedure tokenSetSimilarity
   T_S := extractTokens(n_S)
   T_T := extractTokens(n_T)

   for t_1 ∈ T_S do
      m := 0
      for t_2 ∈ T_T do
         m := max(stringSimilarity(t_1, t_2), m)
      end for
      sum += m
   end for

   for t_2 ∈ T_T do
      m := 0
      for t_1 ∈ T_S do
         m := max(stringSimilarity(t_2, t_1), m)
      end for
      sum += m
   end for

   numOFTokens := size(T_S) + size(T_T)

   return sum / numOfTokens
end procedure
```

Figure 2.10: Calculating similarity between node names

target schema tree. The tokens are compared using a combination of the edit, trigram and Jaro string similarity metrics and an average maximum is computed as shown in Figure 2.10. The data types of the XML elements are also compared during this phase, by using a lookup table that assigns compatibility values between pairs of XML data types. Thus, the data types of the nodes being compared are checked against this table and their compatibility value is combined with the previously computed name similarity value, to obtain their so-called linguistic similarity.

This approach then continues by comparing the nodes according to their context in the tree structures.

Definition 2.6.1 *A node* n *is said to have:*

- *A child context, which contains the children of* n.
- *A leaf context, which contains all leaf nodes descending from* n.
- *An ancestor context, which contains all nodes in the path from the root node to* n.

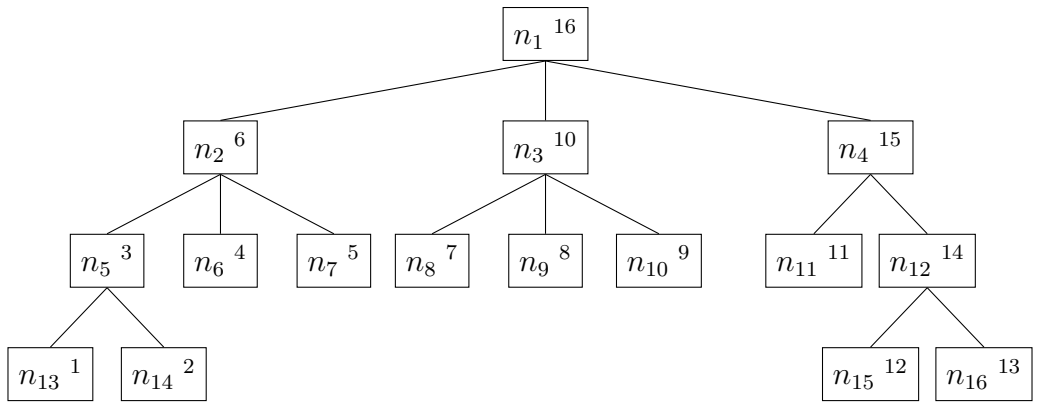

Figure 2.11: Schema tree with post order numbers

To structurally compare a pair of nodes, the similarity of each of their contexts defined in Definition 2.6.1 is measured, and the three values are combined. The child contexts of nodes n_S and n_T are compared by finding, for each child of n_S, the child of n_T which has the maximum linguistic similarity to it, then computing an average of these maximum linguistic similarity values. This grants higher child context similarity values to pairs of nodes with higher proportion of children that are linguistically similar. Since leaf nodes have no children, comparison of child contexts is only done between internal nodes.

Leaf context similarity is based on the gap vectors of the nodes being compared. A gap vector contains the differences between the post order number of a node and that of each of its descending leaves. The similarity between two leaf contexts is then defined as the cosine measure of their gap vectors. As with child context, leaf context similarity is only relevant to internal nodes.

Example 2.6.1 *Figure 2.11 shows a sample schema tree with its post order numbers assigned. The leaf context of node n_2 is $\{n_{13}, n_{14}, n_6, n_7\}$, and its gap vector is $\{5, 4, 2, 1\}$.*

Ancestor context similarity is a measure of how two paths —from the tree root to the node being compared— resemble each other. This is obtained by computing the

weighted sum of three measures, all of them normalized to fit in the [0, 1] range:

- The longest common sequence between the paths, to ensure similar nodes in both paths appear in the same order. To determine if two nodes are common, their linguistic similarity is checked against a predefined threshold.
- A measure of the gaps between the nodes in the paths, in order to give higher scores to nodes that are closer together.
- The difference between the lengths of the paths, assigning a higher value to paths of similar length.

Thus, complex nodes from each tree are paired and the structural similarity metrics defined above are computed on them. Then the combined value of their linguistic and structural similarities are checked against a predefined threshold. The pairs of nodes with a value that exceeds this threshold are put in a set of compatible nodes. For each compatible node, a category set is built from the union of:

- all its children which are leaves,
- all its non-leaf children that are not compatible nodes, and
- all the children of the latter which are leaves.

Finally, for every pair of compatible nodes, the linguistic and structural similarity of the nodes in their category sets are summed. The pairs of nodes with the highest values are then suggested as the best matching candidates.

2.7 The Case of Mecanica Solutions 360 Enterprise

For this research we have teamed up with our industry partner Mecanica Solutions [23] to provide a real-world case study. The people at Mecanica have developed a Product

Lifecycle Management (PLM) software suite called 360 Enterprise (360e) aimed at the optimization of product development throughout an enterprise. Its features are divided into four main modules:

- **Universal Product Management:** Allows the importing of engineering data such as CAD documents, and provides different visualization tools to keep track of product parts and configurations.

- **Business Process Management:** Provides standardization of business processes by switching from manual to automated processes —which can also be saved into templates to be reused. Tasks can be assigned to users according to their type, and automatic messages are sent to the appropriate stakeholders when a status has changed.

- **Enterprise Program Management:** Includes tools for monitoring multiple projects and portfolios. Stores information about human resources which includes a person's skills and provides a view of their current and upcoming workload. It also includes time sheet management to keep track of the tasks an employee has worked on.

- **Document Management System:** Stores all kinds of documents, keeping track of changes and providing document sharing through the network, and access control depending of user privileges.

Enterprise Resource Planning (ERP) systems are integrated solutions that provide tools aimed at managing all aspects of an enterprise. These include modules for accounting, project management, human resources, supply chain management, and manufacturing. There are areas in common between these products and Mecanica's 360e software, which means that some of the information managed by these applications will be of the same type. Moreover, it is not uncommon for a company to

use 360e as their PLM software and an ERP system —such as SAP [41], JD Edwards [32], or PeopleSoft [33]— to cover their accounting and human resources needs for example. In such cases, the ability to share data between applications becomes important.

For example, a company may enter employee time sheet information into their SAP system for payroll purposes, and then import this information into the 360e system to be taken into account for project management. For this to be achieved, the current solution requires Mecanica's developers to manually build a module for the integration of time sheet data between SAP and 360e. This involves:

- The analysis of SAP's time sheet APIs and data structures to find how each concept is represented and how they can be mapped to the representations of the same concepts in 360e's data structures.
- The development of an adaptor process that converts the time sheet data from SAP's structure into 360e's own format.

With the current solution, this would have to be repeated for a client who wants to integrate time sheet information from a different ERP system. Similarly, if a client requires the integration of project information from SAP, the whole process needs to be repeated for this situation. In general, the developers will have to go through this manual process for each pair (E, D), where E is a third-party ERP system and D is some type of data that needs to be integrated.

We can see that the current solution is far from ideal, since it requires potentially repeating these tasks many times. This is why we want to develop a new method, in which service integration is automated as much as possible, from the first step of finding a mapping of semantically equivalent concepts between the two systems involved, to the creation of data adaptors to transform instances of this data. It is not trivial to write an algorithm for these tasks —each case can have arbitrary

complexities depending on how the data structures have been defined— so we cannot expect to achieve a wholly automated solution. However, our work has permitted the development of semi-automated techniques for system integration. Moreover, we have built a service integration assistant that relies on the methods described in this thesis to partially automate the service integration process, easing the work of developers. In the next two chapters we will present our proposed techniques as well as the tool we have built, and the experiments we have conducted to test them.

Chapter 3

Automatic Service Integration

In the previous chapter we offered an introduction to the topic of service integration, introducing the semantic interoperability problem and its challenges, as well as the topics relevant to the developing of solutions to the problem. This chapter presents a more thorough description of the problem as well as the approach we have developed for solving it.

3.1 Presentation of the Problem

To illustrate the topics covered by this work, we provide the following example, based on the case study of the integration of Mecanica Solutions 360e and SAP's ERP system. Let us consider an engineering company ABC which uses 360e for their project lifecycle management tasks. Moreover, ABC uses SAP's ERP solution for several tasks, including their accounting and human resources needs. For accounting purposes, ABC keeps track of how many hours have been spent on specific projects and tasks. Every employee has to enter this information into a time sheet interface in the ERP system, which is later reviewed and approved by a supervisor. Additionally, project managers need to know which employees have worked on which tasks and for how long. To record this, employees are required to re-enter the same information

into the PLM system, which is tedious and error-prone.

This real-world scenario represents a typical situation where system integration is required. Ideally, the employees would need to enter their time sheet information into the ERP system only, and an automated process would transfer this data into the PLM system. SAP has a Web service based API that can be used to query employee time sheet information. On the other hand, 360e also provides Web service operations to enter employee time sheets. In theory, these two pieces of software could be made to communicate with each other through their Web services in order to transfer time sheets from the ERP to the PLM system. However, for this integration to happen, an adaptor is needed since the two applications were not specifically designed to work together.

SAP's Web service exposes an operation called *Find Employee Time Sheet By Employee*, which takes the ID of the employee and a date period as parameters. We will call this the *source service* and its operation the *producer operation*. Figure 3.1 is an excerpt of the WSDL for this service. On the side of Mecanica's 360e, we find an operation called *OpenTSExecute* which can be used to enter an employee's time sheet entry into the system —what we will call the *consumer operation*, provided by the *target service*. Figure 3.2 shows an excerpt of this WSDL. We identify two main incompatibilities between the services: a) the producer operation can extract a whole list of time sheet entries in a single request whereas the consumer operation requires a request per entry being added, and b) the format in which time sheet data is specified is not the same on both systems. A process to transfer data from SAP to 360e will then have to take the following steps:

1. Accept an employee ID and data period as initial input.

2. Invoke SAP's service, passing these parameters to it, and receiving a list of time sheet entries in return.

```
<wsdl:message name="EmployeeTimeSheetByEmployeeQuery">
   <wsdl:documentation>Query for EmployeeTimeSheetByEmployee</wsdl:documentation>
   <wsdl:part element="p2:EmployeeTimeSheetByEmployeeQuery" name="EmployeeTimeSheetByEmployeeQuery"/>
</wsdl:message>
<wsdl:message name="EmployeeTimeSheetByEmployeeResponse">
   <wsdl:documentation>Response for EmployeeTimeSheetByEmployee</wsdl:documentation>
   <wsdl:part element="p2:EmployeeTimeSheetByEmployeeResponse"
        name="EmployeeTimeSheetByEmployeeResponse"/>
</wsdl:message>
<wsdl:message name="StandardMessageFault">
   <wsdl:documentation>Default Standard Fault Message</wsdl:documentation>
   <wsdl:part element="p2:StandardMessageFault" name="StandardMessageFault"/>
</wsdl:message>
<wsdl:portType name="EmployeeTimeSheetByEmployeeQueryResponse_In">
   <wsdl:documentation>Find Employee Time Sheet By Employee</wsdl:documentation>
   <wsdl:operation name="EmployeeTimeSheetByEmployeeQueryResponse_In">
      <wsdl:documentation>Interface for EmployeeTimeSheetByEmployee</wsdl:documentation>
      <wsp:Policy>
         <wsp:PolicyReference URI="#OP_EmployeeTimeSheetByEmployeeQueryResponse_In"/>
      </wsp:Policy>
      <wsdl:input message="p1:EmployeeTimeSheetByEmployeeQuery"/>
      <wsdl:output message="p1:EmployeeTimeSheetByEmployeeResponse"/>
      <wsdl:fault message="p1:StandardMessageFault" name="StandardMessageFault"/>
   </wsdl:operation>
</wsdl:portType>
```

Figure 3.1: Excerpt from SAP's time sheet WSDL

3. Reformat the time sheet entry list into a series of records according to 360e's specification.

4. Send each record in a request to 360e's service.

In general, the producer operation is invoked with an input, and it returns a set of outputs. Therefore, after it has been invoked, the values for both its input and output parameters will be available to be used as input to the consumer operation. We then define a process *transferTimeSheet* that will invoke *Find Employee Time Sheet By Employee* and use its input and output to generate calls to *OpenTSExecute* —the *integrated process*.

```
<wsdl:message name="OpenTSExecuteSoapIn">
   <wsdl:part name="parameters" element="tns:OpenTSExecute"/>
</wsdl:message>
<wsdl:message name="OpenTSExecuteSoapOut">
   <wsdl:part name="parameters" element="tns:OpenTSExecuteResponse"/>
</wsdl:message>
<wsdl:portType name="OpenTSSoap">
   <wsdl:operation name="OpenTSExecute">
      <wsdl:input message="tns:OpenTSExecuteSoapIn"/>
      <wsdl:output message="tns:OpenTSExecuteSoapOut"/>
   </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="OpenTSSoap" type="tns:OpenTSSoap">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="OpenTSExecute">
      <soap:operation soapAction="http://mecanicasolutions/plm360/ws/OpenTSExecute" style="document"/>
      <wsdl:input>
         <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
         <soap:body use="literal"/>
      </wsdl:output>
   </wsdl:operation>
</wsdl:binding>
<wsdl:service name="OpenTS">
   <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">A Web service that performs
        timesheet management.</wsdl:documentation>
   <wsdl:port name="OpenTSSoap" binding="tns:OpenTSSoap">
      <soap:address location="http://localhost/OpenMPM/OpenTS.asmx"/>
   </wsdl:port>
</wsdl:service>
```

Figure 3.2: Excerpt from 360e's time sheet WSDL

Definition 3.1.1 *We define our Web service interface mapping problem over a tuple* $(p, c, A, B, Y, Z, t, ip)$ *where:*

- $p : A \rightarrow B$ *is a producer operation that expects a set of input parameters* A*, and returns a set of output parameters* B*.*
- $c : Y \rightarrow Z$ *is a consumer operation that expects a set of input parameters* Y*, and returns a set of output parameters* Z*.*
- Y *can be generated by a process* $t : (A \cup B) \rightarrow Y$*, which transforms the concepts represented by* A *and* B *into the format required by* c*. This process is in charge of building an input message for* c *—i.e., its set of parameters* Y*—. It does so by using the values in* A *and* B *—which are in the format handled by* p*— and reformatting them to fit the format expected by* c*.*
- $ip : A \rightarrow Z$ *is the integrated process that combines the functionality of* p *and* c*.*

Figure 3.3 shows a model of the process *ip*, where it is fed an input A and it produces an output Z. Internally, *ip* interacts with services p and c and its mediator t to achieve this.

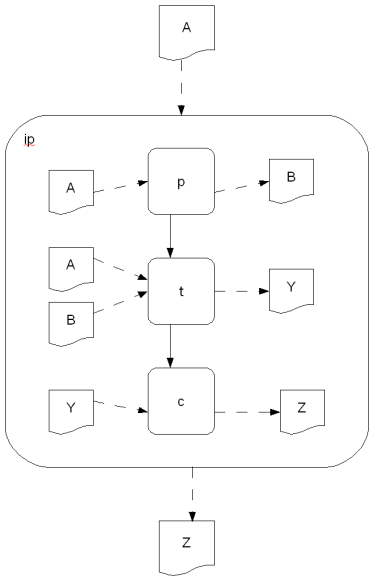

Figure 3.3: Services p and c are integrated into process ip

3.2 Challenges

In order to achieve service integration, two major steps are required. First the APIs for the producer and consumer operations have to be analyzed, looking at their data structures and identifying the pairs of semantically equivalent fields. Once this is done, an adaptor process has to be built, which will handle the interaction between the two operations. This includes the conversion of instances of the data structures handled by the producer operation into the format of the consumer operation. This research contributes to the research in system integration by providing an analysis of the complete requirements for automated service integration, and presenting new techniques for discovering semantically equivalent service parameters.

Definition 3.2.1 *We define semantically equivalent parameters as those which carry information about the same concepts, regardless of the format used to represent this information.*

In the case of Web services, parameters are defined as XSD schemata. Therefore, the identification of semantically equivalent parameters across a couple of services

can be seen as a type of schema matching. However, the nature of WSDL adds some particularities to this variation of schema matching, which makes it worth studying as a separate problem. We summarize some of these issues below:

- **Lack of a formal ontology.** Web service parameters are defined using terms that are meaningful to the designers and the intended main target users of the service. However, these terms do not usually conform to a standard industry-agreed ontology.

- **Stop words.** Some of the terms which appear in parameter names do not contribute to the meaning of the fields in which they occur.

- **Highly context dependent.** The terms used to define fields have a specific meaning in the context of the industry to which the service belongs. Resulting from this, a generic stop word list may not be useful to filter out non-meaningful terms. Similarly, when trying to match terms sharing the same meaning, a standard thesaurus does not usually help.

- **Random schema structure.** Some schema matching techniques put emphasis in the similarity structures [43]. However, the structure of Web service parameters can vary greatly and this practise may not help identifying semantically equivalent elements.

- **Variation in cardinality.** Another item that varies from service to service is the cardinality of the elements. One operation may handle a list of a specific concept while the other one works with individual instances. Therefore, this cannot be used as a comparison factor.

- **Random data types.** Similarly to the previous two points, the choice of XML data types for the definition of atomic fields can vary among multiple services, so using it as a similarity factor may not improve the results.

- **Non-matchable concepts.** The occurrence of concepts in one schema that have no matching counterpart in the other makes some branches of the schema non-matchable.

Example 3.2.1 *Figure 3.4 shows sections of two schema trees, indicating some mappings between semantically equivalent fields. The tree on the left belongs to SAP's service while the one on the right corresponds to Mecanica 360e's service. The first thing to notice is that the term* timeSheet *is a stop word in 360e's tree, appearing in multiple nodes without contributing to their meaning. Furthermore, the mapping between nodes* /EmployeeTimeSheet/EmployeeTime/Item/ID *and* /OpenTSParam/timeEntryId *serves as an example of the lack of a formal ontology: the same concept is called* item *by SAP and* entry *by 360e. Lastly, 360e's service includes a set of fields under the node called* metaDataList *that can be used to pass extra information about the time sheet data. These are fields that have no correspondence on the SAP side.*

Once the pairs of semantically equivalent parameters have been identified, instances of the source service's parameters must be adapted to the format of the target service. The way in which this transformation of the parameter structures is achieved will vary depending on the difference in formatting between a parameter and its matching counterpart. Some possibilities are:

- The trivial case, where both services use the same format to represent a concept. This allows to simply copy the value in question from its containing field in the source structure to the matching field in the target structure.

- Cases where the data has a slightly different format, e.g. *[YYYY-MM-DD]* and *[DD-MM-YYYY]* date fields, or *[FIRST_NAME LAST_NAME]* and *[LAST_NAME, FIRST_NAME]* name fields. These can be reformatted by a script that extracts the individual tokens out of the source value and repositions them according to the target format.

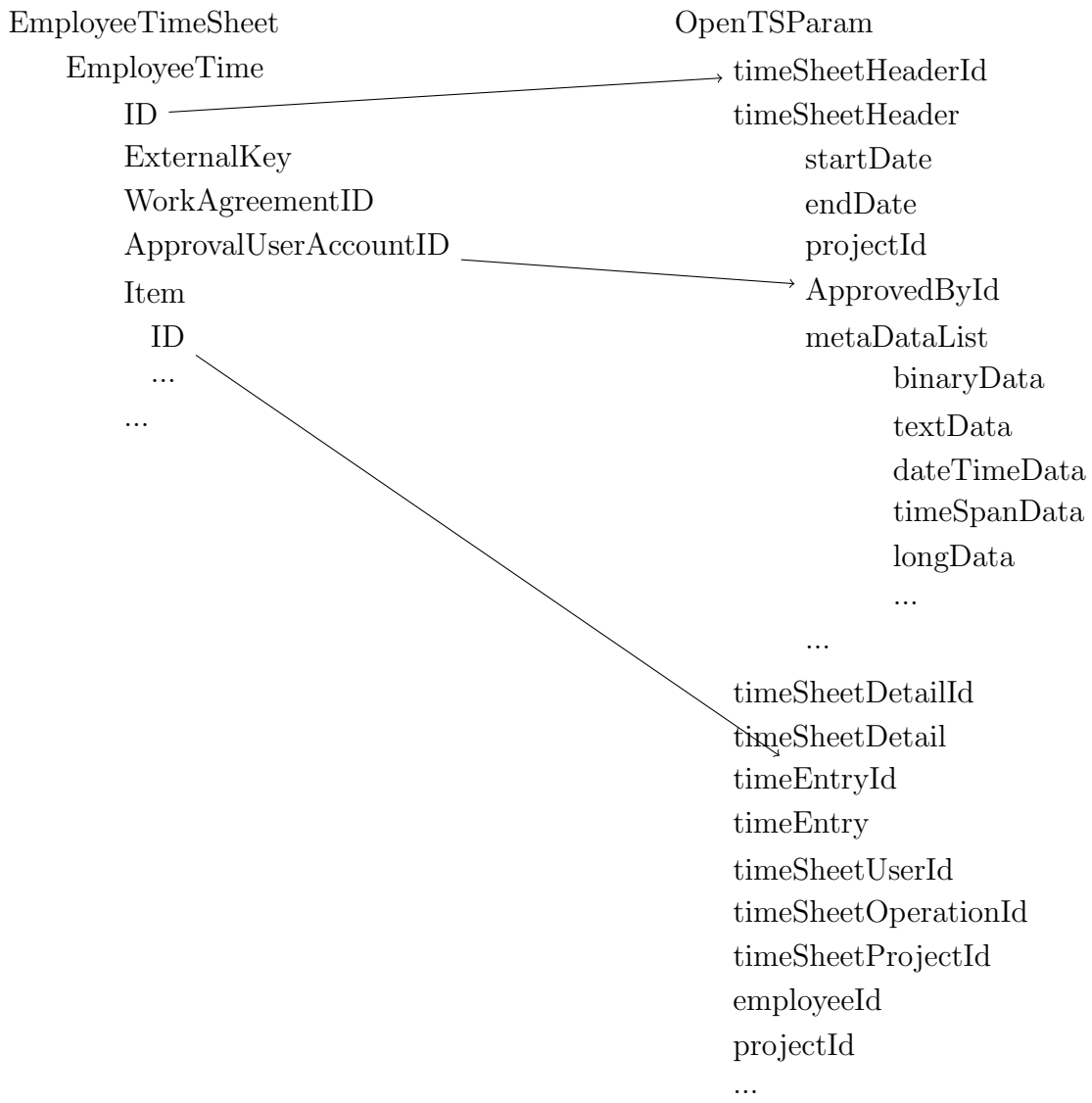

Figure 3.4: Sections of the SAP and 360e schema trees

- When a concept that is represented in one field in an API, is split into two fields by the other, a join or split operation needs to be performed. For example, the values for *firstName* and *lastName* fields will need to be joined to fit into a *fullName* field.

- Sometimes there are multiple ways of referring to the same concept. An example of this is how a company can give their suppliers both an ID and a unique code. In this case, if an API works with a *supplierID* field and another one with a *supplierCode* field, the transformation will require the use of a lookup table to convert from one to the other.

The previous list shows some of the types of transformations that an adaptor process will need to perform —other cases can arise depending on the structures in which data is defined. Moreover, we would like to point out that the employee time sheet example discussed in this section represents a two operation case in which the output of the first operation is used as the input of the second one. More complex situations spanning more than two operations also exist, and they result on a less straightforward business protocol. Handling such cases is also the responsibility of the adaptor process. However, the task of identifying semantically equivalent parameters remains essentially the same regardless of the complexity of the business protocol. The main contribution of this work is the presentation of new techniques for the matching of semantically equivalent service parameters, which have been developed focusing on the particular features of the WSDL format. This is a valuable contribution to the research field because it covers an issue not yet resolved by other work. Some studies, such as [42], have concentrated on other aspects of service integration, assuming the mapping of parameters has previously been done. On the other hand, the authors of [27] have focused part of their study on the problem of parameter mapping but their solution uses the generic schema matching tool COMA++ [8], which does not concentrate on the specific intricacies of WSDL.

The next section describes the details of our proposed techniques, whereas chapter 4 presents the tool we have built leveraging from them, as well as the experiments conducted to test them. Further analysis and development of the data adaptor process is beyond the scope of this thesis, and is left as the topic for future work.

3.3 Techniques for Service Parameter Matching

This section features the techniques for parameter matching contributed by this work.

3.3.1 Preparation of the Data

Our approach starts by building a schema tree for each of the two operations involved. The one corresponding to the consumer operation parameters will be called the *source tree*, while the tree that represents the parameters from the producer operation will be called the *target tree*. The root of the source tree will have two children: one corresponding to the operation's input parameters, the other one to its output parameters. Since the output of the consuming operation is not relevant for the integration, the target tree will only contain the input of said operation.

Having constructed the trees, the next steps will analyze their node names to extract the terms that will be used for comparison. Following are the series of steps performed to accomplish this:

1. **Tokenization.** Node names can often contain multiple terms, formatted in different ways. For instance, they can be camel cased, underscore separated, or space separated. Thus, node names have to be tokenized and normalized, i.e. their individual terms must be extracted and turned into lower case letters. For example, a node called *employeeStartDate* will be tokenized into {*employee, start, date*}.

2. **Elimination of stop words.** The next task we want to accomplish is cleaning up the schema trees from terms that are not relevant, so that they do not interfere with the matching algorithms. For this, a list of stop words will be used. Nodes in both trees will be scanned and any occurrences of stop words in their token sets will be removed from them. It would be possible to use a standard list of stop words such as the one in [19]. However, the same word can either be a stop word or a content word depending on the domain in which it appears [9], especially so in the definitions of Web services parameters, which usually contain a relatively small set of industry-specific terms. So using such a generic list brings in the problem of semantically relevant words being eliminated. Therefore, a better approach consists in using a context-specific list of stop words.

3. **Addition of synonyms.** Similar concepts are often described using different terms by the distinct services. To overcome this issue we use a thesaurus to find synonyms for these terms. For each node in the schema trees, tokens are looked up in the thesaurus, and their synonyms are added to the node's token set. As in the case of stop words, synonyms can also vary greatly depending on the context. For example, the words *field* and *column* are often given the same meaning in the context of relational databases —another example being the terms *record* and *row*. However, Wordnet, being a general purpose tool, does not include these pairs of terms in the same synonym sets. Therefore, we will prefer the use of a context-specific thesaurus. Some of these have been published by a specific industry, for instance the food and agriculture thesaurus Agrovoc [48]. In cases where a domain specific thesaurus is not available, the method proposed in [26] can be used to extract one from Wikipedia.

Example 3.3.1 *For a node named* manager_id*, the tokens* manager, id *are extracted, and the synonym* supervisor *is added for* manager*, so the resulting*

token set is manager, id, supervisor.

4. **Combination of parent and child nodes.** Because of the hierarchical structure of Web service parameters, it is common to have nodes whose names are not sufficiently descriptive of the concept they represent. Instead, the concept is defined by both the node and its ancestors. In particular, a node often represents a property of their parent node, so the node's name refers to that property but not to the concept it is applied to (which is described by the parent's name). To circumvent this issue, we have decided to propagate a node's tokens toward its descendants. One possibility was to assign, for a given node, the union of all tokens found in the path from it to the root. The problem with this option is that it essentially flattens the tree structure, and makes most nodes similar to each other, since nodes at deeper levels will all share the same tokens coming from nodes at the top levels. Therefore, a better option was to combine only a node's tokens with the ones from its parent. As shown on section 4.2.2, our experiments have confirmed that this mechanism produces better results.

   Example 3.3.2 *The schema tree in Figure 3.5 contains two nodes named* id *and two named* code*. They represent properties of their respective parents, namely the* documentHeader *and* detail *nodes. By merging the names of the leaf nodes with their parents', the token set for node 3 is* item, id *while the one for node 2 is* document, header, code.

5. **Removal of non-matchable elements.** As we have already discussed, the services being integrated may have very dissimilar designs. It is not uncommon then for some of the parameters in one service to refer to concepts that do not exist on the other service. These parameters cannot be matched since they have no counterpart in the other schema, so they can be ignored by the algorithm. This means that if we had a way of identifying them, it would help speeding

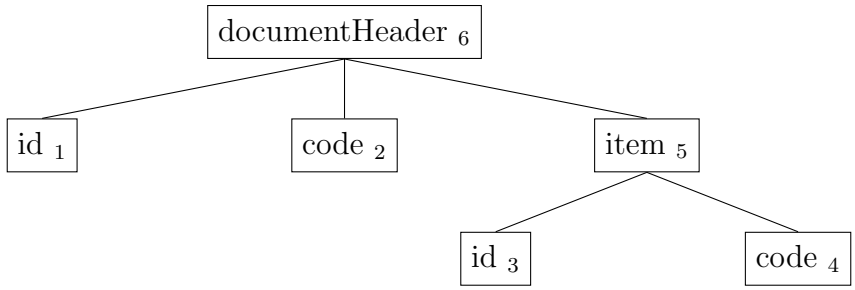

Figure 3.5: Sample schema tree

up the process, as well as making it more effective by filtering potential false positives. For this purpose, we use a list of terms that can be ignored by the matching algorithm —when scanning a schema tree, if a node that includes one of these terms is found, we tag it as non-matchable, so that the next steps of the process will ignore it.

3.3.2 Computing Linguistic Similarities

Having obtained and cleaned up the token sets for the tree nodes, the next step consists in measuring the similarity between nodes in the source and target trees. To this end, we use the measure of linguistic similarity in Definition 3.3.1.

Definition 3.3.1 *A measure of linguistic similarity* linsim *is computed as follows:*

$$linsim(n_1, n_2) = \frac{w_{type} * typecomp(n_1, n_2) + w_{name} * namesim(n_1, n_2)}{w_{type} + w_{name}}$$

where,

- typecomp *is a lookup function that checks the data types of two nodes and returns a value between 0 and 1 representing their compatibility.*

- namesim *is a metric of name similarity between two nodes. A combination of bigram measure and a normalized edit distance is applied to each pair of tokens obtained with a token from* n_1 *and a token from* n_2*), and an average of the maximum similarity pairs is computed. The algorithm on Figure 2.10 shows in more detail how this is calculated. Before choosing this similarity metric, we have tried using different combinations of edit distance, bigram, trigram and Jaro metrics. Our tests produced the best results with the bigram and edit distance measures. This supports previous evidence that bigram is usually more effective than trigram [17]. Moreover, edit distance is not biased against the boundaries of terms, as n-gram measures are [17], so combining both measures provides better coverage of different cases.*

- w_{type} *and* w_{name} *are constants that assign weight to the values of type compatibility and name similarity functions respectively.*

Intuition suggested that name similarity is a more important factor in determining node similarity so we should give it a higher weight than that of data type compatibility. This can be explained by the fact that data type selection can be arbitrary, e.g. one can store a date value in a string field, a numeric field holding a UTC value, or a date field. Our experiments have shown that this is in fact the case, with the best results obtained when the value of w_{name} is five times that of w_{type}.

Following the algorithm in Figure 3.6, a matrix is constructed to hold the linguistic similarity values of nodes.

3.3.3 Comparing Schema Elements

With the linguistic similarities computed, we continue by comparing all pairs of complex nodes (n_S, n_T) where n_S is a node in the source tree, and n_T is a node in the target tree. We use an average of three metrics for this comparison:

```
input: sourceTree, targetTree
procedure buildLingSimilarityMatrix
   for n_S ∈ sourceTree do
      for n_T ∈ targetTree do
         m[n_S][n_T] := linsim(n_S, n_T)
      end for
   end for
end procedure
```

Figure 3.6: Building linguistic similarity matrix

- A child similarity measure to identify pairs of nodes (n_S, n_T) where a high proportion of n_S's children are linguistically similar to children of n_T.
- A leaf similarity measure to identify pairs of nodes (n_S, n_T) where n_S is an ancestor of a high proportion of leaf nodes that are linguistically similar to leaf nodes descending from n_T.
- The linguistic similarity of n_S and n_T.

By using these three metrics, we can consider not only the similarity of the two nodes being compared, but also the similarity of the nodes that descend from them. In other words, for a pair of nodes, the similarity of their descendants contributes to their own similarity.

We then proceed to select compatible nodes, i.e. the pairs of complex nodes whose similarity exceeds a predetermined threshold. For each of these pairs, we extract their category sets as done in [2] and explained in section 2.6.3. Then, for every source node in a category set, we sort in descending order of similarity the nodes in the category set of its corresponding compatible target node. This produces a list of source nodes along with a list of best matching target nodes for each of them.

3.3.4 Obtaining the Matching Pairs

It would be possible at this point to take the top target node —or the top k target nodes— for each source node and present them as the potential matches. However,

some of these candidate matches may not be strong enough, and we would like to filter them out in order to avoid returning false positives. What the authors of [2] propose is to take the candidates with similarity values exceeding a predefined threshold. However, we have found there is considerable variation between the similarity values of different correct matches. In other words, by using a single threshold to determine whether a candidate is accepted or not, we would either filter out many correct matches (if the threshold is set too high), or accept many wrong matches (if the threshold is set too low.) To get a better filtering mechanism, we propose to compare the computed similarity values against two thresholds. First, we compare a source element's top candidate against a top candidate threshold (TH_{tc}). If the similarity value exceeds the threshold, we take the α next best candidates for the node, where α is a predefined constant, and compute the average similarity value among them. If the difference between the top candidate's similarity value and this average is larger than the difference threshold ((TH_{df})), we consider the candidate as a positive match. This method guarantees that the selected candidate not only has a generally high similarity value but that the value is also significantly higher than that of other candidates for the same source node. Figure 3.7 shows the algorithm used for this, and section 4.2.3 offers a comparison of the results obtained using the one-threshold and two-threshold selection methods.

```
input: sourceTree, potentialMatchLists, similarityValues, α, (TH_tc), (TH_df)
procedure suggestMatchings
   for n_S ∈ sourceTree do
      potentialMatches := potentialMatchLists[n_S]
      n_T := potentialMatches[0]
      topCandidateSimilarity := similarityValues[n_S, n_T]

      if topCandidateSimilarity >= (TH_tc) then
         averageSimilarity := 0

         for i := 1 to α do
            averageSimilarity += similarityValues[potentialMatches[i]]
         end for
         averageSimilarity := averageSimilarity / α

         if averageSimilarity >= (TH_df) then
            matchings := matchings ∪ (n_S, n_T)
         end if
      end if
   end for

   return matchings
end procedure
```

Figure 3.7: Algorithm to compute matchings

3.4 Evaluation Criteria

To evaluate the effectiveness of our approach, we run it over a series of data sets and analyze the results obtained, looking at their correctness and completeness. The parameter matching techniques presented in the previous section take two Web service operations as input and produce a list of suggested mappings between their parameter sets. Previously to running the experiment on a data set, we manually identified the semantically equivalent parameters between the two services involved —creating a list of correct element mappings— to be able to compare the algorithm's results against the expected ones. We then check how many of these computed mappings are in the list of correct mappings —these are called the *true positives*. Conversely, the computed mappings that do not occur in the list of manually identified mappings are the *false positives*. Finally, the source nodes for which no match has been found are also checked against the correct mappings. The ones that have in fact a correct mapping are our *false negatives*. With these three values, we can compute three measures: precision, recall and F-measure, as shown below [31].

$$precision = \frac{tp}{tp + fp}$$

$$recall = \frac{tp}{tp + fn}$$

$$F - measure = \frac{2}{\frac{1}{precision} + \frac{1}{recall}}$$

Precision measures the correctness of the results. It is the rate of true positives against the total matches returned by the algorithm, i.e. among all the suggested matching parameters, how many are correct. Recall, on the other hand, is a measure of the completeness of the results. This measure is computed as the ratio of true positives against the total correct matches, i.e. among all the actual matching parameters, how many have been found by the algorithm. Usually, there is a trade off between precision and recall. This is why, the F-measure is defined to combines both of these metrics in a single value by computing their harmonic mean. As in the case of precision and recall, F-measure is in the (0, 1) range, with higher values meaning better results.

3.5 Data Collection

In order to test our method for service parameter matching, we have compiled a series of sample data sets for both real-world and synthetic cases. Each of these data sets is a pair of Web service definitions, which have not been designed to work together. However, each service in the pair provides an operation that can be combined with an operation provided by its counterpart.

A real-world example using the software from our industry partner Mecanica Solutions is taken from their employee time sheet solution, which has an API based on

SOAP Web services. We obtained the WSDL document corresponding to the definition of this service from the developers at Mecanica. This document includes the definition of an operation called *OpenTSExecute* which is defined in a way so that it can be used for a variety of actions on time sheet data. Instead of providing an individual service operation for each type of action, this standard operation includes an input parameter called *method* which is used to identify the type of action to be performed. To allow for this operation to be flexible, its parameters include all the data fields associated with time sheet information, as well as extra generic fields that can be used when an action needs to handle some unforeseen information. As a matching counterpart we selected an operation called *Find Employee Time Sheet By Employee* provided by SAP. The latter's online documentation includes specifications for their Web service-based APIs, from which we acquired the corresponding WSDL document [40]. This case study is based on a real world situation —the task of importing time sheet information from SAP into Mecanica's 360e.

As a synthetic example, we defined a pair of sample services based on Amalgam, a schema and data integration benchmark suite developed at the University of Toronto [25]. This suite contains a set of schemata of bibliographic databases. We used two of these schemata as a basis for two library operations called *getBookByAuthor* and *addBook*.

Two more case studies were defined in the field of a parts supplier business — for example for computer, or household appliances parts. To build the interfaces of these web services, we took the names for the source operations' elements from the TPC-H relational database schema, which has tables for orders, parts, suppliers, and customers [7]. To define the interfaces of the target services we used a parts, distributors and orders data model freely available online [49]. Although the two pairs of services are in the same domain, one of them deals with parts suppliers, with the source service offering a *getSupplier* operation and the target service offering its

counterpart called *addSupplier*. The second pair of services deals with orders of parts, and the operations being integrated are *getOrder* and *placeOrder*.

3.6 Support Tool

To make the techniques described throughout this chapter available to developers working in service integration, an interactive tool which implements them is needed. We have identified the following requirements for this tool:

- Guide the developer along the process of entering the necessary information about the services and the additional data used in the integration process.
- Provide a graphical view of the interfaces of the services being integrated, along with the mapping between elements in these interfaces.
- Since the automated methods cannot guarantee 100 percent efficacy, allow for manual input and adjustments to the computed results.

As part of this work, we have built a support tool that provides these features. The next chapter explores its details more closely.

Chapter 4

Tool Design and Experiments

This chapter is divided into two main sections. The first one includes a description of the service integration tool we have developed. In the second portion, we describe the experiments conducted to test our method.

4.1 The Service Integration Assistant

As part of the work for this thesis, we have developed a proof of concept tool aimed at alleviating the task of service integration. Specifically, this tool applies the concepts presented in the previous chapter to aid developers in the finding of semantically equivalent parameters across two SOAP Web services. The user interface for this service integration assistant follows a wizard style which guides the user through the steps of the process.

4.1.1 Building of Schema Trees

In the first step of the wizard, the user is prompted for the location of the WSDL documents that define the Web services involved. Once this is done, the system scans the files to find the available operations. In case a file defines more than one operation,

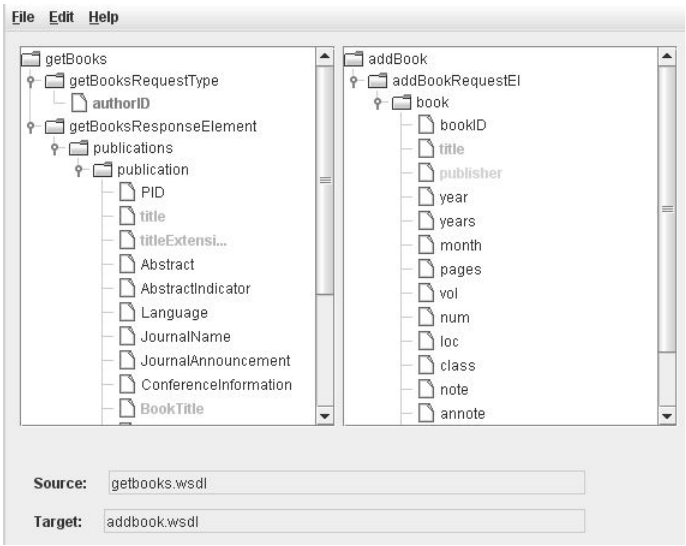

Figure 4.1: Schema trees on the service integration assistant

these are presented for the user to select the desired one. For each of the two Web service operations, its parameter structure is loaded, and a schema tree is constructed to represent it. Figure 4.1 shows the tool's main window with two schema trees in it.

4.1.2 Removal of Stop Words

As mentioned in section 3.3, stop words should be filtered out from the schema trees in order to only consider relevant terms during the matching process. The wizard will prompt the user to provide a list of stop words. Ideally, this list will be a domain-specific one because generic lists can contain terms that should not be considered as stop words in the context of the services in question. Regardless of the kind of list provided, the tool will in a later step give the user a chance to validate it before it is used, as shown in Figure 4.2. Once the stop word list is loaded, the schema trees will be scanned and a new list will be constructed. We will call this the list of potential stop words and it will take three steps to build it. First, the terms that appear both in the trees and in the provided stop word list will be added.

Proposition 4.1.1 *Terms appearing at the root of the tree are likely to describe a*

general concept —such as the purpose of the service itself— rather than a more specific detail.

Following proposition 4.1.1, the terms that constitute the name of the root node of each tree will be added to our set of potential stop words. The set will then be presented to the user, who will mark the words that should be validated as stop words in the context of the particular services being integrated. Then, as a last step the user can manually append additional stop words to the list.

Having now a set of stop words, the system scans the schema trees, removing any stop words that occur inside node names. It is important to do this filtering of stop words at this point in order for the words not to be considered in any of the following steps of the process.

Example 4.1.1 *Figure 4.3 illustrates a possible structure for the services offered by a city. The Service Integration Assistant will suggest a list of stop words* $SW = \{$ *"city", "services", "and"*$\}$*, where the terms "city" and "services" are obtained from the root node, while the term "and" comes from the provided list of stop words.*

4.1.3 Node Filtering and Addition of Synonyms

Once stop words have been removed, the tool will help overcome two issues: a) the presence of concepts that are described using different terms by the different services, and b) the presence of concepts in one service that do not exist in the other one. This will be done by first counting the frequency in which each term appears in the schema trees. This count will allow us to generate the list of terms that occur the most frequently in a tree, but do not exist in the other one.

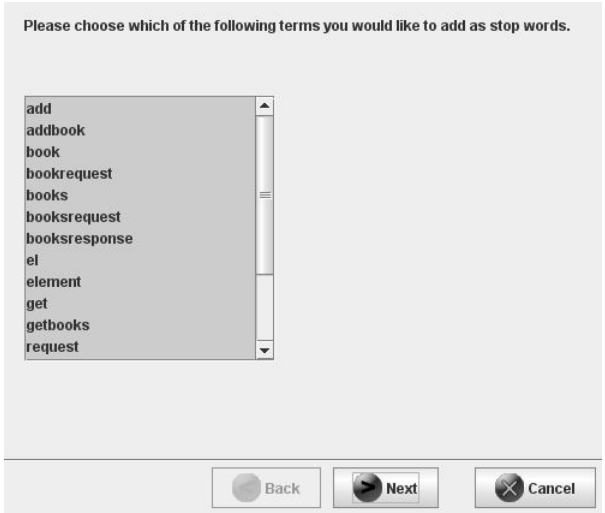

Figure 4.2: Suggested stop words on the service integration assistant

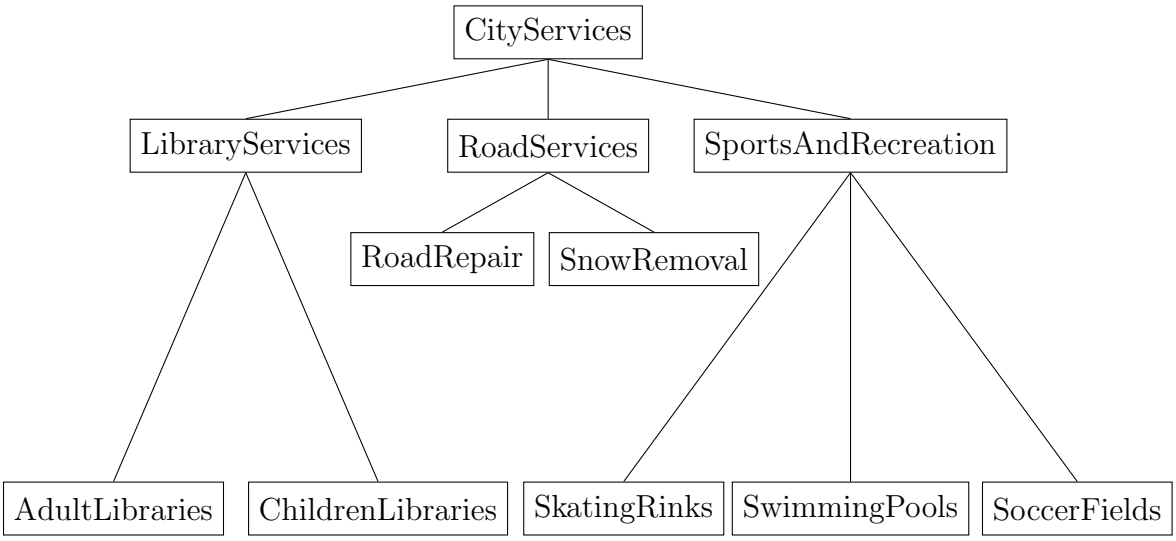

Figure 4.3: City services schema tree

Proposition 4.1.2 *Let T and R be two schema trees, and $N = \{t_1, t_2, ..., t_n\}$, $M = \{r_1, r_2, ..., r_m\}$ their respective sets of terms. If $t_a \in T, t_a \notin R$ we can say that either:*

- *t_a represents a concept that exists in* T *but does not exist in* R*, or*
- *t_a represents a concept that exists in both* T *and* R *but the latter uses a different term to refer to this concept.*

These terms will be presented to the user in two separate tables, one for each tree —sorted by term frequency in descending order. For each of the terms, the user can choose to perform two actions. First, the term can be added to a list that will be used to filter non-matchable concepts. Nodes that contain terms that appear in that list will be ignored by the matching algorithms. Alternatively, the user can choose to enter a list of synonyms for the term. These synonyms will be appended to the nodes containing the original term, aiding in the matching of concepts that are represented by different terms. Figure 4.6 illustrates this interaction. In the following step, the wizard offers the option of loading a thesaurus file, providing additional synonyms for the schema trees' terms.

Example 4.1.2 *Figures 4.4 and 4.5 show two different schema trees representing a university course. The term "code" occurs frequently in the second tree but is not present in the first tree. Therefore, this term will be presented to the user, who will be able to decide if the nodes containing it should be ignored or synonyms should be added for it. In this case, the term "id" is an appropriate synonym to be used.*

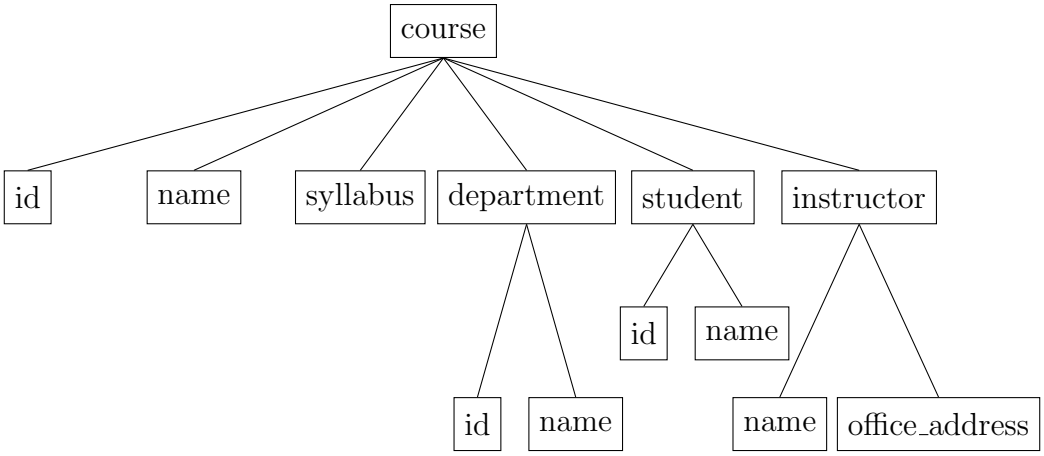

Figure 4.4: A university course schema tree

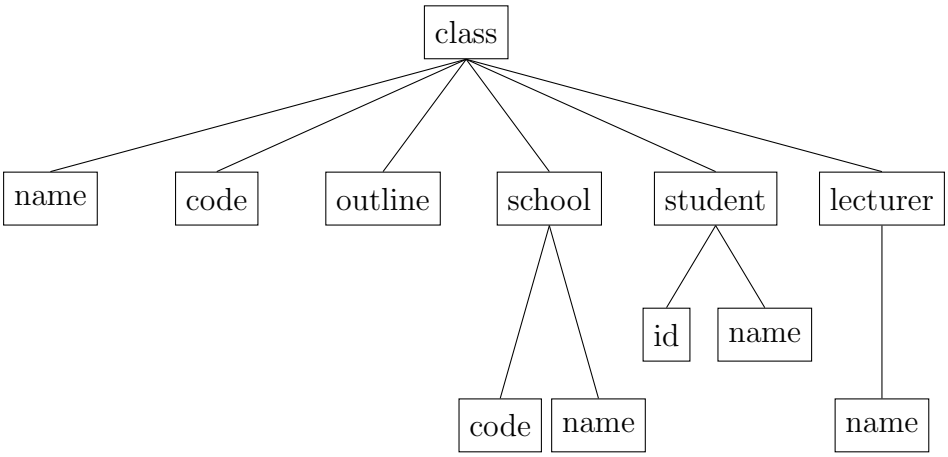

Figure 4.5: A different university course schema tree

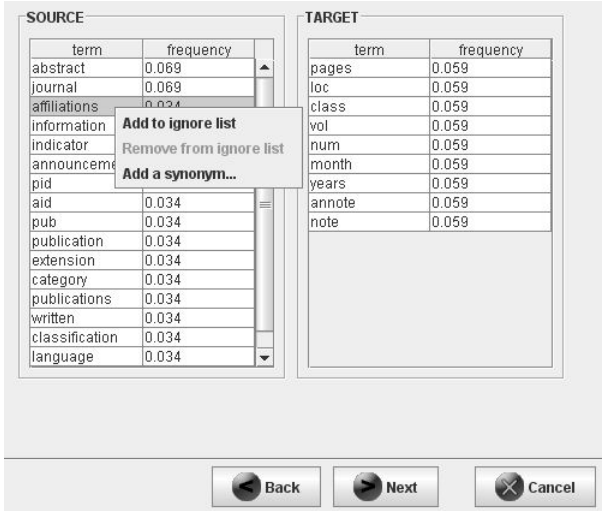

Figure 4.6: Lists of exclusive terms sorted by the frequency in which they appear

4.1.4 Computing Similarity and Obtaining Matching Pairs

With the schema trees built and cleaned up, the data is ready to be run through the similarity computations. Following the techniques described in section 3.3, linguistic and structure similarity values are computed and combined for the pairs of nodes from the source and target trees, producing a set of source nodes along with a list of best matching candidate target nodes for each of them. To determine which of these candidates to present to the user, a set of values is needed:

- The top candidate threshold TH_{tc}, which is the minimum similarity value that a best matching candidate must have to be considered.

- The number of top candidates to consider α, which specifies, for each source node, how many of its top matching candidates will be compared to its best matching candidate.

- The difference threshold TH_{df}, which is the minimum difference between the best candidate's similarity value and the average similarity of the α top candidates, for a matching candidate to be considered.

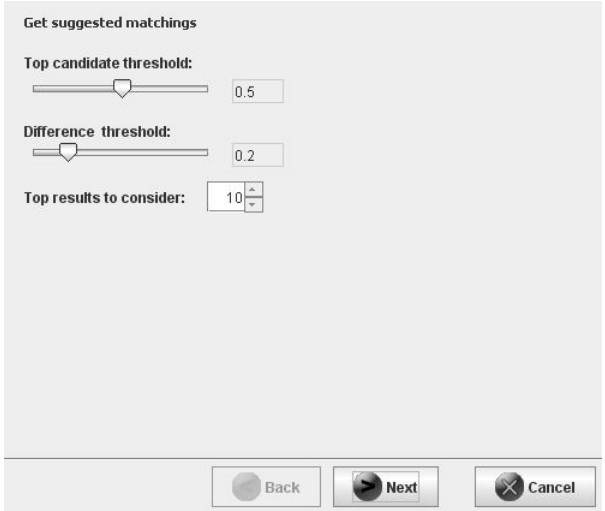

Figure 4.7: Service integration assistant prompts for threshold values

The wizard prompts the user for these three values on the screen shown in Figure 4.7, and feeds them to the algorithm that produces the matching candidates. The latter are then showed to the user using colour codes to mark the matching tree nodes.

Once the software has presented the suggested matchings, the user can review them on the tree components, and export them to an XML file to be used in later steps of service integration. Additionally, the tool allows mappings to be manually added or removed by the developers in case the proposed results need tweaking.

4.2 Experiments

To evaluate our method, we have run tests on the series of sample data described in Section 3.5. The test process accepts the following information as input:

- A pair of WSDL documents describing the source and target operations.
- A thesaurus file containing the domain-specific synonyms used.
- A file containing the pre-computed matching parameters for the pair of operations.

- A list of non-matchable concept terms.
- A list of stop words.
- The value of the top candidate threshold.
- The value of the difference threshold.
- The number of next best candidates used for comparing their average value to the top candidate's value.

After a test is run, the following output is produced:

- The number of true positives, false positives, and false negatives.
- The values of precision, recall and F-measures.

All the results presented in the following sections have been obtained by running an implementation of the technique described in section 3.3. The lists of stop words used were obtained using the method described in section 4.1.2. The lists of non-matchable terms and the thesauri were built by inspecting the sets of exclusive terms described in section 4.1.3 —each term was either added to the non-matchable list, or looked up for synonyms in a thesaurus. It is also important to mention that, even though our Service Integration Assistant allows manual adjustments, the tests results presented are the ones produced directly by the tool, and have not been manually corrected.

4.2.1 Data Sets

Using the data sets we have compiled, we have run the experiments to evaluate the performance of our method. Our main data set is the pair of employee time sheet operations from SAP and Mecanica 360e. The additional data sets are synthetic examples of service integration in the domains of parts suppliers and orders, and

academic bibliographies. We have created these using the names of fields in the TPC-H database schema for parts and suppliers [7], a free parts and distributors data model [49] and the amalgam project [25] respectively. Table 4.1 shows, for each data set, the number of nodes in the source and target services as well as the number of matching pairs of nodes manually identified.

	Suppliers	Biblio	Orders	SAP - 360e
Nodes (source / target)	15 / 19	18 / 31	19 / 50	157 / 250
Matching pairs	7	8	12	17

Table 4.1: Data sets

4.2.2 Effect of Combining Parent Node Names

In this section we show the effect of combining the node tokens with the ones from its parent node. For each of the data sets, we have run the process three times:

- One without combining the tokens, so that a node includes only the terms from its own name.
- A second time adding to a node the group of tokens from the whole path starting at the root of the tree.
- A third time adding to a node the tokens from its parent's name.

Table 4.2 lists the results from these experiments, which show that the best results are obtained when combining a node's tokens with the ones from its parent.

	TP (nodes)	FP (nodes)	FN (nodes)	Prec.	Rec.	F-meas.
Suppliers						
Not combining	2	0	5	1	0.2857	0.4444
Combining all	2	0	5	1	0.2857	0.4444
Combining parent	5	2	2	0.7143	0.7143	0.7143
Biblio						
Not combining	5	2	3	0.714286	0.625	0.6667
Combining all	0	1	8	0	0	0
Combining parent	5	2	3	0.714286	0.625	0.6667
Orders						
Not combining	3	0	9	1	0.25	0.4
Combining all	2	0	10	1	0.1667	0.2857
Combining parent	4	0	8	1	0.3334	0.5
SAP - 360e						
Not combining	4	13	13	0.2353	0.2353	0.2353
Combining all	1	0	16	1	0.0588	0.1112
Combining parent	7	18	10	0.28	0.4118	0.3334

Table 4.2: Results from testing the effect of combining parent node names

4.2.3 Effect of Two Threshold Selection Method

Here we present a comparison between two methods for the selection of results. In the first case, a single threshold value is used. If a candidate pair's similarity exceeds this value, it is considered a positive result. For the second case, a second threshold is added —to check that a candidate pair's similarity value is considerably higher than the one of the next best candidates. Table 4.3 compares the results obtained using each of these methods. We can see that the two-threshold method either outperforms or ties with the one-threshold method.

	TP (nodes)	FP (nodes)	FN (nodes)	Prec.	Rec.	F-meas.
Suppliers						
One threshold	5	2	2	0.7143	0.7143	0.7143
Two thresholds	5	2	2	0.7143	0.7143	0.7143
Biblio						
One threshold	5	2	3	0.7143	0.6250	0.6667
Two thresholds	5	2	3	0.7143	0.6250	0.6667
Orders						
One threshold	4	2	8	0.6667	0.3333	0.4444
Two thresholds	4	0	8	1.0000	0.3333	0.5
SAP - 360e						
One threshold	8	26	9	0.2353	0.4706	0.3137
Two thresholds	7	18	10	0.2800	0.4118	0.3334

Table 4.3: Results from comparing two matching pairs selection methods

4.2.4 Effect of Auxiliary Data

For each of the data sets, we have run our service parameter matching process multiple times, alternately supplying it or failing to supply it with lists of stop words and terms to ignore. To build these lists we have used the techniques implemented in our service integration tool, as explained in section 4.1. Specifically, the stop words have been chosen from the collection of terms occurring at the root of the tree, whereas the ignored terms come from the sets of most frequent terms that appear exclusively in one of the trees. We have found that the results get consistently better —or in the worst case stay unchanged— as the algorithm is provided with more of this kind of auxiliary information. Particularly, this data helps in the trimming of false positives from the results. The following examples offer closer looks at each of the case studies, each one showing the different results obtained by running experiments with varying

amounts of auxiliary information provided. In all cases, the tables show the number of true positives, false positives and false negatives (in number of nodes), as well as the values of precision, recall and F-measure.

Example 4.2.1 *For the parts suppliers services, we notice that adding either the list of terms to ignore or the list of stop words increases the number of true positives, with no additional gain from using them both at the same time.*

	TP (nodes)	FP (nodes)	FN (nodes)	Prec.	Rec.	F-meas.
No stop words No ignored terms	4	2	3	0.6667	0.5714	0.6154
With stop words No ignored terms	5	2	2	0.7143	0.7143	0.7143
No stop words With ignored terms	5	2	2	0.7143	0.7143	0.7143
With stop words With ignored terms	5	2	2	0.7143	0.7143	0.7143

Table 4.4: Experiment results for parts suppliers services

Example 4.2.2 *For the academic bibliographies services, we notice that neither the list of terms to ignore nor the the list of stop words have an effect on the results.*

	TP (nodes)	FP (nodes)	FN (nodes)	Prec.	Rec.	F-meas.
No stop words No ignored terms	5	2	3	0.7143	0.625	0.6667
With stop words No ignored terms	5	2	3	0.7143	0.625	0.6667
No stop words With ignored terms	5	2	3	0.7143	0.625	0.6667
With stop words With ignored terms	5	2	3	0.7143	0.625	0.6667

Table 4.5: Experiment results for academic bibliographies services

Example 4.2.3 *As in the case of the academic bibliographies services, the parts orders results were not affected by the introduction of stop words. However, the supplying of terms to ignore reduces the number of false positives.*

	TP (nodes)	FP (nodes)	FN (nodes)	Prec.	Rec.	F-meas.
No stop words No ignored terms	4	1	8	0.8	0.3334	0.4706
With stop words no ignored terms	4	1	8	0.8	0.3334	0.4706
No stop words With ignored terms	4	0	8	1	0.3334	0.5
With stop words With ignored terms	4	0	8	1	0.3334	0.5

Table 4.6: Experiment results for parts orders services

Example 4.2.4 *In the case of SAP and Mecanica 360e, we can see that both the list of stop words and the terms to ignore have an individual positive impact on the results of the experiment. Moreover, the results are best when both of them are supplied.*

	TP (nodes)	FP (nodes)	FN (nodes)	Prec.	Rec.	F-meas.
No stop words No ignored terms	7	24	10	0.2258	0.4118	0.2917
With stop words No ignored terms	7	20	10	0.2593	0.4118	0.3182
No stop words With ignored terms	7	21	10	0.25	0.4118	0.3112
With stop words With ignored terms	7	18	10	0.28	0.4118	0.3334

Table 4.7: Experiment results for SAP and 360e timesheet services

4.2.5 Test Environment and Efficiency

We have also timed the execution of the algorithm to test its efficiency. As expected, the SAP - 360e data set takes longer to execute because of its larger number of nodes. Table 4.8 shows the time (in seconds) taken by each of the data sets. For these experiments, the lists of stop words and terms to ignore were supplied in all cases. The following platform was used for the tests:

- CPU: Intel Core 2 Duo 3.00 GHz
- RAM: 2.0 GB DDR PC-6400
- OS: Windows XP Professional SP2
- Java: 1.6.0.32

	Suppliers	Biblio	Orders	SAP - 360e
Number of nodes	34	49	69	407
Tree building	0.063	0.015	0.078	1.234
Matching process	0.062	0.11	0.157	15.828

Table 4.8: Execution time by data set

4.2.6 Comparison with COMA++

The COMA++ system is a generic schema matching tool developed at the University of Leipzig, aimed at different schema formats and application domains. Its approach consists of a combination of multiple matching techniques —including both instance and schema-level approaches— based on the linguistic and structural similarity of elements. The researchers behind the development of COMA++ have made available a prototype implementation which includes a graphical user interface [21]. We have run this tool on our case studies to compare the results. Since COMA++ is a generic schema matching tool, it does not support WSDL documents as an input format. However, it does include support for XSD files, which has allowed us to use XSD documents extracted from the *types* section in each of our WSDL files as input. For these experiments, the same thesauri were used with our method and COMA++.

Table 4.9 show the results obtained for each of the data sets, using both our technique and COMA++. In three of the four cases, the outcomes from our approach are better than the ones from COMA++. This works as evidence that our contributed methods provide an improvement when working with the matching of Web services, since they are specifically directed toward the features of the WSDL format —including the presence of stop words, the lack of a formal ontology, and the occurrence of concepts that are only present in one of the services being matched. There

	TP (nodes)	FP (nodes)	FN (nodes)	Prec.	Rec.	F-meas.
Suppliers						
Our method	5	2	2	0.7143	0.7143	0.7143
COMA++	2	3	5	0.4	0.2857	0.3334
Biblio						
Our method	5	2	3	0.7143	0.625	0.6667
COMA++	5	1	3	0.8334	0.625	0.7143
Orders						
Our method	4	0	8	1	0.3334	0.5
COMA++	4	4	8	0.5	0.3334	0.4
SAP - 360e						
Our method	7	18	10	0.28	0.4118	0.3334
COMA++	1	32	16	0.0303	0.0588	0.04

Table 4.9: Comparing results from our technique and COMA++

is one case where COMA++ delivers slightly better results. Being a generic schema matcher, COMA++ differs from our Web service oriented approach in a number of items. For instance, it does not include removal of stop words from elements' token sets nor does it allow ignoring of elements containing specific non-matchable terms. In our experiment results shown in section 4.2.4 where we measure the effect of these kinds of auxiliary data, we can see that the case of academic bibliographies is not affected by the presence of auxiliary data. This case is the same one where the results from COMA++ are slightly better than the ones given by our technique, therefore showing that our method's positive impact is more important when used on data sets that include stop words or unmatchable terms.

4.3 Concluding Remarks

As we have seen in the case studies presented in the previous sections, mapping the equivalent elements between the interfaces of services not designed to work together is not a trivial task. In fact, a pair of services often includes elements that are not matchable at all —when an element in one service describes a concept that does not exist in the other one. For instance, in our real-world example of employee time sheet operations, our source tree consists of 157 nodes, of which only 17 can be mapped to nodes in the target tree. This does not mean that the services cannot be integrated to work together —it only shows that the information that can be shared between services is only a small portion of the whole data set handled by them. Among these parameters that can be mapped, our tool was able to identify slightly over 40 percent, with the value of the F-measure being 0.3334. This is better than the results obtained using a generic schema matching tool. However, the manual editing feature of our service integration assistant is still necessary so that the rest of the mappings can be added.

Chapter 5

Conclusions

The growing popularity that SOA has seen as a choice of software development paradigm among businesses increases the importance of research on the subject. In particular, the issue of automated service integration has received close attention because of the potential gains that could be obtained from it. In spite of this increased interest, it remains a complex subject that requires further work. The open-standard XML-based nature of Web services results in a technology that is easily adapted by the industry. In addition, it provides a standard syntax that is independent of underlying platforms, greatly facilitating system integration. In spite of this, the lack of a semantic standard for service definitions is a major obstacle to the automation of this integration.

In this thesis we have offered a look into the major issues of the automated service integration problem. We have discussed the steps in the development of an adaptor to mediate between two services. Having identified the issue of finding semantically equivalent elements across service definitions as a major step toward adaptor construction, we have focused on the development of techniques to automate this. Our methods leverage schema matching techniques to take advantage of the XSD format of Web service operation parameters, while at the same time expanding them to tackle

specific features of WSDL —namely, the lack of formal ontologies, highly context dependent meaning of terms, randomness in the schema structures and cardinality of elements and data types, and the presence of non-matchable terms. We have built a prototype of an interactive tool which implements our techniques and tested this on several data samples. The latter include the real-world case study of the integration of time sheet information between Mecanica Solutions' 360e and SAP's ERP software. The results from our experiments look promising, comparing positively against the ones obtained by using a general-purpose schema matching tool. Moreover, our tool provides developers with the option of manually adjusting the results, so that mappings not automatically discovered can be added as well.

The method proposed in this thesis examines WSDL documents to determine a mapping between service interfaces. We have taken the decision of using only this information since message instances —from service invocations and returned results— are not always available during the adaptor development process. However, we acknowledge the fact that in the cases where this data is available, it can help significantly in the matching process. Therefore, as future research, we will work on extending our current method with techniques that exploit this information, with the goal of improving the efficacy of the automatic interface matching. Additionally, when documentation is available on the service definitions, this information can also be inspected to further deduce element mappings.

Moreover, the main focus of the method we have developed in this thesis is the identification of semantically equivalent elements in service interfaces. The results of this operation can then be used in the building of a script to transform instances of messages between the formats of the services involved. In future work, we will concentrate our efforts on the automation of this task, which will bring us closer to the goal of an automated service integration solution.

Bibliography

[1] L. Ai. *QoS-aware Web service composition using genetic algorithms*. PhD thesis, Queensland University of Technology, 2011.

[2] A. Algergawy, E. Schallehn, and G. Saake. Improving XML schema matching performance using Prüfer sequences. *Data Knowl. Eng.*, pages 728–747, 2009.

[3] A. Algergawy, E. Schallehn, and G. Saake. A sequence-based ontology matching approach, 2009.

[4] A. Bansal, S. Kona, L. Simon, and T. D. Hite. A universal service-semantics description language. In *Proceedings of the Third European Conference on Web Services*, ECOWS '05, pages 214–, Washington, DC, USA, 2005. IEEE Computer Society.

[5] D. Beckett. RDF/XML syntax specification. `http://www.w3.org/TR/REC-rdf-syntax/`, 2004.

[6] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IIWeb*, pages 73–78, 2003.

[7] T. P. P. Council. Tpc-h benchmark. `http://www.tpc.org/tpch`, 2011.

[8] H.-H. Do and E. Rahm. COMA: a system for flexible combination of schema matching approaches. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 610–621. VLDB Endowment, 2002.

[9] E. Dragut, F. Fang, P. Sistla, C. Yu, and W. Meng. Stop word and related problems in web interface integration. *Proc. VLDB Endow.*, 2:349–360, August 2009.

[10] D. Engmann and S. Mamann. Instance matching with COMA++. In *BTW Workshops'07*, pages 28–37, 2007.

[11] G. M. Erik Christensen, Francisco Curbera and S. Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, 2001.

[12] D. M. et al. OWL-S: Semantic markup for Web services. `http://www.w3.org/Submission/OWL-S/`, 2004.

[13] R. F. et al. Hypertext transfer protocol – HTTP/1.1. `http://www.w3.org/Protocols/rfc2616/rfc2616.html`, 1999.

[14] W. O. W. Group. OWL 2 Web Ontology Language. `http://www.w3.org/TR/owl2-overview/`, 2009.

[15] M. J. Hadley. Web application description language (WADL). *Search*, 12(TR-2006-153):19, 2006.

[16] java.net. Java API for XML Web services. `http://jax-ws.java.net/`, 2008.

[17] G. Kondrak. N-gram similarity and distance. In *Proc. Twelfth Intl Conf. on String Processing and Information Retrieval*, pages 115–126, 2005.

[18] W. Kongdenfha, H. R. Motahari-Nezhad, B. Benatallah, F. Casati, and R. Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of Web service adapters. *IEEE Trans. Serv. Comput.*, 2(2):94–107, Apr. 2009.

[19] lextek. Default stop word list. `http://www.lextek.com/manuals/onix/stopwords1.html`.

[20] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 49–58, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[21] S. Mamann, S. Raunich, D. Aumller, P. Arnold, and E. Rahm. Evolution of the coma match system. In P. Shvaiko, J. Euzenat, T. Heath, C. Quix, M. Mao, and I. F. Cruz, editors, *OM*, volume 814 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[22] D. V. Mcdermott. Estimated-Regression Planning for Interactions with Web Services. In M. Ghallab, J. Hertzberg, and P. Traverso, editors, *AIPS'02: Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*, pages 204–211, Toulouse, France, Apr. 2002. AAAI.

[23] Mecanica. Mecanica Solutions 360 enterprise website. `http://www.360enterprisesoftware.com`, 2012.

[24] G. A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, Nov. 1995.

[25] R. J. Miller, D. Fisla, M. Huang, D. Kymlicka, F. Ku, and V. Lee. The Amalgam Schema and Data Integration Test Suite. www.cs.toronto.edu/ miller/amalgam, 2001.

[26] D. Milne, O. Medelyan, and I. H. Witten. Mining domain-specific thesauri from wikipedia: A case study. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, WI '06, pages 442–448, Washington, DC, USA, 2006. IEEE Computer Society.

[27] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *Proceedings of the 16th*

international conference on World Wide Web, WWW '07, pages 993–1002, New York, NY, USA, 2007. ACM.

[28] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, Mar. 2001.

[29] S.-C. Oh, D. Lee, and S. R. T. Kumara. A comparative illustration of ai planning-based Web services composition. *SIGecom Exch.*, 5(5):1–10, Jan. 2006.

[30] S.-C. Oh, B.-W. On, E. J. Larson, and D. Lee. Bf*: Web services discovery and composition as graph search problem. In *In Proceedings of IEEE EEE, Hong Kong*, pages 784–786, 2005.

[31] D. L. Olson and D. Delen. *Advanced Data Mining Techniques*. Springer, 2008.

[32] Oracle. JD Edwards website. `http://www.oracle.com/us/products/applications/jd-edwards-enterpriseone/index.html`, 2012.

[33] Oracle. Peoplesoft website. `http://www.oracle.com/us/products/applications/peoplesoft-enterprise/index.html`, 2012.

[34] M. P. Papazoglou. *Web Services: Principles and Technology*. Pearson, 2008.

[35] K. P. Paul V. Biron and A. Malhotra. XML schema part 2: Datatypes second edition. `http://www.w3.org/TR/xmlschema-2`, 2004.

[36] K. P. Paul V. Biron and A. Malhotra. SOAP version 1.2 part 0: Primer (second edition). `http://www.w3.org/TR/soap12-part0/`, 2007.

[37] S. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for Web service composition. In *Proceedings of the 11th International WWW Conference (WWW2002)*, Honolulu, HI, USA, 2002.

[38] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, Dec. 2001.

[39] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, 2007.

[40] SAP. Employeetimesheetbyemployee WSDL. `http://esworkplace.sap.com/socoview(bD1lbiZjPTAwMSZkPW1pbg==)/render.asp?packageid=DE0426DD9B0249F19515001A64D3F462&id=54163F5B5FE311DA36BB000F20DAC9EF_WSDL`, 2012.

[41] SAP. SAP website. `http://www.sap.com`, 2012.

[42] Z. Shan, A. Kumar, and P. Grefen. Towards integrated service adaptation a new approach combining message and control flow adaptation. In *Proceedings of the 2010 IEEE International Conference on Web Services*, ICWS '10, pages 385–392, Washington, DC, USA, 2010. IEEE Computer Society.

[43] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. In S. Spaccapietra, editor, *Journal on Data Semantics IV*, volume 3730 of *Lecture Notes in Computer Science*, pages 146–171. Springer Berlin / Heidelberg, 2005.

[44] K. Sivashanmugam, K. Verma, A. Sheth, and J. Miller. Adding semantics to Web services standards, 2003.

[45] M. Strommer, F. Kromer, C. Pichler, and C. Huemer. Business document transformation using core components and XSLT. In *Proceedings of the 2011 IEEE 13th Conference on Commerce and Enterprise Computing*, CEC '11, pages 129–136, Washington, DC, USA, 2011. IEEE Computer Society.

[46] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated discovery, interaction and composition of semantic Web services. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1), 2011.

[47] WebserviceX.net. Geoipservice. http://www.webservicex.net/geoipservice.asmx, 2012.

[48] B. Williams. Agricultural information management standards. http://aims.fao.org/standards/agrovoc/about, 2011.

[49] B. Williams. Library of free data models. http://www.databaseanswers.org/data_models/, 2011.

[50] D. Wu, B. Parsia, E. Sirin, J. Hendler, , D. Nau, and D. Nau. Automating DAML-S Web services composition using SHOP2. In *In Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, 2003.

[51] Y. Yan, M. Chen, and Y. Yang. Anytime QoS optimization over the plangraph for web service composition. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1968–1975, New York, NY, USA, 2012. ACM.

[52] Y. Yan, P. Poizat, and L. Zhao. Repair vs. recomposition for broken service compositions. In P. P. Maglio, M. Weske, J. Yang, and M. Fantinato, editors, *ICSOC*, volume 6470 of *Lecture Notes in Computer Science*, pages 152–166, 2010.

[53] Y. Yan, P. Poizat, and L. Zhao. Repairing service compositions in a changing world. In R. Y. Lee, O. Ormandjieva, A. Abran, and C. Constantinides, editors, *SERA (selected papers)*, volume 296 of *Studies in Computational Intelligence*, pages 17–36. Springer, 2010.